KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Formal Reasoning about Hardware Capability Architectures

Thomas Van Strydonck

Supervisors:
Prof. dr. D. Devriese
Prof. dr. ir. F. Piessens

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2022

Formal Reasoning about Hardware Capability Architectures

Thomas VAN STRYDONCK

Examination committee:
Prof. dr. ir. B. Puers, chair
Prof. dr. D. Devriese, supervisor
Prof. dr. ir. F. Piessens, supervisor
Prof. dr. B. Jacobs
Prof. dr. M. Denecker
Dr. L. Daniel
Prof. dr. P. Sewell
(University of Cambridge, Cambridge, UK)
Prof. dr. D. Dreyer
(MPI-SWS, Saarbrücken, Germany)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2022

Uitgegeven in eigen beheer, Thomas Van Strydonck, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Preface

Many people contributed to this PhD in various ways, both directly in a technical sense and indirectly in an emotional one. I would like to spend some time here to acknowledge the people that helped me realize this thesis.

First, I want to thank my supervisors, Dominique and Frank, for making the following paragraph incredibly easy to write. Both Dominique and Frank are technically exceptionally strong researchers, whose expertise and insight helped me out tremendously throughout this doctorate. I am especially thankful to Dominique for almost literally dragging me over the first obstacles at the start of my research career, when I had no idea what I was doing. In spite of their busy agendas, both Dominique and Frank were always approachable, and made time whenever I needed it most. They taught me rigid and exact thinking, but also a healthy dose of pragmatism when it counted. I learned a lot during this PhD thanks to you two, and will be able to bank on this invaluable experience for years to come!

I am grateful to the local members of my jury; Bart Jacobs, Marc Denecker, Lesly-Ann Daniel, and the external jury members Peter Sewell and Derek Dreyer, for their willingness to spend their time on my dissertation, and for the helpful suggestions and feedback they provided. I thank Bob Puers for chairing the jury. More generally, I want to thank the reviewers I was assigned over the years for keeping me grounded, and for offering helpful directions for improvement and future research. My thanks go out to the Research Foundation – Flanders (FWO) for believing in my research project and supporting it financially. *I hold a PhD Fellowship from the Research Foundation - Flanders (FWO).*

Next, I want to thank my collaborators on the different research projects presented here. I am very grateful to Lars Birkedal for hosting me at Aarhus University for a five-month stay, and being very warm, welcoming and helpful, in spite of the difficult Covid circumstances. You made me feel right at home in Denmark! I also thank Aïna, Armaël, Alix, and Amin for the smooth and pleasant collaboration, showing me the Iris ropes and for accepting me despite the lack of A's in my initials. Thank you to the entire research group at Aarhus University for making the stay as enjoyable as it was. I extend my gratitude to David, Jen, and Leonardo from the University of Birmingham for their help and expertise during the collaboration on the TEE work. I am also thankful to Job for our collaboration on the same project; your professional approach to coding (and CPL projects! 😄) and easy-going nature were very pleasant to work with. Besides research, I also enjoyed some great collaborations as part of my teaching assistant duties. There, I was inspired by the competence and teaching skills of various peers: Roald, Ingmar, Simon, Andreas, ...

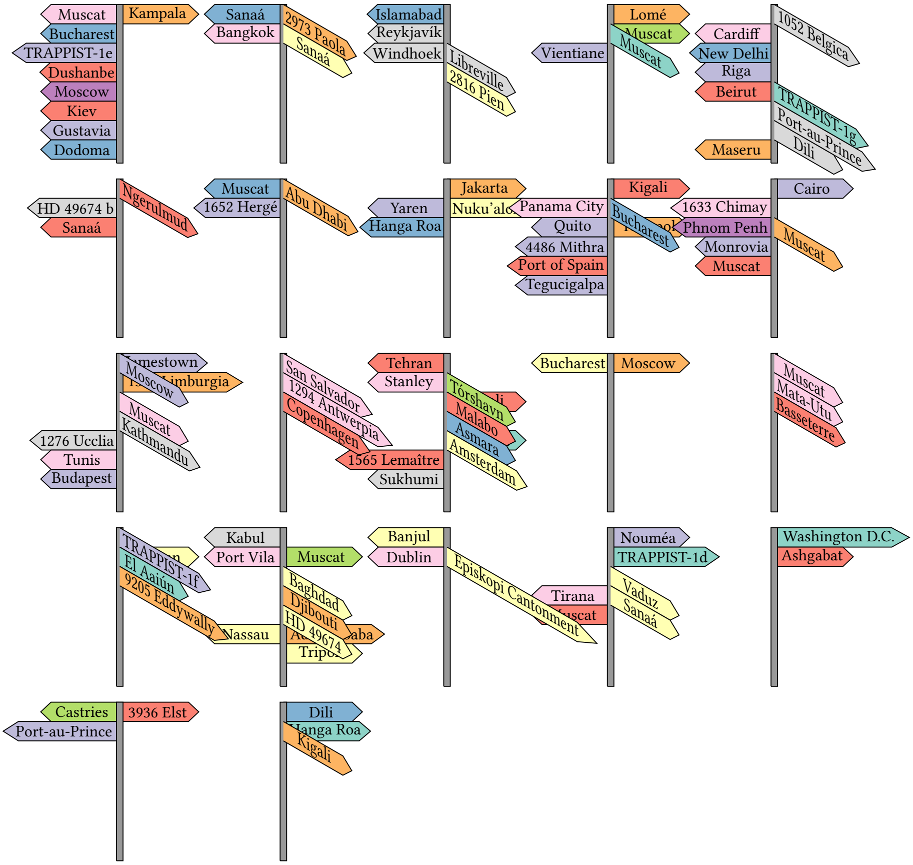
I want to thank my colleagues in Leuven for the years we spent together and the interesting discussions we had (both scientific and non-scientific). First and foremost, the 2.34-gang, whom I spent most time with; Kobe for teaching me about the value of noise-canceling headphones, but also sparking my interest into TEEs, Bob for showing me the programming ropes and bestowing me with the gift of his config files, Twinant for teaching me about typesetting (and allowing me to steal his fonts for this document), Majid for putting up with us, Matteo whose visit was short-lived but well-scented, and then my sports buddies, Yana and Stef from 2.34v2. I thank my new office mates of the 3.127; Andreas, Antoine, Elias, Emiel, Gilang, and Joris for the relaxed atmosphere and random developments throughout the day. I am indebted to the good people of the lunch group (which includes the aforementioned colleagues) for the enjoyable times we shared, as well as to some other colleagues for discussions that did not necessarily take place around a lunch table: Márton, Victor, Neline, Gertjan, Vincent, Jonathan, Hans, Koen, Steven, Sander, Annick, Kim, Vik, Katrien, Justus, JT, Nima, ... The DistriNet business office (Annick, Katrien, An) and the people of the secretariat perhaps deserve the largest gratitude, for keeping our entire operation rolling smoothly and always being helpful and responsive. Thank you!

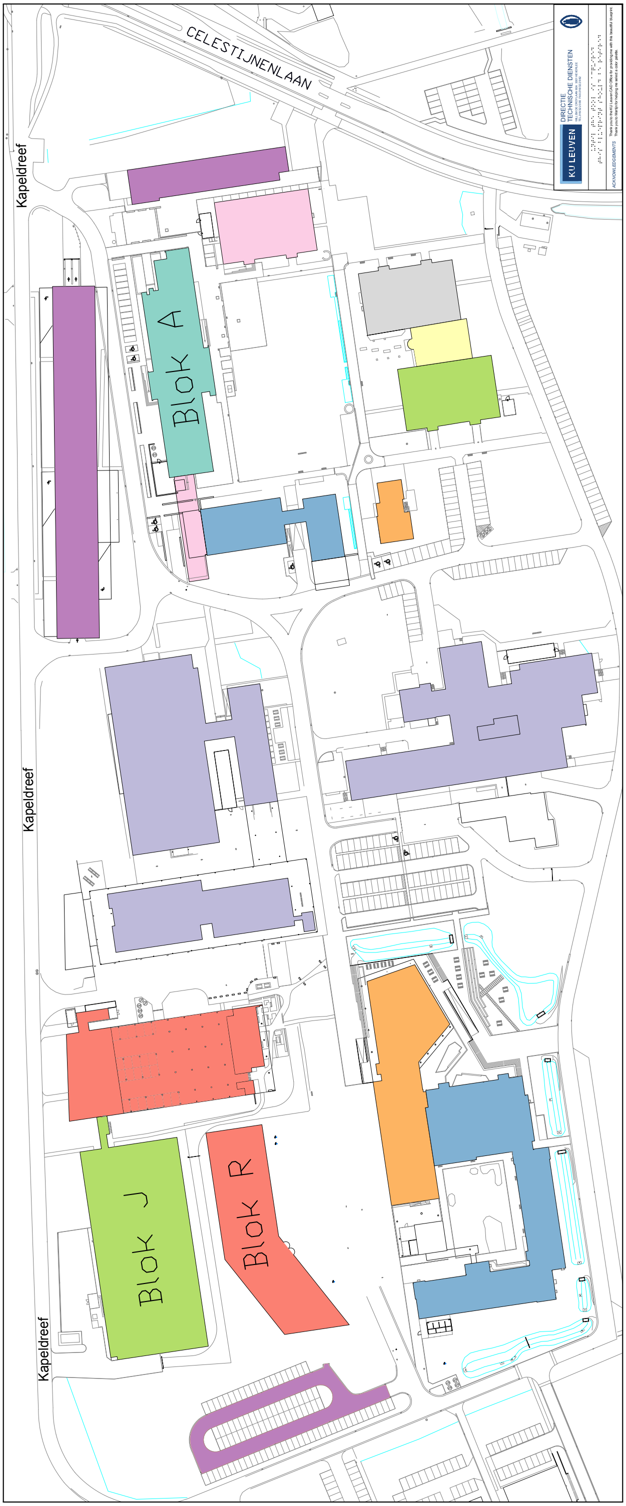
Lastly, I express immense gratitude to my family and friends: without the blissful moments we shared as a foundation, none of this thesis would have materialized. Special thanks to my mom for caring—I know you will be sleeping more soundly now that this is over ☺—and to both my parents for their invaluable advice, and unconditional love and support throughout. Quero agradecer especialmente à minha namorada, Cat, pelo seu amor e apoio; a tua presença na segunda parte do meu doutoramento tornou-a inesquecível!

Thomas Van Strydonck

P.S.: Thanks to Wessel and Annelies for proofreading the below ACROSTIC[1], and the thrilling ride that follows. In this context, I want to acknowledge LaTeX in general and TikZ in particular for showing me the limits of my patience and persistence.

[1] Oops, I did not mean to write Capitals here... Johnny Cash and Dolly Parton would be disappointed.


Muscat
Bucharest
TRAPPIST-1e
Dushanbe
Moscow
Kiev
Gustavia
Dodoma
Kampala
Sanaá
Bangkok
2973 Paola
Sanaá
Islamabad
Reykjavík
Windhoek
Libreville
2816 Pien
Vientiane
Lomé
Muscat
Muscat
Cardiff
New Delhi
Riga
Beirut
Maseru
1052 Belgica
TRAPPIST-1g
Port-au-Prince
Dili
HD 49674 b
Sanaá
Ngerulmud
Muscat
1652 Hergé
Abu Dhabi
Yaren
Hanga Roa
Jakarta
Nuku'alo
Panama City
Quito
4486 Mithra
Port of Spain
Tegucigalpa
Kigali
Bucharest
1633 Chimay
Phnom Penh
Monrovia
Muscat
Cairo
Muscat
mestown
Moscow
imburgia
Muscat
Kathmandu
1276 Ucclia
Tunis
Budapest
San Salvador
1294 Antwerpia
Copenhagen
1565 Lemaître
Sukhumi
Tehran
Stanley
Tórshavn
Malabo
Asmara
Amsterdam
Bucharest
Moscow
Muscat
Mata-Utu
Basseterre
TRAPPIST-1f
El Aaiún
9205 Eddywally
Nassau
Kabul
Port Vila
Muscat
Baghdad
Djibouti
HD 49674
aba
Tripo
Banjul
Dublin
Episkopi Cantonment
Tirana
uscat
Nouméa
TRAPPIST-1d
Vaduz
Sanaá
Washington D.C.
Ashgabat
Castries
Port-au-Prince
3936 Elst
Dili
anga Roa
Kigali



ACKNOWLEDGEMENTS

Thank you to the KU Leuven CAD Office for providing me with this beautiful blueprint.
Thank you to Martijn for helping me select a color palette.

Abstract

Our modern society increasingly relies upon computing devices for its proper functioning. With their increased presence, the number of interactions between different software components also increases. To secure this interaction and contain the impact of bug exploits (both malicious and accidental in nature), some form of security primitive to enforce the interface between different components is required.

In this dissertation, we study *hardware capabilities* as a possible solution to this problem. Capabilities are hardware-enforced, bounds-checked pointers that carry permissions. They are an assembly-level hardware primitive that allows for fine-grained spatial protection within components, and lightweight compartmentalization between components. They have been researched since the 60s, but have recently seen renewed interest thanks to the CHERI project at the University of Cambridge.

Given their low-level and flexible nature, writing correct and secure capability-manipulating programs is difficult. To remedy this problem, this thesis studies methods to reason *formally* about the security offered by capabilities. We capture these guarantees in a form of general security contract for a capability machine, which we call a *universal contract.* This universal contract describes how capabilities limit the power of *arbitrary, untrusted* code. Thus, it enables us to verify whether concrete application code satisfies certain properties of interest, even in the presence of arbitrary untrusted code.

In the first chapters of this thesis, we illustrate this approach in various settings. We introduce formal, mechanized models of different capability machines that leverage intra-language universal contracts to reason about concrete assembly code.

The first model, *Cerise*, illustrates how one can reason about the spatial protection and compartmentalization provided by a vanilla capability machine. To illustrate the model, we verify a counter that depends on encapsulation of private state and a heap-based calling convention that uses dynamic memory allocation. We also verify an example of dynamic sealing, illustrating that hardware capabilities are sufficiently powerful to implement design patterns from the object capability literature at the assembly level.

Next, support for effects in the form of *Memory-Mapped I/O* (*MMIO*) is added to the Cerise model. Compartmentalization offered by capabilities is used to build lightweight, nestable security wrappers around I/O devices. The wrappers enable layered enforcement of full-system safety properties on MMIO traces admitted by the capability machine. Interestingly, different layers of wrappers can be verified under different attacker models, and the results obtained by verifying layers closer to the hardware can be reused when verifying higher layers. This nesting and the ensuing opportunities for verification are difficult to achieve efficiently on commodity hardware. To illustrate the model, we verify two examples in a three-layer system of wrappers, where each layer reuses results from the previous one and considers a more refined attacker model.

The last instance of universal contracts we focus on is in efficiently supporting a form of *enclaved execution* on top of capability hardware; a combination that had not been achieved before. Enclaved execution is a popular mechanism for dynamically creating Trusted Execution Environments (TEEs) called enclaves. Enclaves are isolated execution contexts that protect integrity and confidentiality of software inside the enclave (even against compromised system software) and that support attestation. We demonstrate a novel, bottom-up design for flexible enclaves on top of a capability machine and present three different implementations of the design, providing preliminary performance benchmarks. Pinpointing the correct formal treatment of enclaved execution as a universal contract is ongoing work, and discussed in the conclusion chapter.

Reasoning about code at the assembly level is hard and error-prone. In the final content chapter, we therefore shift our focus from intra-language assembly-level reasoning to reasoning about secure compilation to capability architectures. The standard criterion of *full abstraction* is used to instantiate secure compilation, and a compiler from separation-logic-verified C-like code to C-like code with support for so-called linear capabilities (a non-duplicable type of capability) is proven fully abstract. The intuition behind the compiler is that the linear separation logic resources can be reified and represented by linear capabilities at the target level. The full abstraction proof roughly implies that attacks that are possible on the target-level capability architecture must have already been possible at the verified source level, i.e. the compiler did not create additional security issues not present at the source level. In the long run, the hope is to support a form of gradual verification, where security-critical parts of C codebases can be verified and compiled with this compiler, in such a way that security guarantees are upheld in the presence of buggy or untrusted code.

Beknopte samenvatting

Onze moderne samenleving boogt in toenemende mate op computerapparaten voor haar goede werking. Met hun verhoogde aanwezigheid neemt ook het aantal interacties tussen verschillende softwarecomponenten toe. Om deze interacties te beveiligen en de impact van bug-exploits (zowel kwaadaardig als accidenteel van aard) te beperken, is een soort beveiligingsprimitief vereist om de interface tussen verschillende componenten te handhaven.

In dit proefschrift bestuderen we *hardware capabilities* als mogelijke oplossing voor dit probleem. Capabilities zijn pointers die inherent permissies bevatten, waarvan de grenzen door de hardware afgedwongen worden. Ze zijn een hardwareprimitief op assembleertaalniveau dat voorziet in een fijnkorrelige spatiale bescherming binnen componenten, en een efficiënte compartimentering tussen componenten. Ze worden al sinds de jaren 60 onderzocht, maar verkregen recent hernieuwde belangstelling dankzij het CHERI project aan de Universiteit van Cambridge.

Gezien hun weinig abstracte en flexibele karakter is het moeilijk om correcte en veilige programma's te schrijven die capabilities manipuleren. Om dit probleem te verhelpen bestudeert dit proefschrift methoden om *formeel* te redeneren over de beveiliging die geboden wordt door capabilities. We leggen deze garanties vast in een vorm van algemeen veiligheidscontract voor een capability machine dat we een *universeel contract* noemen. Dit universele contract beschrijft hoe capabilities de macht van *willekeurige*, *onvertrouwde* code inperken. Het stelt ons bijgevolg in staat om te verifiëren of de concrete code van een applicatie aan bepaalde interessante eigenschappen voldoet, zelfs in de aanwezigheid van willekeurige onvertrouwde code.

In de eerste hoofdstukken van dit proefschrift illustreren we deze benadering in verscheidene situaties. We introduceren formele, gemechaniseerde modellen van verschillende capability machines die gebruik maken van universele contracten voor één taal om te redeneren over concrete assembleertaalcode.

Het eerste model, *Cerise*, illustreert hoe men kan redeneren over de spatiale bescherming en compartimentering die een doorsnee capability machine biedt.

Om het model te illustreren verifiëren we een teller die afhankelijk is van de encapsulatie van zijn private state, en een calling conventie op de heap die gebruik maakt van dynamische geheugenallocatie. We verifiëren ook een voorbeeld van dynamisch sealen, wat illustreert dat hardware capabilities voldoende krachtig zijn om ontwerppatronen uit de literatuur over object capabilities op assembleertaalniveau te implementeren.

Vervolgens wordt ondersteuning voor effecten in de vorm van *Memory-Mapped I/O* (*MMIO*) toegevoegd aan het Cerise-model. De compartimentering die capabilities bieden wordt gebruikt om efficiënte, nestbare beveiligingswrappers rond I/O-apparaten te construeren. De wrappers maken het mogelijk om verschillende lagen van safety-eigenschappen voor het volledige systeem af te dwingen over de MMIO-traces van de capability machine. Interessant genoeg kunnen verschillende, gelaagde wrappers worden geverifieerd onder verschillende modellen van aanvallers, en de resultaten die worden verkregen door het verifiëren van lagen die zich dichter bij de hardware bevinden, kunnen worden hergebruikt bij het verifiëren van hogergelegen lagen. Deze nesting en de daaruit voortvloeiende mogelijkheden voor verificatie zijn moeilijk efficiënt te realiseren op hedendaagse standaardhardware. Om het model te illustreren verifiëren we twee voorbeelden in een systeem van wrappers bestaande uit drie lagen, waarbij elke laag de resultaten van de vorige hergebruikt en een verfijnder model beschouwt voor de aanvaller.

De laatste instantiatie van universele contracten waar we ons op richten, is het efficiënt ondersteunen van een vorm van *enclaved executie* bovenop capability hardware; een combinatie die nog niet eerder gerealiseerd was. Enclaved executie is een populair mechanisme voor het dynamisch creëren van Trusted Execution Environments (TEEs), ook wel enclaves genoemd. Enclaves zijn geïsoleerde uitvoeringscontexten die de integriteit en vertrouwelijkheid van software binnen de enclave beschermen (zelfs tegen gecompromitteerde systeemsoftware) en die ondersteuning bieden voor attestatie. We demonstreren een nieuw, bottom-up ontwerp voor flexibele enclaves bovenop een capability machine en presenteren drie verschillende implementaties van dit ontwerp, vergezeld van eenvoudige prestatiebenchmarks. Het bepalen van de juiste formele behandeling van enclaved executie als een universeel contract is lopend werk, en wordt besproken in het conclusiehoofdstuk.

Redeneren over code op assembleertaalniveau is moeilijk en leidt dikwijls tot fouten. In het laatste inhoudelijke hoofdstuk verleggen we daarom onze focus van redeneren op assembleertaalniveau (i.e., binnenin één taal) naar redeneren over veilige compilatie naar capability architecturen. Het standaardcriterium van *full abstraction* wordt gebruikt om veilige compilatie te instantiëren, en een compiler van C-achtige code die in separation logic geverifieerd is, naar C-achtige code met ondersteuning voor zogenaamde lineaire capabilities (een niet-dupliceerbaar type capabilities), wordt fully abstract bewezen. De intuïtie achter de compiler is dat de lineaire separation logic resources kunnen worden gereïficeerd en gerepresenteerd door lineaire capabilities in de targettaal. Het full abstractionbewijs impliceert ruwweg dat aanvallen die mogelijk zijn op targettaalniveau in de capability architectuur al mogelijk moeten zijn geweest op het geverifieerde brontaalniveau, i.e. de compiler creëerde geen extra veiligheidsproblemen die niet ook al aanwezig waren op het brontaalniveau. Op de lange termijn is de hoop om een vorm van graduele verificatie te ondersteunen, waarbij delen van codebases in C die kritiek zijn voor de veiligheid kunnen worden geverifieerd en gecompileerd met deze compiler, zodanig dat veiligheidsgaranties kunnen worden afgedwongen in aanwezigheid van buggy of onvertrouwde code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With each passing year, our modern society relies more heavily upon computing devices for its proper functioning. As machines permeate all aspects of our daily lives, their size decreases and their applicability broadens in scope, moving beyond the classical mainframe and desktop models, and into novel modes of use, such as cloud computing, embedded devices and the internet of things. Along with this increased presence, the number of scenarios in which multiple distrusting parties need to have their software interact securely has rapidly increased as well. This interaction either takes place over a network, or within the same machine, and needs to be secured efficiently. Even within a single trust domain, there is a need for *fault isolation*: if the software or hardware running on a certain node contains mistakes (commonly referred to as *bugs*), errors can propagate to different parts of the system. These errors should be isolated, i.e. their scope should be limited to the component in which the error occurred, no matter whether it was caused by regular execution or triggered by an attacker.

Consequently, a *security primitive* to *protect* the interfaces between different components is required. *Compartmentalization* is an important aspect of protection; by identifying different compartments and isolating them at runtime, both within and between trust domains, the attack surface can be decreased, and the impact of bugs limited. Additionally, measures to enforce security properties of interest at component boundaries are often also required. For example, photo editing software might want to revoke access to an image buffer after an untrusted plug-in has applied a transformation to it, or an assembly function might want to enforce that a function it calls cannot return to a different function. In this dissertation, we interpret protection very broadly: any security primitive that aids in compartmentalizing different components and/or can be used to enforce properties of interest at component boundaries is considered a protection primitive. Many existing primitives fit under this wide umbrella: software-based fault isolation (SFI), hardware security features like enclaves, process-based isolation, type systems and verification, etc. Each primitive has its own advantages and disadvantages.

Ideally, a protection primitive satisfies the following properties:

- **Flexibility**: The primitive should provide *economy of mechanism* [135], i.e. it should be applicable in many scenarios, all the while being implemented by simple mechanisms. The ability to provide fine-grained protection increases flexibility. Typically, low-level primitives enable greater flexibility by allowing the definition of bespoke, higher-level primitives in terms of themselves. On the other hand, higher-level primitives are more portable across architectures, contributing to our next point.
- **Backwards Compatibility**: Better compatibility with existing *Instruction Set Architecture*s (*ISA*s), binary-level codebases, source languages and compilers makes companies more likely to transition to the new protection primitive. The abstraction layer that the protection primitive is defined at influences how its backwards compatibility can be ensured.
- **Cost**: To allow for general adoption, the cost related to the use of the primitive should not be prohibitive, both in terms of runtime efficiency as well as in terms of other resources (hardware cost, energy consumption, additional developer effort, latency, etc.).
- **Limited Trust**: The primitive should not unduly increase the attack surface, usually specified in terms of the *trusted computing base* (*TCB*), i.e. the collection of hardware and software of which correct operation is required for security guarantees to hold.

In this thesis, we will focus on a specific primitive, namely *hardware capabilities*, a type of bounds-checked pointer that carries permissions. Capabilities are supported on a special type of ISA called a *capability machine*. Due to the bounds checking and the fact that capabilities are *unforgeable*, capability machines can guarantee a type of *spatial memory safety*. By adding a form of controlled invocation between components, an efficient and flexible form of fine-grained compartmentalization can be built on top of this memory safety. Within the capability machine design space, we draw most inspiration from the recent CHERI capability machine [176]. As we will discuss, CHERI satisfies the majority of the desired properties above.

Within the context of hardware capabilities and CHERI, the two main research directions explored in this dissertation are as follows:

1. Given their flexibility and low-level nature, writing *correct and secure* code that makes use of capabilities is difficult. This difficulty creates opportunities for

formal reasoning techniques. How can we formally reason about programs manipulating capabilities, even in the presence of arbitrary, untrusted attackers? How does introducing new features to the capability machine change the formal reasoning?

2. Both writing code at the assembly-level and reasoning about it is hard and error-prone. Can we instead build a *secure compiler* that targets a capability machine architecture, allowing us to program and reason at a higher level of abstraction?

In the remainder of this chapter, Section 1.1 first provides more background information on capability machines, particularly their philosophy and appeal. Next, Section 1.2 delves into the first research direction and discusses how one can reason about code protection in a generic way, making use of so-called *universal contracts*, and what instantiations of universal contracts we will consider. Section 1.3 then discusses the second research direction. It specifies what we mean by *secure compilation* in this thesis, and lifts the notion of universal contracts to a cross-language setting. Finally, Section 1.4 summarizes by providing an outline of the following chapters.

1.1 Capability machines

Capability machines are an instantiation at the ISA level of the capability-based security philosophy [105], which spans both lower and higher levels of abstraction. Section 1.1.1 first discusses this general notion of capability-based security, after which Section 1.1.2 discusses capability machines, and Section 1.1.3 discusses CHERI specifically.

1.1.1 Capability-based security

Capability-based security is a philosophy for designing secure systems and programming languages that relies on a primitive called *capability*. Generally speaking, capabilities are a type of unforgeable token or key that inherently carries the authority to access resources (e.g. segments of memory, files, I/O devices, sockets, library APIs, ...). *Unforgeable* here means that capabilities cannot be created out of thin air, but must rather be derived from more potent capabilities, a property known as capability *monotonicity*. Thus, each component's authority decreases monotonically (i.e., can never increase) in the absence of communication: passing capabilities over communication channels between different components is the only way of sharing authority.

Capabilities hence explicitly represent a subject-centric notion of authority, as opposed to more classical object-centric notions where authority over a resource is either governed by the resource itself or by a trusted centralized party (as with e.g. access control lists or file permissions). Without a proper capability to do so, no resources can be accessed. In other words, capabilities avoid *ambient authority*, a type of omnipresent authority held by many components, that can be exercised without requiring an explicit capability to do so. Global variables are an example of ambient authority.

The appeal of explicitly representing and delegating authority in this way is that applications become easier to compartmentalize and audit for security, and flexible, powerful design patterns emerge. We discuss these advantages in some more detail below. Examples of capability systems are Capsicum [172], EROS [139] and CHERI [176]. Examples of capability-based languages are E [105], Pony [35] and Wyvern [112].

One of the main motivators for capability-based security is the *Principle Of Least Authority* (*POLA*), also called *Principle Of Least Privilege* [135]: it stipulates that each component should only possess direct authority over those resources it requires for its proper and legitimate functioning. For example, a user application that solely requires access to directories under a certain path should not be provided with authority to access the entire file system. Implementing POLA becomes far simpler in a capability setting, because capabilities explicitly represent authority, thereby avoiding ambient authority, and because they can carefully be handed out on a need-to-know basis. Adhering to POLA decreases the coupling between different components and increases compartmentalization, making components easier to audit and increasing their security. As a side benefit, this principle reduces the impact of so-called *confused deputy attacks*; attacks where a bug in a component causes it to (accidentally or maliciously) misuse its authority. A bug in the aforementioned POLA-compliant user application can only lead to abuse under the path it has received authority for, as opposed to the entire file system.

The *Principle of Intentional Use* is a second principle that is satisfied by capabilities [107]. It further reduces the risk of confused deputy attacks. This principle dictates that any component must express its intent to exercise authority explicitly: for example, a call into a file system library should require a capability to access the appropriate part of the file system, and at the assembly level, ISA instructions must explicitly provide capabilities for memory as arguments. This principle makes confused deputy attacks harder to execute, since an attacker needs to trick its victim into misusing a *specific* capability, rather than tricking the victim into exercising its general authority.

Lastly, in order to prevent different components from accessing each other's capabilities a mechanism to limit the scope of capability authority is required. In other words, a controlled form of non-monotonicity needs to be supported, such that the accessible set of capabilities can be swapped during a context switch. Additionally, a form of controlled invocation is desired, such that not all components can freely invoke one another, as this constitutes a form of ambient authority. Thirdly, flexibility of the system is improved if the right to invoke other components is made transferable. The capability-style solution to all of these concerns is to provide support for *object* capabilities: a special type of unforgeable, transferable reference that represents the reified authority to invoke (part of) the interface of another object or component. Closures and software objects are two ways of implementing object capabilities. Note that treating objects like capabilities brings other benefits related to compositional, object oriented design [105], which we will not be focusing on in this introduction.

Faithfully adhering to the capability philosophy makes a system or language *capability safe*, and renders it impossible for a component to access any resource without a proper capability to do so. At the systems level, commodity operating systems often do not achieve capability safety, because they do not break up authority to access e.g. the file system, and do not handle authority as a first class citizen, allowing processes to share it in restricted ways.

At the language level, note that many (statically or dynamically) type-safe languages (e.g. Java or Rust) are memory safe, but not capability-safe: they enforce memory safety (through a type system, garbage-collection, dynamic bounds checking, etc.) but they *do* allow arbitrary external libraries to be imported and invoked, without requiring object capabilities to do so. This increases the trusted computing base of any code and makes auditing for security more difficult, since the code needs to be inspected to find out its concrete dependencies. Thus, memory safety can be seen as a weaker, memory-only approach to capability safety. Even here, many backdoors in existing safe languages undermine the provided safety guarantees. For example, Rust's `unsafe` construct, *Foreign Function Interfaces* (*FFI*s), Java's reflection, and Haskell's `unsafePerformIO` all provide different forms of ambient authority and potentially undermine their respective type system's guarantees.

1.1.2 Capability machines

Capability machines are a specific type of ISA that implements support for *hardware capabilities*[1], the primitive of interest in this dissertation. Capabilities are a low-level, flexible primitive that represents hardware-enforced, bounds-checked authority to access system resources. The main appeal of capabilities is that they allow for fine-grained memory protection and compartmentalization, as we will discuss below. They additionally satisfy the principle of intentional use, because ISA instructions that

[1] Further mentions of "capabilities" refer to this specific variant.

manipulate system resources require the proper authority of capabilities in their argument registers.

In the 1960s, Dennis and Van Horn [42] were the first to define a notion of capability. The book of Levy [94] provides a more general historical introduction[2]. The field has a rich history with many research and industry projects, but never saw general adoption. Over the last decade however, Cambridge's *Capability Hardware Enhanced RISC Instructions* (*CHERI*) project [175] has renewed interest into the technology, and paved a path towards practical usage. One reason for this turnaround was mentioned in the introduction: the growing importance of security alongside performance, necessitated by an increasing number of software interactions. Recently, Arm has taken a keen interest in the CHERI project, and announced the Morello prototype, an experimental capability board that augments an Armv8-processor with CHERI-like extensions [12, 11]. In a recent research paper, Intel demonstrated a probabilistic take on capabilities [91].

Capabilities can be used to protect various types of resources:

- *Memory capabilities* are the most primitive type of capability, authorizing certain memory operations (e.g. reading R, writing W, executing X, or combinations thereof) within a hardware-enforced, bounds-checked memory region. Figure 1.1 illustrates the fields typically present in a memory capability: the capability permits exercising permissions *perm* on a memory segment from address *base* to *end*, and currently points to address *addr*.
- *Enter capabilities* (also called *sentry capabilities* [175]) represent the right to invoke a procedure, potentially in another trust domain [28]. They cannot be executed directly and disallow regular reads or writes to their address range. They can only be jumped to, causing the enter capability to transition into an executable memory capability. The resulting capability is loaded into the *program counter* (pc) register, effectively causing a context switch to occur. In other words, enter capabilities are an instantiation of object capabilities comparable to closures in high-level languages.
- *I/O capabilities* protect access to specific devices or files. In case the ISA supports *Memory-Mapped I/O* (*MMIO*), I/O capabilities can be represented by regular memory capabilities.

Some architectures support variants of the above capabilities (e.g. non-duplicable memory capabilities called *linear* capabilities [144]), or entirely different capabilities (e.g. *sealing* capabilities for the symbolic encryption of other capabilities [175]), which we will introduce throughout the text as required.

[2] Some of the capability machines described in this book, e.g. the one by Dennis and Van Horn, are not implemented in the ISA, but rather, consist of a supervisor- or hypervisor-level software implementation. Nevertheless, there are large similarities between both styles of implementation at this level of abstraction.

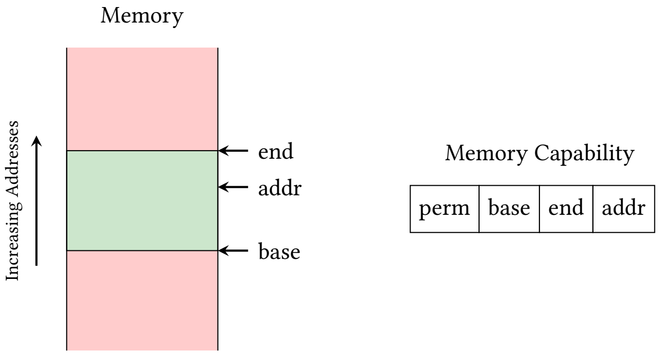


Figure 1.1: Typical fields and authority of a memory capability.

One of the main selling points of capability machines is that they provide a deterministic form of *spatial memory safety*: memory capabilities track bounds and permissions on (subranges of) allocated pieces of memory, and they prohibit accesses to memory that fall outside of these bounds. This prevents e.g. buffer overreads and -writes by construction. On the other hand, *temporal memory safety* disallows dereferencing pointers to freed memory (so-called *dangling pointers*) or uninitialized memory and is harder to guarantee on a capability machine. However, enforcing it is equally critical, since temporal violations allow attackers to copy or manipulate the victim's capabilities and hence its authority. One line of research has investigated low-overhead ways of guaranteeing temporal heap safety in CHERI through parallelizable revocation sweeps [182, 55], whereas other capability machines have tried to incorporate temporal safety by design, by including identities that are bound to an allocation's lifetime into the capability metadata [106, 134].

Guaranteeing memory safety (both spatial and temporal) in low-level languages is an important issue for which tools and solutions exist but are insufficiently employed: Microsoft reports that over the last decade, ~70% of annually filed *CVEs* (*Common Vulnerabulity Exposures*, a type of bug report) across their products were caused by memory safety issues [153], and Apple reports consistent numbers around 60% for consecutive versions of macOS and iOS [82]. Microsoft estimates that in 2019, at least 31% of the reported vulnerabilities in their products would be prevented by applying the type of spatial memory safety found in CHERI, and at least 67% by including temporal safety [75].

On top of this (spatial) memory safety, enter capabilities can provide support for lightweight compartmentalization. When jumping to an enter capability, the caller causes an intra-address-space domain transition to the trust domain of the callee. This domain transition can be implemented without requiring intervention of a centralized authority [see e.g. 94, chapter 4], and hence incurs less overhead than a process-level context switch on a commodity architecture. Hence, enter capabilities allow for a move away from virtual memory as a coarse-grained security primitive, replacing it with fine-grained components in scenarios with complex interactions between different stakeholders. An example application is kernel protection, where enter capabilities can help reimplement a monolithic single-protection-domain kernel in a compartmentalized way.

The security properties we discussed are predicated on the unforgeability of capabilities: an attacker that can create capabilities out of thin air (i.e. in a non-monotonic way) can undermine capability safety. At the same time, unforgeability provides inherent security benefits as well, because it allows distinguishing between code and data, thereby making e.g. control-flow hijacking more difficult. Different capability machines ensure unforgeability in different ways: either a hardware *tag bit* denotes whether a memory location corresponds to a capability (as is the case in CHERI), or the capability's metadata fields are stored separately in protected shadow memory and indexed by either the pointer's address (e.g. in Hardbound [43] or Intel MPX [117], although the latter is arguably not a capability system) or a pointer identity stored with the pointer itself (as is the case in HeapCheck [134]). The different solutions trade off between the desired protection properties we mentioned in the introduction: backwards compatibility with existing binaries and ISAs, support for temporal memory safety, runtime overhead, spatial locality, virtual memory size and fine-grainedness of protection; Saileshwar et al. [134] provide a high-level discussion.

1.1.3 CHERI

Throughout the thesis, we will use formal models of various capability machines, all of which draw inspiration from the CHERI capability machine. For this reason, and because CHERI is spearheading the new wave of capability machines, this subsection briefly focuses on CHERI specifically. The discussion here will be restricted to the high-level concepts and design philosophy underlying CHERI [174], as Chapter 2 in particular provides a gentle introduction to the more technical aspects of the semantics and models we use, and how to program in them.

The aim of CHERI is to extend existing ISAs with support for capabilities, obtaining the benefits of fine-grained memory protection and compartmentalization we mentioned in Section 1.1.2. However, CHERI additionally considers *performance* and *backwards compatibility* as important design constraints that must be met if a capability machine is to see large-scale practical adoption. This gives rise to a careful design process, where different trade-offs must be made.

Initially, CHERI was a specification for 256-bit architectural capabilities on the 64-bit MIPS architecture. In the meantime, the same protection mechanisms have also been applied to other RISC architectures: to 32/64-bit RISC-V and to an experimental version of 64-bit ARMv8-A (as part of the Morello project mentioned above [11]), demonstrating portability of the protection model [175]. As part of this expansion to different architectures, a compressed capability format in floating point style called *CHERI Concentrate* was developed to decrease memory footprint and hence increase practical applicability [179]. This resulted in compressed 128-bit capabilities on 64-bit architectures (CC128) and 64-bit capabilities on 32-bit architectures (CC64).

For each variant of CHERI, a formal specification in the *Sail* ISA specification language [13] has been written [14, 132, 18]. The Sail models can be used to generate a C or OCaml emulator for the processor, to validate processor implementations against, and they can be exported to definitions in various proof assistants. They have proven useful in the past to verify various security properties of interest over the full-scale CHERI variants [110, 18]. Section 2.10.3 discusses the relationship to our models and proofs in more detail.

CHERI is a tag-based capability machine; metadata is stored in-band (i.e., adjacent to the raw pointer data), and capabilities are protected using a single, unforgeable tag bit, that denotes whether a specific capability-aligned memory region contains a capability or not. Since currently no DRAM memory with built-in tag bits exists, CHERI stores tags in a separate shadow region of main memory. However, since caches can more easily be widened, tag bits *are* threaded through the cache hierarchy, avoiding split data and tag caches. By compressing the shadow tag region and organizing it in a hierarchical way, while also introducing a bottom-level cache for tags that further exploits spatial and temporal tag locality, DRAM traffic overhead caused by tags was reduced to $< 5\%$ on pointer-intensive workloads, and $< 1\%$ on most regular workloads [74].

All CHERI capabilities are of the memory capability format presented in Figure 1.1, with an additional *otype* (*object type*) field that is used for sealing, which will be explained in Chapter 4. In other words, no special type of capabilities for I/O is provided, as these are subsumed by memory capabilities through the use of MMIO. Enter capabilities are implemented by having memory capabilities take on a special sentry permission E, similar to the historic M-Machine [28].

We now discuss the two aforementioned design principles for CHERI, backwards compatibility and performance, in more detail.

First, to support backwards compatibility, CHERI follows an *incremental deployment* philosophy: existing compiled codebases should be able to run unaltered on top of CHERI hardware and benefit from coarse-grained protection, and critical parts can be gradually recompiled to benefit from more fine-grained memory protection. One of the main motivators for this philosophy is the existence of large, compiled C/C++ codebases made up of systems code (for e.g. kernels or language runtimes). For this approach to work, good tooling support is required: the compiler should for example be able to use source-level information to transform references in the source language into capabilities at the target level, while requiring only minimal rewrites for edge cases at the source level. To this end, the Clang/LLVM-compiler [64] has been extended to provide support for the architectures that CHERI runs on.

To have the choice of fine- versus coarse-grained protection in the back-end, two different sets of load and store instructions are present: one where an offset is loaded into a general purpose register and a coarse-grained data capability present in the *DDC* (*Default Data Capability*) register grants the necessary authority, and an alternative fine-grained set of instructions where an authorizing capability is directly provided in a general purpose register. Supporting both sets of load and store instructions does not waste opcode space in CHERI-RISCV (where opcode space is more limited than in CHERI-MIPS), since a *mode bit* flag on the pc capability is used to select between both sets of encodings for the different load and store instructions.

When running compiled legacy code, the DDC approach can be used to compartmentalize the unaltered component as a whole, whereas during recompilation *pure capability code* can be generated to replace all pointers by proper capabilities, so that the more fine-grained set of instructions can be used. A *hybrid code* approach also exists, where static information or programmer annotations are used by the compiler to only protect *some* specific pointers. This is mostly useful in code that bridges between pure capability and legacy code, for example when converting between the *ABI*s (Application Binary Interfaces) of pure capability and legacy code. Orthogonal to these considerations, explicit compartmentalization using enter capabilities can also be applied by the compiler.

In order to further improve backwards compatibility, CHERI is designed as a *hybrid* capability machine, in the sense that it supports virtual memory alongside architectural capabilities. This allows conventional software that relies upon this feature to run unaltered on CHERI. If virtual memory is present and enabled, capability authority is interpreted as a virtual address range within the active page table mappings. Currently, page tables are implemented in a non-capability-aware way: any software with sufficient permissions to change page mappings can install any mapping from virtual to physical memory, without having to demonstrate authority to access said memory. However, a proposal to refine this and replace physical addresses with physical capabilities in page table entries exists [175, section D.14.4].

To illustrate backwards compatibility, the FreeBSD operating system [145] was first run on CHERI unmodified [180]. It was then gradually adapted to provide pure capability support to user processes through an ABI adapter called CheriABI, requiring only minor changes to the operating system itself [39].

Concerning the second design principle of performance, CHERI manages to keep overhead relatively low. On CHERI-MIPS (with 256-bit capabilities and without optimized tag storage and caching), capability-protected code incurred an overhead of $\leq 20\%$ on pointer-intensive benchmarks compared to unaltered MIPS code running on the same processor [180]. In the meantime, the transition to 128-bit capabilities has reduced memory footprint and cache pressure. Optimized tag handling has reduced DRAM overhead, as previously described. Comparing unoptimized CHERI-MIPS to BERI (the processor CHERI-MIPS is an extension of) showed a clock frequency reduction of 8.1% and an increase in logical elements of 32% [180].

Although these numbers have likely been improved in consecutive versions and implementations of CHERI, their utility is limited for 2 reasons. First, they were obtained from an FPGA model of CHERI, not a physical chip implementation. Second, these numbers reflect the micro-costs associated with CHERI, but neglect the *macro-benefits*. The protection and compartmentalization that CHERI provides allows applications to be implemented differently, reducing e.g. the required number of costly domain switches or run-time checks.

The ARM Morello project will close this knowledge gap. First, it provides the first physical processor running CHERI protection. Second, it promises to deliver the software models required to test the true, practical overhead incurred by real-world applications running on CHERI. Large companies like Google and Microsoft have committed themselves to writing software for the Morello platform [12]. This will help overcome one of the biggest hurdles in hardware design: the cyclic dependency between novel hardware and software models developed for this hardware. Without practical hardware implementations, software developers are reluctant to invest sufficient funds and manpower into corresponding software models. Without software models to develop hardware for, it is difficult to demonstrate the utility of the hardware implementation.

To summarize, CHERI's protection characteristics can be decomposed as follows in terms of the four desirable protection properties mentioned in the introduction:

- **Flexibility**: Capabilities are a very low-level, fine-grained and general-purpose primitive. Their utility has been demonstrated across architectures.
- **Backwards Compatibility**: Legacy binary code can run on CHERI unaltered. When recompiling source code, CHERI allows for incremental deployment, in order to apply its fine-grained protection gradually.
- **Cost**: We discussed the cost of extending the BERI processor to implement an unoptimized version of CHERI-MIPS above. The Morello project will provide us with more accurate estimates of CHERI's real world cost, in terms of runtime overhead, but also hardware and energy cost, and developer overhead.
- **Limited Trust**: CHERI has so far been added to RISC architectures, as a relatively small set of ISA instructions. This provides us with a small TCB. However, since capabilities are a low-level primitive, using them to write correct

and secure applications is difficult. Good tooling support (compiler, higher-level languages, …) is hence required, and CHERI programs are good candidates for verification, as we will discuss in this thesis.

1.2 Formal reasoning about capability code

How can we be sure that the security claims made by the designers of CHERI and other capability machines hold up? In large part, this question is answered by proving that the machine is indeed capability safe, i.e. that there is no way to escalate capability authority. We did not have the reasoning tools and techniques to do this when capabilities were invented in the 60s, but in the meantime, formal methods have been developed that allow us to prove capability safety.

For practical purposes however, reasoning about safety of a capability machine does not suffice: the end goal is to reason about the behavior of *concrete* code of interest, and how it can satisfy properties of interest in the presence of *untrusted* code. The reason we can sensibly reason about concrete code is that capability safety provides an upper limit that restricts the behavior of the untrusted code. It guarantees that no matter what untrusted code the concrete application code runs against, this code can never escalate its privileges to trivially compromise security. In this sense, capability safety can be seen as a *universal contract* [162] that restricts the authority of arbitrary adversarial code, enabling modular verification of concrete code.

Abstracting away from our capability setting, a universal contract is a formal expression of the restrictions that *some* protection primitive places on arbitrary, untrusted code. It is *universal* in the sense that it holds for *any* code that the protection primitive is applied to. The *contract* terminology indicates that a certain upper bound on the behavior of untrusted code is captured and enforced. Given our broad interpretation of protection, the applicability of universal contracts is equally broad. A few examples are as follows:

- Polymorphic functions in System F [63] cannot inspect type variables, satisfying a form of parametricity [169]. Wadler's *free theorem* terminology has a similar meaning to our universal contracts.
- Haskell code that does not make use of the I/O monad or unsafe functions is guaranteed to be side-effect-free.
- On a capability system, if a component has no access to a capability to conduct I/O with, invoking this component will not cause side effects.

We provide further discussion of universal contracts throughout the following chapters, particularly in Section 2.10.4.

Section 1.2.1 now describes a general template methodology for reasoning about concrete code in the presence of untrusted code, which we will follow. This template demonstrates the role universal contracts play in the bigger picture. Section 1.2.2 discusses the meaning of universal contracts in our capability machine setting, and outlines the different protection properties we will formalize through universal contracts within this dissertation.

1.2.1 Template for formal reasoning about mixed-trust systems

Figure 1.2 illustrates a common series of steps involved when formally verifying properties of interest for concrete code that interacts with arbitrary, untrusted code. This schema transcends our specific capability setting: it applies to any low- or high-level setting in which known code that can be verified with a program logic interacts with untrusted code that has its authority restricted by some protection primitive [see e.g. 152, 78, 142, 136].

The reasoning occurs in three different steps. First, any initial conditions of the system (e.g. the state of memory and registers, the state of hardware) are translated from the physical domain to conditions in the logical domain (①). Then, a program logic along with universal contracts is used to reason about the execution of the program in the logical domain (②). The result will be some statement in the logic (e.g. in the form of a Hoare triple), saying that the code satisfies a property of interest. To transport this statement back from the logical to the physical domain, a *soundness* (sometimes also called *adequacy*) property is required (③), which states that claims that the logic makes about physical execution, actually hold true for the execution in the physical domain. The rest of this section describes step ② in more detail.

To perform reasoning in the logical domain, a program logic is built to reason about concrete code, and a semantic model (typically a *logical relations* model, see e.g. [128]) is built to capture the guarantees provided by the universal contract. Once the notion of universal contract has been defined, a well-known type of theorem called the *Fundamental Theorem of Logical Relations* (*FTLR*, cfr. Figure 1.2) is used to prove that the universal contract is indeed satisfied for arbitrary code. To be precise, in order for the fundamental theorem to apply, the untrusted code is still required to satisfy some syntactic well-formedness property (e.g. well-typedness in a certain syntactic type system, not having access to certain system resources or capabilities, requiring some hardware TCB to be set up correctly, …). The fundamental theorem hence proves that code satisfying this *syntactic* property also satisfies the *semantic* logical relation.

It is convenient for the semantic model to reuse judgments from the program logic (e.g. Hoare triples or weakest preconditions) in its definition. This enables incorporation

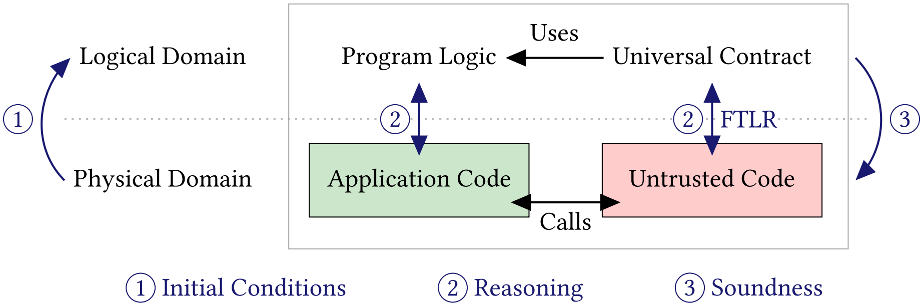


Figure 1.2: High-level set-up involved when reasoning about trusted Application Code that interacts with Untrusted Code.

of proofs or invariants for concrete code into universal reasoning when concrete code calls untrusted code, and allows doing the opposite when untrusted calls concrete code. This also allows reuse of e.g. Hoare triples proven in the program logic in the proof of the FTLR. Lastly, because judgments of the program logic are reused in universal contracts, the soundness statement of the program logic can be used to transport proofs involving both concrete and untrusted code back to the physical domain in step (3).

1.2.2 Outline: the different protection mechanisms and universal contracts we consider

In our setting, the protection primitive that gives rise to a universal contract will always be some variant of capabilities. The spatial protection and compartmentalization that capabilities provide, naturally enables intra-address-space protection in the presence of untrusted code. In this sense, the universal contract can be seen as an expression of the capability safety of the modeled capability ISA. The capability machine models employed in this thesis are simple, to allow prototyping novel concepts required for formal reasoning. As we will discuss in Chapter 6, we hope to introduce more automation and extend the complexity of our models to realistic capability machines in future work.

We now discuss the different protection mechanisms and corresponding instantiations of the universal contract philosophy that arise throughout the thesis.

First, Chapter 2 provides a gentle introduction to the thesis, describing *Cerise*: the combination of an operational semantics, a program logic and universal contracts (with FTLR) for a basic capability machine. The semantics only supports two types of capabilities: regular memory capabilities, and enter capabilities that are encoded as memory capabilities with an E permission. The universal contract is formalized as a logical relation that captures the spatial protection provided by memory capabilities, and the compartmentalization provided by enter capabilities. As an example of the type of reasoning possible in Cerise, consider an application that contains a trusted and an untrusted compartment. The untrusted code solely has object capabilities to invoke the trusted code, and will notably not receive any (direct or indirect) memory capabilities to the local state of the trusted compartment. This syntactic condition ensures that the FTLR applies. Compartmentalization then allows the trusted compartment to uphold arbitrary invariants over its encapsulated local state, while interacting with the untrusted code.

Cerise is in fact a simplified version of a more complicated model we developed in prior research [62], which included two additional types of capabilities, namely *local* and *uninitialized* capabilities. Local capabilities were already present in CHERI, but the concept of uninitialized capabilities was original to this work. This work has not been included in this thesis. I discuss the publication details and my contribution in more detail in Section 1.4. The goal of the work was to guarantee a type of *stack safety* (more precisely, *local state encapsulation* of stack frames and *well-bracketed control flow*) within a single address space on a capability machine, without requiring a trusted stack manager to handle calls between different trust domains. To this end, the universal contract described how local capabilities allowed for a form of revocable memory access on top of spatial memory safety, and how uninitialized capabilities additionally enforced a write-before-read memory discipline.

Next, Chapter 3 studies the addition of effects, in the form of *Memory Mapped I/O* (*MMIO*), to the simple Cerise model. The universal contract remains largely unaltered, except for the fact that untrusted code cannot get direct access to capabilities that provide access to MMIO (or at least some privileged MMIO regions). We use this model to reason about encapsulation of security wrappers around MMIO devices, and how they can guarantee safety properties over interaction traces with the environment of the capability machine. Using capabilities, it is possible to nest these security wrappers in a lightweight way without the need for address-space-based isolation.

Chapter 4, in turn, builds a lightweight notion of *enclaves* (also known as *Trusted Execution Environments* or *TEEs*) on top of our capability machine. Enclaves are isolated execution contexts that protect integrity and confidentiality of software inside the enclave (even against compromised systems software) and that support attestation. Our enclave design requires *sealed* capabilities to be added to the set of capabilities present in Cerise. These capabilities can be used to implement a form of symbolic encryption, enabling secure communication. Additionally, a hash is required to compute enclave identities; digests of memory regions. This chapter does not yet discuss how we formally model enclaves in Cerise, or how we capture the meaning of trusted execution in a universal contract. This is currently ongoing research, which the future work in Section 6.1.1 provides some more insight into.

Finally, Chapter 5 takes a different, cross-language approach to reasoning, and studies capability machines as a target for verifiably secure compilation. Concretely, the compiler translates separation-logic-verified C-like code into lower-level C-like code with support for so-called *linear* capabilities; a type of capabilities that cannot be duplicated. The intuition behind the compiler is that the affine separation logic resources for memory can be reified during compilation, and represented by semantically similar linear capabilities at the target level. The source-level separation logic contracts are enforced as a property at the target level by a combination of spatial memory safety from regular capabilities, revocation allowed by linear capabilities and contract checking stubs at the boundaries between trusted and untrusted code. Since this chapter reasons *vertically* (cross-language) about the security of the compiler, as opposed to *horizontally* (intra-language) in the previous chapters, the form of the universal contract differs from what we discussed in Section 1.2.1. In the next section, we discuss the general reasoning involved in secure compilation, and how this secure compiler establishes a cross-language universal contract.

1.3 Secure compilation to capability architectures

In this section, we first describe a general notion of secure compilation and outline the high-level reasoning involved in the secure compilation proofs of Chapter 5. We also relate the universal contracts from last section to this notion of secure compilation.

Secure compilation is concerned with formally proving that a given compiler preserves security properties of interest from the source to the target language. At a more fundamental level, it also seeks the *right* security properties to formalize compiler security with, evaluates the usefulness of different properties and investigates how difficult the ensuing security proofs are. There is a consensus among researchers that no single *best* approach to secure compilation exists, but rather, that different secure compilation criteria are best applied in different circumstances.

Initially, Abadi proposed *full abstraction* [1], a now standard way of instantiating secure compilation [see e.g., 2, 57, 121, 108, 45, 144]. Patrignani et al. [122] provide a good survey of the research on the topic. Full abstraction is the notion of secure compilation we employ in Chapter 5, and will be defined in more detail there. Intuitively, it states that any *attack* that is possible on a compiled target program, must have already been possible on the original source program. In other words, the compiler is not allowed to introduce new attack vectors, and can hence not increase the *power* of the attacker. More concretely, a successful attack is formalized as the non-satisfaction of *contextual equivalence*, a binary notion of observational equivalence where two programs are considered equivalent if and only if no adversarial context exists that can distinguish the two. A compiler is fully abstract if it preserves (source-to-target) and reflects (target-to-source) contextual equivalence. The former captures the above intuition, whereas the latter is a form of compiler correctness.

Although full abstraction has seen a lot of use, it comes with certain trade-offs and is not unconditionally the best criterion to apply in any given scenario. This has to do with the fact that the exact property that is preserved from the source to the target language (and reflected from the target to the source) is only implicitly present in the definition: it is embedded in the above notion of contextual equivalence, and determined by what the context can and cannot observe. This has the advantage of not requiring explicit formalization of the property one intends to preserve and avoiding the definition of additional relations between e.g. values or effects in source and target languages, but it has disadvantages as well.

First, it potentially preserves security-irrelevant equivalences, if the observational model in the target language is overly strong. A classical example is fully-abstract compilation to an assembly language where a context can observe the size of a program, which might not be security-relevant, but now has to be hidden from the attacker. Preserving additional equivalences not only comes at the cost of a harder proof, but potentially at a runtime cost as well, if the target language enforces properties of interest using dynamic checks rather than a type system or program logic. Second, the converse situation occurs if the observational model in the source language is too weak. For example, naively proving full abstraction for a source language that supports C-style undefined behavior is difficult, since optimizing compilers often refine undefined behavior to improve performance. Third, because program equivalence is only defined horizontally, compilers that invalidate simple properties (e.g. safety properties such as “programs can never output false”) from the source to the target language can still qualify as fully abstract [125]. This last argument seems of mostly theoretical importance, since fully abstract compilers in the literature often make use of *vertical* proof techniques (e.g. cross-language logical relations or simulation relations) to prove the horizontal contextual equivalences, avoiding such pitfalls.

Given these criticisms, parallel research has investigated a whole lattice of secure compilation criteria, stated as different forms of robust preservation of hyperproperties (sets of sets of program execution traces) [4]. This research uses trace-based semantics in order to provide an intentional, explicit account of the properties that secure compilation preserves. No practically useful relationship between full abstraction and hyperproperty preservation has been demonstrated at the time of writing, although Abate et al. [4] showed that some of their criteria imply full abstraction, and Abate et al. [5] derived a criterion that subsumes both full abstraction and robust hyperproperty preservation. However, the practicality of this criterion remains to be illustrated. Other criteria have been proposed as well, including modular [123] and probabilistic [2] variants of full abstraction, trace-preserving compilation [125], and variants that allow for static and dynamic compromise of components [76, 3].

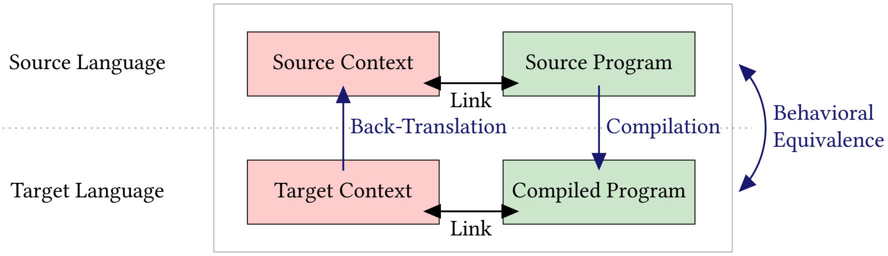


Figure 1.3: High-level overview of how security is proven for a specific compiler: a Source Program and its compilation Compiled Program and a Target Context are given. A Source Context has to be constructed such that an appropriate behavioral equivalence exists between source and target code.

What most of these approaches have in common, is that security is proven by demonstrating that an attacker context has no more power in the target language than in the source. This is achieved by means of a so-called *back-translation*: a target-to-source transformation, either of the target context directly, or of its trace semantics. The back-translation operates as a dual to compilation, and aims to represent the target-level attacker in the source semantics. Figure 1.3 illustrates this abstract view of a secure compilation proof involving a back-translation: a Source Context needs to be constructed that guarantees behavioral equivalence between source and target code. The intuition is that if such a context can be constructed, then all possible attacks that can be performed in the target language could already be performed in the source language as well, meaning that the compiler preserved the security properties of interest. The chosen secure compilation criterion and proof technique can place further constraints on the back-translation (e.g. whether it can depend on the Compiled Program) and on the notion of behavioral equivalence (e.g. its arity; how many Source Programs have to considered simultaneously).

Interestingly, the notion of behavioral equivalence in Figure 1.3 gives rise to the following cross-language notion of universal contracts: arbitrary, untrusted target-language contexts can be represented by some behaviorally equivalent source-level representation in the source language. This might not seem like a real contract at first, but it implies that the back-translated target context can be interpreted as a source-language specification of the target context. In the compiler of Chapter 5, the universal contract interpretation becomes more explicit, since the back-translation translates the protection mechanism of capabilities to code that is verified against an actual contract (in separation logic). This contract is a generally applicable correctness contract that represents the authority carried by capabilities in the original target-level code, and which all back-translated code is verified against. In other words, this contract captures the guarantees we obtain from the semantics of (linear) capabilities.

1.4 Contents of the thesis

Having provided an overview of the reasoning techniques involved in the coming chapters throughout the previous sections, Section 1.4.1 briefly enumerates the contents and motivation of each chapter. Then, Section 1.4.2 lists the work that has not been included into the thesis.

1.4.1 Included contents

Chapter 2 presents the Cerise capability machine, its program logic and universal contracts, and discusses the examples we verified in it. The Cerise model represents a least common denominator, that can be extended in various ways to prototype formal reasoning techniques for different capability features. To illustrate the model, we verify a counter that depends on encapsulation of private state and a heap-based calling convention that uses dynamic memory allocation. We also verify an example of dynamic sealing [152], illustrating that hardware capabilities are sufficiently powerful to implement design patterns from the object capability literature at the assembly level.

Next, Chapter 3 adds MMIO to the Cerise model, and discusses how this allows verification of potentially nested, lightweight security wrappers. The wrappers enable layered enforcement of full-system safety properties on MMIO traces admitted by the capability machine. Interestingly, different layers of wrappers can be verified under different attacker models, and the results obtained by verifying layers closer to the hardware can be reused when verifying higher layers. This nesting and the ensuing opportunities for verification are difficult to achieve efficiently on commodity hardware. To illustrate the model, we verify two examples in a three-layer system of wrappers, where each layer reuses results from the previous one and considers a more refined attacker model.

Chapter 4 introduces sealed capabilities to the model, and illustrates how enclaves can be built on top of capability hardware in a flexible, bottom-up way. The chapter discusses our novel capability-based enclave design, culminating in three implementations: two implementations on top of RISC-V architectures and one on top of ARM Morello. As mentioned, the formal treatment of enclaved execution is ongoing work and is discussed in Section 6.1.

Chapter 5 shifts the focus from intra-language to cross-language reasoning, and discusses a full abstraction proof for a secure compiler from verified C-like code to C code with support for linear capabilities. In the long run, the hope is to support a form of gradual verification, where security-critical parts of C codebases can be verified and compiled with this compiler in such a way that security guarantees are upheld in the presence of buggy or untrusted code.

Finally, Chapter 6 offers conclusions and general perspectives about the work. Notably, it briefly summarizes ongoing efforts to place the implementation efforts from Chapter 4 onto formal footing, sketching how logical relations can be used to reason about enclaved execution. Additionally, it discusses ways to lift the abstraction level at which we reason from assembly to a higher level of abstraction.

The papers included in Chapters 2 to 5 have generally not seen their contents altered compared to the original publications. The majority of changes that were made to the papers were esthetical, to fit the figures and text within the adjusted margins of the thesis, and to provide the different chapters with a somewhat unified presentation style (references to figure/section names, capitalization, . . .). Additionally, some minor typos and errors were corrected. A few minor additions have been made to the contents, usually to further clarify specific points compared to the original publications. These minor changes (if any) will be highlighted at the start of each chapter.

1.4.2 Omitted work

The aforementioned work on the formal treatment of stack safety on a capability machine with local and uninitialized capabilities was omitted in favor of the Cerise work. The reason is that the Cerise paper was derived from this work and provides a more general, didactic introduction to reasoning about capability machines. Concretely, the publications related to stack safety are two-fold. First, a thesis student of mine implemented the concept of uninitialized capabilities in the formal CHERI-MIPS Sail model and published the design on arXiv:

> Sander Huyghebaert, Thomas Van Strydonck, Steven Keuchel, and Dominique Devriese. Uninitialized capabilities. arXiv: `2006.01608 [cs]`

Second, we published a POPL paper describing the formal treatment of these capabilities as a tool to guarantee stack safety in a more complicated version of the Cerise model:

> Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proceedings of the ACM on Programming Languages*, 5(POPL):6:1–6:30, Jan. 2021

I was not the principal researcher of this work, but I contributed to it as part of a five-month research visit to the research group of Lars Birkedal at Aarhus university. My main contributions were in helping to develop the program logic, verify the FTLR and set up the examples.

Lastly, another thesis student of mine published a workshop paper on a variant of capabilities called *borrowed* capabilities, a novel alternative for revoking authority on a capability machine:

> Thijs Vercammen, Thomas Van Strydonck, and Dominique Devriese. Borrowed capabilities: flexibly enforcing revocation on a capability architecture. *Workshop on the Security of Software/Hardware Interfaces (SILM)*, 2021

Borrowed capabilities constitute another point in the capability revocation design space, next to linear and local capabilities. Their semantics are similar to the notions of ownership and borrowing in substructural type systems like Rust's.

Chapter 2

Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code

Publication Data

This paper is currently in submission at the Journal of the ACM (JACM). Its publication data are as follows:

> Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Cerise: program verification on a capability machine in the presence of untrusted code. *In Submission*

It is derived from the previously mentioned paper about a secure calling convention employing local and uninitialized capabilities that was accepted at POPL, but is more accessible and contains novel examples. The Cerise work was hence included in the latter paper's stead. Additionally, this paper is helpful in understanding the foundations of the formal reasoning we require in Chapters 3 and 4 of this dissertation, where we will reason about additional features on top of the ones presented here.

As with the secure calling convention research, I am not the principal author of this work. My contribution to this specific paper is limited to the proofs about the capability machine itself; I helped set up the program logic for the machine, and assisted in proving its fundamental theorem. I did not contribute to the novel examples worked out here (although I did help proofread the corresponding sections).

A prior, more limited, French version of this paper has been peer-reviewed and presented at Journées Francophones des Langages Applicatifs (JFLA):

> Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu. French. In *Journées Francophones des Langages Applicatifs 2021*. Institut de Recherche en Informatique Fondamentale, Apr. 2021

Abstract

A capability machine is a type of CPU allowing fine-grained privilege separation using *capabilities*, machine words that represent certain kinds of authority. We present a mathematical model and accompanying proof methods that can be used for formal verification of functional correctness of programs running on a capability machine, even when they invoke and are invoked by unknown (and possibly malicious) code. We use a program logic called Cerise for reasoning about known code, and an associated logical relation, for reasoning about unknown code. The logical relation formally captures the capability safety guarantees provided by the capability machine. The Cerise program logic, logical relation, and all the examples considered in the paper have been mechanized using the Iris program logic framework in the Coq proof assistant.

The methodology we present underlies recent work of the authors on formal reasoning about capability machines [62, 143, 160], but was left somewhat implicit in those publications. In this paper we present a pedagogical introduction to the methodology, in a simpler setting (no exotic capabilities), and starting from minimal examples. We work our way up to new results about a heap-based calling convention and implementations of sophisticated object-capability patterns of the kind previously studied for high-level languages with object-capabilities, demonstrating that the methodology scales to such reasoning.

2.1 Introduction

A capability machine is a type of CPU that enables fine-grained memory compartmentalization and privilege separation through the use of *capabilities*. This type of hardware architecture has been studied since the 1960's [42, 94], and in particular more

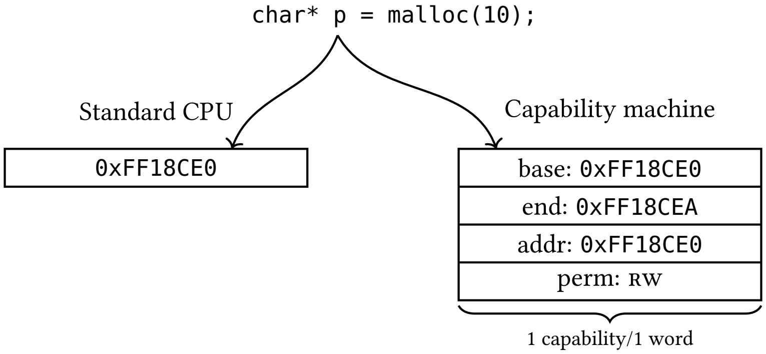


Figure 2.1: Representation of a pointer in a standard architecture vs. a capability machine. A capability is similar to a pointer with extra meta-data.

recently as part of the CHERI project [175]. Capability machines offer fine-grained and scalable privilege separation at the hardware level and they are a compelling target for secure compilation [144, 51, 162, 32].

Capability machines distinguish, at the level of hardware, between machine integers and capabilities; and a capability can be understood as a pointer with associated metadata, cfr. Fig 2.1. A machine word containing an integer value can only be used for numerical computations and cannot be used as a pointer to access memory. On the other hand, a machine word containing a capability can be used to access a given portion of memory, depending on the metadata contained in the capability. We also say that the capability *has authority over* some fragment of memory.

A capability thus corresponds to a native machine value, and can be stored in a CPU register or in memory. While this might seem wasteful due to the amount of extra metadata that needs to be carried around, for realistic capability machines a lot of work has been dedicated to the design of compressed representations for capabilities, see, e.g., [179, 28]. In this paper, we will abstract from these details and represent capabilities in their uncompressed form, as a tuple carrying the metadata.

A capability machine guarantees the integrity of capabilities: one cannot create fresh capabilities out of thin air or modify the metadata of existing capabilities in arbitrary ways. For instance, CHERI associates tags to machine words to identify whether they represent a capability or an integer. Such a tag bit is checked and set by the machine, and is not directly accessible by software. More generally, new capabilities can only be derived from existing capabilities using a restricted set of operations provided by the machine. As such, all capabilities on the system are recursively derived from the full-authority capabilities that are initially provided to software at boot time. Intuitively, the machine ensures that a given program cannot forge capabilities and obtain more authority than it held previously, a property sometimes referred to as capability monotonicity [110].

Capabilities therefore allow a piece of code to interact securely with untrusted third-party code, even within the same address space, by restricting the set of capabilities the untrusted code (transitively) has access to. In a system composed of mutually untrusted components (which might even contain malicious code), capabilities provide a way of enforcing that the overall system nevertheless satisfies some security properties.

Note, however, that capabilities are low-level, flexible, building blocks, which operate at the level of the machine code and whose metadata "just" triggers some additional runtime checks by the machine. This means that the *properties* we can actually enforce using capabilities crucially depend on how we *use* capabilities: the variety of properties that can be enforced stems from how one can use and combine capabilities.

In this paper we show how we can formally *prove* that security properties are enforced for some known verified code, *even when* that code is linked with unverified untrusted third-party code. Our model of interaction between the known and unknown code is very simple: we assume the code is in the same address space and that control is transferred from one to the other using an ordinary jump instruction. We focus on a restricted subset of the capabilities present in the CHERI architecture (using only "normal" read/write capabilities and a kind of so-called sentry capabilities, which provide a basic form of encapsulation, see Section 2.2.4). Because the security properties we consider hold even in the presence of unverified unknown code, they are sometimes referred to as *robust safety* properties [152]. The security properties we focus on are centered around memory compartmentalization, in particular, local state encapsulation. We consider a range of examples, starting with very basic examples (sharing a buffer with some unknown code), through implementations of closures with encapsulated state, and end up with a quite sophisticated low-level implementation of an interval library, for which we show that certain representation invariants are preserved, even when interacting with unknown code.

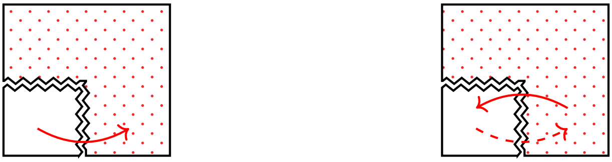
We proceed as follows:

- We first explain informally how one can program with capabilities and use capabilities to enforce memory compartmentalization (Section 2.2).
- We then introduce the formal operational semantics of a simple capability machine with sentry capabilities (Section 2.3).
- We define the Cerise program logic which can be used to formally verify the correctness of programs running on the capability machine. Our program logic is defined by instantiating the Iris framework [80], which provides an expressive separation logic with powerful reasoning principles, including, in particular, the notion of a *logical invariant* (Section 2.4).

- We define, using our program logic, the specification of what a “safe” capability and a “safe” program is. Intuitively, a capability (respectively, a program) is “safe” if it cannot be used to invalidate an invariant at the logical level. Hence, safe capabilities can be shared freely with unknown code. Safety of a capability is defined in the program logic as a unary logical relation (Section 2.5).
- We show that if a program only has access to “safe” values, then running the program itself is also “safe”. This is a global property of the capability machine, expressing its capability safety: it is not possible to increase one’s authority beyond what was available initially, independently of the sequence of instructions that one executes (Section 2.5). Concretely, the theorem takes the form of a contract that holds for arbitrary code,[1] and which can be combined in the program logic with manual proofs for trusted code. The last piece of the puzzle is then a so-called Adequacy theorem (Section 2.4), which relates invariants established in the program logic to the operational semantics of the machine. Given a concrete scenario (typically, a complete system mixing known verified code with unknown untrusted code), this makes it possible to obtain a theorem about the execution of the system which only depends on the operational semantics of the machine (not on the program logic).
- In Section 2.6 we then return to the examples from Section 2.2 and show how to use Cerise to formally prove that the desired memory compartmentalization results really do hold.
- In Section 2.7 we consider more sophisticated examples, which involve dynamic memory allocation. We focus on the low-level implementation of ML-like programs, and introduce a heap-based calling convention for closures implementing ML functions. We extend the earlier Adequacy theorem to account for dynamically allocated memory.
- In Section 2.8 we demonstrate how to use our methodology to establish correctness of object capability patterns (OCPs) from the literature. In particular, we consider the OCP of dynamic sealing, as presented by [152] in the context of a high-level language and we demonstrate that Cerise can be used to prove similar results about a low-level implementation of their example.
- Section 2.9 offers some perspectives on the relevance of our technical contributions and how we envision them being used in the development of secure systems.
- Finally, we discuss related work in Section 2.10.

This paper pedagogically introduces and explains the methodology underlying a sequence of recent research papers [142, 143, 62, 160], in the form of the Cerise program logic, but also contributes new material. The operational semantics, program logic and logical relation discussed in Sections 2.3, 2.4 and 2.5 are based on those used

[1] Because it holds for arbitrary code, we sometimes refer to this as a *universal contract.*



(a) Scenario 1: passing control to untrusted code

(b) Scenario 2: being called by untrusted code (possibly many times)

Figure 2.2: Two scenarios where a (trusted) component interacts with its (untrusted) context. The trusted component is represented with a plain background, while the untrusted context is represented with a red dotted background.

by [62] (but we have removed local and uninitialized capabilities as well as Kripke indexing for simplicity and instead added much more extensive explanations and proofs). Sections 2.2 and 2.6 are new; they provide a clear and accessible introduction to capability machine programming and our reasoning tools. The examples in Sections 2.7-2.8 are also new and represent a non-trivial verification effort.

The results and examples presented here have been fully formalized in Coq, and are available online: `https://github.com/logsem/cerise`. The development can also be viewed online at `https://logsem.github.io/cerise/journal/`; we use circled numbers such as ① to link directly to corresponding Coq formal statements in the following.

2.2 Programming with capabilities

Let us give a taste of how one might use capabilities when writing programs with the goal of enforcing some additional memory protection or encapsulation guarantees. We consider a fairly simple but quite general adversarial model, where we wish to verify the correctness of a *known component* interacting with a possibly adversarial *third-party component* whose code is unverified and untrusted.

In this section we detail two concrete example programs, which use capabilities in two different scenarios. In the first scenario, illustrated in Figure 2.2a, we consider a program that eventually passes control to the untrusted third-party code, but uses capabilities to protect a region of memory containing some secret data from being accessed by the untrusted code. In the second scenario (Figure 2.2b), we consider the case of a verified component being called by the third-party code. The goal is then for the verified component to use capabilities to protect (or "encapsulate") a piece of private memory, which it may access during its execution, but which should remain inaccessible to the unverified code.

2.2.1 Anatomy of a capability (in our model)

We are interested in a subset of the capabilities available in a CHERI capability machine. We thus work with a simplified machine model, featuring basic capabilities that are used to give access to a range of memory, as well as so-called "sealed entry" capabilities (abbreviated as "sentry" capabilities [175, §3.8]) that provide encapsulation features. The sentry capabilities were also called "enter" capabilities in earlier work, e.g., in the M-Machine by [28].

Concretely, we model capabilities as 4-tuples (p, b, e, a). In actual hardware, capabilities are encoded as fixed-size binary words, but here we abstract over their concrete representation.

Capability: (p, b, e, a)

$p \in \{\mathrm{O}, \mathrm{RO}, \mathrm{RX}, \mathrm{RW}, \mathrm{RWX}, \mathrm{E}\}$	permission
$b \in \mathit{Addr}$	base address
$e \in \mathit{Addr}$	end address
$a \in \mathit{Addr}$	current address

A capability (p, b, e, a) represents a machine word that can be used to access memory within the region $[b, e)$ delimited by its base address b and end address e. The permission p specifies what is possible to do within this memory range: permission O specifies that the capability actually gives no access rights, RO grants read-only access to memory, RX grants the right to read and execute the contents of the memory, RW gives read and write access, and RWX gives read, write, and execute access. Capabilities with permission E behave a bit differently (they are used to provide a form of encapsulation), and will be explained later in Section 2.2.4.

A capability is meant to be used as a pointer, and thus additionally points to a specific address a (typically, but not necessarily, belonging to the range $[b, e)$). Each time the capability is used to access memory, the machine will automatically check that a is between bounds b and e, and that the access is permitted according to p. From a capability (p, b, e, a) it is easy to derive another capability (p, b, e, a') pointing to a different address a' also within range $[b, e)$ – in other words, while a capability points to a specific address, it really holds authority over the whole region delimited by its beginning and end address.

Note that, on a capability machine, machine words can represent not only binary-encoded capabilities, but also traditional fixed-size integers. However, unlike on a traditional computer architecture, integers cannot be used as pointers. In other words, without holding a capability, one cannot access memory at all. In this paper, we rely on difference in notation to distinguish between capabilities and integers. In actual hardware, this is done by associating an extra one-bit tag to each word to distinguish capabilities from integers.

2.2.2 Sometimes, failure is a good thing

It is worth pointing out a sometimes counter-intuitive aspect of reasoning about security of programs running on a capability machine, especially for readers with a background in reasoning about safety in higher-level languages. For a high-level language, program safety can be seen as the absence of undefined behavior or runtime errors. For instance, an out-of-bounds array access is undefined behavior in C, and it leads to a runtime error, such as raising an exception, in memory-safe languages such as Rust or OCaml. We are instead interested in *security* properties for which a runtime failure can actually be considered a good thing.

Generally speaking, a low-level machine has many cases where it can fail at runtime, stopping the normal course of execution. In a standard (non-capability) machine, this can happen, e.g., if the machine attempts to execute an invalid instruction which cannot be decoded. The addition of capabilities only adds more possibilities for runtime faults: each time a capability is used, the capability machine will check that it has adequate permission and bounds, and raise a runtime fault otherwise.

Now, the point is that, from a security perspective, these additional runtime faults are a good thing. Using these additional checks, the capability machine turns dangerous behavior (out-of-bounds accesses leading to buffer overflow attacks, etc.) into proper faults before they can cause damage. Thus, for our purposes, it is always safe for the machine to fail: it means that an illegal operation may have been attempted, and the execution has been stopped in response.

Of course, when writing concrete programs, we will typically want to verify that we do not involuntarily trigger faults, as this would make our programs less useful. But when interacting with adversarial code, this is a possibility that we have to take into account anyway: we cannot prevent unknown code from shooting itself in the foot, e.g. by trying to access memory it does not have a valid capability for, or by decoding illegal instructions.

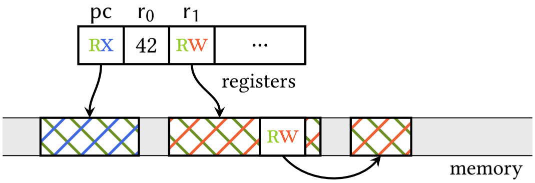
To sum up, in this work we reason about security properties that are not violated in the case of machine failure. This includes, for example, integrity of private data: no data can be compromised if the machine stops running. It is therefore useful to keep in mind that we consider failure to be trivially safe!

2.2.3 Restricting access to memory by constraining available capabilities

Consider Scenario 1 from Figure 2.2a: how can one write a program which passes control to untrusted code while protecting some secret data? That is, we wish to write a program that sets up capabilities so that its secrets are preserved even after it runs untrusted code.

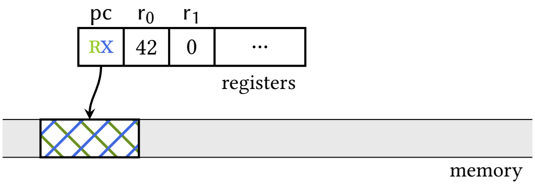
The key intuition is that, at any point of the execution, one can only access the part of memory that is accessible using the currently available capabilities. In other words, the authority of a running program comes from the set of capabilities which are transitivitely reachable from the CPU registers.

This is illustrated below, in a scenario where the pc register ("program counter") contains a capability with permission RX pointing to some memory region (containing the code of the program being executed), and register r_1 contains a capability with permission RW, pointing to a region of memory, which itself contains a RW capability pointing to another memory region. The collection of the "hatched" memory regions corresponds to the overall subset of memory accessible by the program.

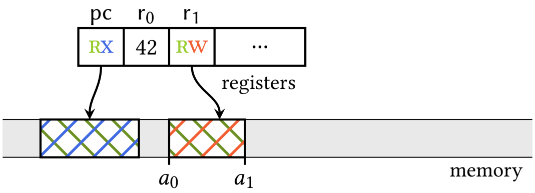


If one wishes to reduce the set of available memory or its associated access rights—for instance to protect secrets from being leaked to an adversary—then it is be enough to constrain the capabilities currently available. This can be done in a few different ways.

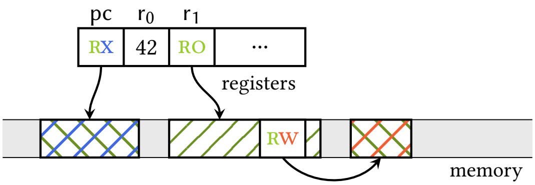
First, one can simply remove a capability from registers in order to remove access to the memory it was giving access to. For instance, after executing the instruction "`mov r1 0`", which overwrites the contents of register r_1 with the integer 0, one loses access to the memory regions which were previously accessible from the capability stored in that register.



Alternatively, it is possible to restrict the range of a capability to point to a smaller memory region. This changes the set of accessible memory to a subset of what was previously available. For instance, starting from our initial scenario and running the instruction "`subseg r1 a0 a1`" will change the range of the capability stored in register r_1 to $[a_0, a_1)$. (The machine will check that $[a_0, a_1)$ is indeed included in the range of the original capability.) In our example scenario (illustrated below), we then only keep the beginning of the region accessible from r_1, and this entails that the third region of memory becomes inaccessible, since it was only reachable from a capability stored at the end of the region accessible from r_1.



Finally, one can restrict the permission of a capability to a permission that grants less access rights. For instance, running the instruction "`restrict r1 RO`" in our initial scenario modifies the capability stored in r_1 to only grant read-only access to its corresponding memory region. Note that we still have read-write access to the last memory region, as we can still read the capability (with permission RW) pointing to it.



Example: sharing a sub-buffer with unknown code Using some of the mechanisms detailed above, we can implement a very simple program that shares a

```
; initially, PC = (RWX, code, end, code)
;             r0 = (unknown) pointer to the continuation
code:
  mov r1 PC                  ; r1 = (RWX, code, end, code)
  lea r1 [data-code]         ; r1 = (RWX, code, end, data)
  subseg r1 [data] [data+3] ; r1 = (RWX, data, data+3, data)
  jmp r0                     ; jump to unknown code: we give it read-write
                             ; access to the first 3 words of the data,
                             ; but not the secret value
data:
  ; the first 3 data words contain public data that will be passed
  ; to the unknown code (the "Hi" string)
  'H', 'i', 0,
  ; they are followed by secret data (the integer 42)
  42
end:
```

Figure 2.3: Program sharing a buffer with possibly adversarial code.

buffer with unknown, possibly adversarial, code while using capabilities to protect some data that would otherwise be vulnerable to buffer overflow attacks.

The assembly code for the program is shown in Figure 2.3. It consists of a code section containing the instructions of the program, followed by some data which (for simplicity) we simply assume to be statically allocated. The data section holds the zero-terminated string "Hi", which we wish to share with the untrusted code, and the integer 42 which represents our secret data.

Initially, we assume the program counter to contain a RWX capability for the whole region holding our program. This capability serves two purposes: it allows the machine to execute our program, but can also be manipulated by the program itself to derive a capability pointing to its own data. By convention, the register r_0 is assumed to contain a pointer to the continuation of the program, i.e. other code that the program will pass control to after it is done executing. As no assumption is made about the contents of r_0, it is conservatively assumed to point to unknown, arbitrary code.

Our program executes as follows: it first loads the capability held by the program counter into register r_1. Then, using the `lea` instruction, it changes the "current address" of the capability to point to the `data` label (`lea` modifies a capability by adding an offset to its "current address"). In assembly programs, we use the brackets notation `[...]` to denote an arithmetic expression that is computed statically when assembling the program.

At this point, the capability held in r_1 points to the start of the `"Hi"` string, but has (RWX) authority over the whole code and data section. This capability would be unsafe to share with the untrusted code, as they could simply use `lea` to increment the capability's current address past the end of the string, and obtain a valid capability to the secret value (thus performing a basic "buffer overflow" attack). To prevent this from happening, we use the `subseg` instruction to obtain a capability whose range of authority is restricted to the sub-buffer holding the `"Hi"` string. Finally, we pass control to the untrusted code by using the `jmp` instruction, loading the contents of register r_0 into pc.

This example illustrates that even a basic mode of use of capabilities (restricting them appropriately) can easily prevent buffer overflow attacks. In Section 2.6.1, we show how we can formally prove that, for any untrusted code, the value of the secret data will be equal to 42 at every step of the execution, including after control has been passed to the untrusted code. We have also developed a relational model, which can be used to prove that the secret value cannot even be read by the unknown code, but the details of this relational model are out of scope of this paper.

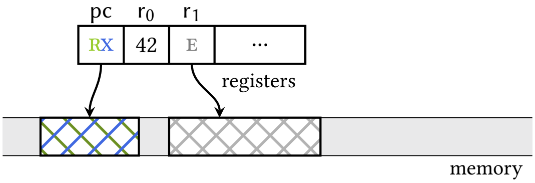
2.2.4 Securely encapsulating code and private capabilities

The previous example illustrates how to restrict available capabilities to prevent an adversary from accessing secret data. However, what if we additionally want our program to be called back by the untrusted code, as in Scenario 2.2b? In that case, when the trusted code gets invoked again we would like to recover access to the capabilities it previously had to its private state.

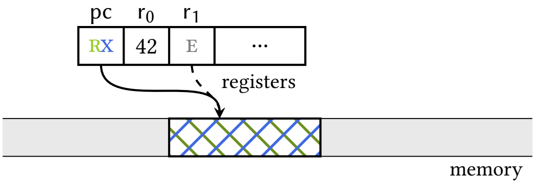
This is unfortunately not achievable with the capabilities that we have described so far. If we remove capabilities to private memory before passing control to untrusted code, then there is no way for us to get them back later on: the only capabilities we will get access to in a further invocation are capabilities the untrusted code itself has access to.

Sentry capabilities provide this missing feature. They implement a form of encapsulation that resembles the use of closures with encapsulated local state in high-level languages, and they allow implementing *compartments* which encapsulate private state and capabilities but can be called from untrusted code. From a security perspective, sentry capabilities allow setting up protection boundaries: the code executing before and after an invocation of a sentry capability has different authority and thus represent distrusting components. We denote sentry capabilities with permission E (for "Enter", a terminology originating from the M-machine [28]).

One typically creates a sentry capability pointing to a region of memory describing a compartment containing executable code and local state (or private capabilities to that local state). A sentry capability is opaque: it cannot be used to read or write to the memory region it points to, and it cannot be modified using `restrict` or `subseg`. It can thus be safely shared with untrusted third-parties: they will not be able to access the encapsulated code and data. In the figure below, the memory region pointed to by r_1 (hatched in gray) is not accessible for either reading or writing.



The only possible operation is to "invoke" the sentry capability using the `jmp` instruction, thus passing control to the code held in the region pointed to by the capability (in other words, "running" the compartment). When `jmp` is called on a sentry capability, it turns the capability into a capability with permission read-execute (RX) over the same memory region, and puts it into the program counter register pc. This simultaneously runs the encapsulated code, and gives access to the data and capabilities stored there, which were previously inaccessible. Running instruction `jmp r1` on the scenario of the previous figure leads to the machine state shown below.



Register pc now contains an RX capability to the previously opaque region, meaning that code contained in that region can execute. Furthermore, it may access other capabilities stored in that region, which can in turn be used to transitively access other private regions of memory.

Example: a counter compartment To illustrate the use of sentry capabilities, let us consider the example of a simple secure compartment implementing a counter. An instance of the counter holds a private memory cell containing the current (integer) value of the counter. Every time the code in the counter's compartment is invoked, it increases the value stored in the memory cell. Using a sentry capability, one can expose the counter to an untrusted context, without giving it direct access to the counter value.

It is worth pointing out that this is similar to the use of closures encapsulating local state in high-level languages. Typically, a similar counter program could be implemented in a high-level language as follows, using a function closure to encapsulate a reference holding the counter value.

$$\text{let } x = \text{ref } 0 \text{ in } (\lambda().\ x := !x + 1; !x)$$

As before, our actual counter program is implemented in assembly, and its code appears in Figure 2.4. Its implementation is divided into two parts. First, the code starting at label `init` (and ending at `code`) is used to set up the counter compartment; it is intended to run only once at the beginning of the program. Then, the region between `code` and `end` corresponds to the contents of the counter compartment itself, including its executable code (between `code` and `data`) and private data (between `data` and `end`).

The role of the initialization code is to create a sentry capability encapsulating the `code`–`end` region, and then pass control to the (untrusted) context, giving it access to the newly created sentry capability. Additionally, the initialization code stores at address `data` a capability giving read-write access to the compartment's region, and pointing to the counter's value at address `data`+1.

One might wonder why we have this extra indirection to the counter's value through the capability in `data`. Recall that after calling `jmp` on a sentry capability, the program counter is only provisioned with an RX capability. For the counter code to be able to actually increment the counter value (at address `data`+1), it needs to have write access to it. The additional RWX capability stored at address `data` by the initialization code is thus used to "promote" read access on the compartment's region into read-write access to that same region.

The code of the counter's compartment can then run many times, once each time the context chooses to invoke the sentry capability it got from the initialization code. At each invocation, the counter's implementation (at address `code`) reads the RWX capability stored in the data section, uses it to increment the value of the counter, and passes control back to its caller.

Let us walk through the details of the code. The initialization code is assumed to run starting with a program counter giving RWX access over the whole program region. The first four instructions derive, from the program counter, RWX capabilities pointing to addresses `data` and `data`+1. Then, using the `store` instruction, the capability (RWX, `init`, `end`, `data`+1) is stored at address `data`. Next, after using `lea` and `subseg` to adjust the address and bounds of the capability, a sentry capability is created pointing to the compartment's region [`code`, `end`). This is done using the `restrict` instruction, turning a capability with permission RWX into a capability with permission E. Register r_2 is then cleared, to make sure that the RWX capability

```
; initially, PC = (RWX, init, end, init)
;             r0 = (unknown) pointer to the context
init:
  mov r1 PC              ; r1 = (RWX, init, end, init)
  lea r1 [data-init]     ; r1 = (RWX, init, end, data)
  mov r2 r1              ; r2 = (RWX, init, end, data)
  lea r2 1               ; r2 = (RWX, init, end, data+1)
  store r1 r2            ; mem[data] <- (RWX, init, end, data+1)
  lea r1 [code-data]     ; r1 = (RWX, init, end, code)
  subseg r1 [code] [end] ; r1 = (RWX, code, end, code)
  restrict r1 E          ; r1 = (E, code, end, code)
  mov r2 0               ; r2 = 0
  jmp r0                 ; jump to unknown code: we only give it access
                         ; to an enter capability pointing to ’code’
; when ’code’ gets executed from the E capability,
;   PC = (RX, code, end, code)
;   r0 = (unknown) return pointer to the continuation
code:
  mov r1 PC              ; r1 = (RX, code, end, code)
  lea r1 [data-code]     ; r1 = (RX, code, end, data)
  load r1 r1             ; r1 = (RWX, init, end, data+1)
  load r2 r1             ; r2 = <counter value>
  add r2 r2 1            ; r2 = <counter value> + 1
  store r1 r2            ; mem[data+1] <- <counter value> + 1
  mov r1 0               ; r1 = 0
  jmp r0                 ; return to unknown code
data:
  0xFFFF, ; will be overwritten with (RWX, init, end, data+1), i.e.
          ; a read-write capability to the counter value
  0       ; our private data: the current value of the counter
end:
```

Figure 2.4: Program implementing a secure counter.

pointing to the counter value is not leaked to the context. Finally, the initialization code jumps to the pointer in r_0, which by convention points to the context.

The compartment's code (starting at address `code`) then gets executed each time the context invokes the sentry capability. Because we have only shared a sentry capability (E, `code`, `end`, `code`) with the context, we know that when the compartment gets executed, the program counter must contain (RX, `code`, `end`, `code`). By reading the program counter, the first two instructions of the code then derive an RX capability pointing to address `data`, and use it (with `load`) to read the capability that was stored there, granting RWX access to `data`+1. The subsequent `load`, `add` and `store` instructions use this second capability to increment the value of the counter. Finally, before returning to the context by jumping to r_0, the program takes care of clearing register r_1, overwriting its contents with 0. This is quite crucial, as otherwise an RWX capability would be leaked to the context, giving it direct access to the counter's private state!

To sum up, our example program carefully selects which capabilities it shares with unknown code, and leverages the encapsulation properties of sentry capabilities provided by the machine. Consequently, it should seem clear, at least informally, that the integrity of the counter's value is guaranteed through the execution. More precisely, we should be able to formally prove some invariant about it: for instance, that it is nonnegative at every step of the execution, for any untrusted context. In Section 2.6.2, we show in more detail how to formally establish this property.

In this section, we have showcased how one might program with capabilities in order to obtain security guarantees, and make it possible to interact with adversarial code while protecting private data and invariants.

In the rest of this paper, we show how we can make the intuitions that we have developed so far more precise, and formally prove capability safety for machine code programs that interact with untrusted code. Namely:

- We expect to have some concrete known code, which has some private data and invariants, and interacts with untrusted code.
- We formally define the operational semantics of the capability machine that we consider (Section 2.3). This precisely defines the behavior of the machine on which the rest of our framework is built.
- Then we develop (Section 2.4) a program logic which supports formally verifying correctness properties about known code. Given some verified known code, we would then like to be able to conclude some result about a complete execution of the machine, when it runs a combination of the known code and some arbitrary untrusted code.
- To that end we need a way of formally capturing the fact that the machine

effectively restricts the behavior of arbitrary code at runtime, by limiting the capabilities it has access to. We do this (Section 2.5) by defining a logical relation capturing "capability safety" of arbitrary code.

- By combining the Adequacy theorem of our program logic and the Fundamental theorem of our logical relation, we can prove safety of concrete examples (Section 2.6) and obtain theorems about complete executions of the machine.

2.3 Operational semantics of a capability machine

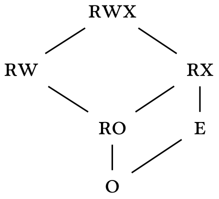
The very basis of our framework is a formal description of the capability machine we consider: which instructions it supports, and its behavior when it runs and executes programs. Technically speaking, this description corresponds to the operational semantics of the machine, upon which the program logic described next in Section 2.4 is built.

Our capability machine draws inspiration from CHERI [175], albeit in a simplified form, and only covers a subset of the features found in CHERI machines. Compared to a realistic CHERI machine, we consider a number of simplifications: our instruction set is minimal, our machine does not have virtual memory or different privilege levels, machine words can store unbounded integers, every instruction can be encoded in a single machine word, we do not consider memory alignment issues, and we abstract away from the binary encoding of capabilities. Nevertheless, our semantics does capture many of the aspects that make reasoning about machine code programs challenging: our machine has a finite amount of memory, a fixed number of registers, and there are no distinctions between code and data nor structured control flow for programs, owing to the fact that program instructions are simply encoded and stored in memory as normal integers.

Figure 2.5 gives the basic definitions that will play a role in the operational semantics of machine instructions. The set of memory addresses Addr is finite, and corresponds to the integer range $[0, \mathrm{AddrMax}]$. A memory word $w \in \mathrm{Word}$ is either an (unbounded) integer or a capability c. Capabilities are of the form (p, b, e, a), giving access to the memory range $[b, e)$ with permission p, while currently pointing to a. The permissions p are ordered according to the lattice appearing at the top-right of the figure, inducing a bottom-to-top partial order $\preccurlyeq$ on permissions. There are six different permissions; the null (O), read-only (RO), enter (E), read-write (RW), read-execute (RX) and read-write-execute (RWX) permissions.

The state of the machine is modeled by the semantics as a pair of an execution state s and a configuration φ. An execution state flag indicates whether the machine is presently running (Running), has successfully halted (Halted), or has stopped execution by raising an error (Failed). A configuration φ contains the state of the

$$
\begin{array}{llll}
a & \in \text{Addr} & \triangleq & [0, \text{AddrMax}] \\
p & \in \text{Perm} & ::= & \textsc{o} \mid \textsc{e} \mid \textsc{ro} \mid \textsc{rx} \mid \textsc{rw} \mid \textsc{rwx} \\
c & \in \text{Cap} & \triangleq & \{(p, b, e, a) \mid b, e, a \in \text{Addr}\} \\
w & \in \text{Word} & \triangleq & \mathbb{Z} + \text{Cap} \\
reg & \in \text{Reg} & \triangleq & \text{RegName} \rightarrow \text{Word} \\
m & \in \text{Mem} & \triangleq & \text{Addr} \rightarrow \text{Word} \\
s & \in \text{ExecState} & ::= & \text{Running} \mid \text{Halted} \mid \text{Failed} \\
\varphi & \in \text{ExecConf} & \triangleq & \text{Reg} \times \text{Mem}
\end{array}
$$



Lattice defining the $\preccurlyeq$ relation.

(We have $p_1 \preccurlyeq p_2$ if there is a path going up from p_1 to p_2 in the diagram.)

$$r \in \text{RegName} ::= \text{pc} \mid \text{r}_0 \mid \text{r}_1 \mid \ldots \mid \text{r}_{31} \qquad \rho \in \mathbb{Z} + \text{RegName}$$

$$
\begin{array}{ll}
i \; ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{mov}\ r\ \rho \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \\
& \texttt{lt}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{restrict}\ r\ \rho \mid \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{isptr}\ r\ r \mid \texttt{getp}\ r\ r \mid \\
& \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Figure 2.5: Base definitions for the machine's words, state, and instructions.

registers $\varphi.reg$ and the memory $\varphi.mem$. A register file *reg* consists of a map from register names r to machine words, while the memory m maps addresses to words.

Next, Figure 2.5 shows the list of instructions of our machine. An instruction i typically operates on register names r, but can also sometimes take integer values as parameters; ρ denotes an instruction parameter which can be either a register name or a constant integer. Our machine features general purpose registers (r_0 – r_{31}), on top of the pc register, which points to the address in memory where the currently executing instruction is stored. (Technically speaking, pc must point to a memory cell containing an integer which can be successfully decoded into an instruction.) pc should therefore always contain a capability with at least permission RX; in any other case, the machine fails immediately.

Figure 2.6 defines the small-step operational semantics for the capability machine. The rule ExecSingle describes how a single instruction gets executed. An execution step requires an executable and in-bounds capability in the pc register, and fails otherwise. It expects the memory cell pointed to by the capability to store an integer z, decodes it into an instruction and executes the instruction on the current state φ; the new configuration is denoted $[\![decode(z)]\!](\varphi)$. The table making up most of Figure 2.6 defines the operational behavior $[\![i]\!](\varphi)$ for each instruction i of the machine.

The auxiliary functions present in Figure 2.6 are defined in Figure 2.7. Most instructions use the auxiliary function updPC to increment the pc register after

ExecSingle

$$(\text{Running}, \varphi) \to \begin{cases} [\![\mathit{decode}(z)]\!](\varphi) & \text{if } \ \varphi.\text{reg}(\text{pc}) = (p, b, e, a) \land b \leq a < e \land \\ & \quad p \in \{\textsc{rx}, \textsc{rwx}\} \land \varphi.\text{mem}(a) = z \\ (\text{Failed}, \ \varphi) & \text{otherwise} \end{cases}$$

i	$[\![i]\!](\varphi)$	Conditions
`fail`	(Failed, φ)	
`halt`	(Halted, φ)	
`mov` r ρ	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$w = \text{getWord}(\varphi, \rho)$
`load` r_1 r_2	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto w])$	$\varphi.\text{reg}(r_2) = (p, b, e, a)$ and $w = \varphi.\text{mem}(a)$ and $b \leq a < e$ and $p \in \{\textsc{ro}, \textsc{rx}, \textsc{rw}, \textsc{rwx}\}$
`store` r ρ	$\text{updPC}(\varphi[\text{mem}.a \mapsto w])$	$\varphi.\text{reg}(r) = (p, b, e, a)$ and $b \leq a < e$ and $p \in \{\textsc{rw}, \textsc{rwx}\}$ and $w = \text{getWord}(\varphi, \rho)$
`jmp` r	$(\text{Running}, \varphi[\text{reg.pc} \mapsto \mathit{newPc}])$	$\mathit{newPc} = \text{updatePcPerm}(\varphi.\text{reg}(r))$

`jnz` r_1 r_2	if $\varphi.\text{reg}(r_2) \neq 0$
	then $(\text{Running}, \varphi[\text{reg.pc} \mapsto \mathit{newPc}])$
	else $\text{updPC}(\varphi)$
		$\mathit{newPc} = \text{updatePcPerm}(\varphi.\text{reg}(r_1))$

`restrict` r ρ	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, b, e, a)$ and $p' = \text{decodePerm}(\text{getWord}(\varphi, \rho))$ and $p' \preccurlyeq p$ and $w = (p', b, e, a)$
`subseg` r ρ_1 ρ_2	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, b, e, a)$ and for $i \in \{1, 2\}$, $z_i = \text{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and $b \leq z_1$ and $0 \leq z_2 \leq e$ and $p \neq \textsc{e}$ and $w = (p, z_1, z_2, a)$
`lea` r ρ	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, b, e, a)$ and $z = \text{getWord}(\varphi, \rho)$ and $p \neq \textsc{e}$ and $w = (p, b, e, a + z)$
`add` r ρ_1 ρ_2	$\text{updPC}(\varphi[\text{reg}.r \mapsto z])$	for $i \in \{1, 2\}$, $z_i = \text{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and $z = z_1 + z_2$
`sub` r ρ_1 ρ_2	$\text{updPC}(\varphi[\text{reg}.r \mapsto z])$	for $i \in \{1, 2\}$, $z_i = \text{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and $z = z_1 - z_2$
`lt` r ρ_1 ρ_2	$\text{updPC}(\varphi[\text{reg}.r \mapsto z])$	for $i \in \{1, 2\}$, $z_i = \text{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and if $z_1 < z_2$ then $z = 1$ else $z = 0$
`getp` r_1 r_2	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto z])$	$\varphi.\text{reg}(r_2) = (p, \_, \_, \_)$ and $z = \text{encodePerm}(p)$
`getb` r_1 r_2	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto b])$	$\varphi.\text{reg}(r_2) = (\_, b, \_, \_)$
`gete` r_1 r_2	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto e])$	$\varphi.\text{reg}(r_2) = (\_, \_, e, \_)$
`geta` r_1 r_2	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto a])$	$\varphi.\text{reg}(r_2) = (\_, \_, \_, a)$
`isptr` r_1 r_2	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto z])$	if $\varphi.\text{reg}(r_2) = (\_, \_, \_, \_)$ then $z = 1$ else $z = 0$
_	(Failed, φ)	otherwise

Figure 2.6: Operational semantics: execution of a single instruction.

$$\mathrm{updPC}(\varphi) = \begin{cases} (\mathsf{Running}, \varphi[\mathrm{reg.pc} \mapsto (p, b, e, a + 1)]) & \text{if } \varphi.\mathrm{reg}(\mathrm{pc}) = (p, b, e, a) \\ (\mathsf{Failed}, \varphi) & \text{otherwise} \end{cases}$$

$$\mathrm{getWord}(\varphi, \rho) = \begin{cases} \rho & \text{if } \rho \in \mathbb{Z} \\ \varphi.\mathrm{reg}(\rho) & \text{if } \rho \in \mathrm{RegName} \end{cases}$$

$$\mathrm{updatePcPerm}(w) = \begin{cases} (\textsc{rx}, b, e, a) & \text{if } w = (\textsc{e}, b, e, a) \\ w & \text{otherwise} \end{cases}$$

Figure 2.7: Operational semantics: auxiliary definitions.

their proper operations. Because the address space is finite, pointer arithmetic such as incrementing pc can result in illegal addresses. To avoid notational clutter, we will always write as if arithmetic operations succeed, and consider that otherwise the machine transitions to a Failed state. The auxiliary function getWord is used to get the value corresponding to the argument ρ of an instruction: either its corresponding integer value if it is an immediate integer, or the contents of the corresponding register if it is a register name. The auxiliary function updatePcPerm is used in the definition of the behavior of the `jmp` and `jnz` instructions to unseal sentry capabilities. As mentioned previously, an additional effect of these jump instructions is to unseal sentry (E) capabilities into RX capabilities.

We now describe the semantics of the instructions of the machine, as formally defined in the table of Figure 2.6. The `fail` and `halt` instructions stop the execution of the machine, in the Failed and Halted state respectively. `mov` r ρ copies ρ (either an immediate value or the contents of the corresponding register name) into register r. The instructions `load` and `store` allow reading and writing memory: `load` r_1 r_2 reads the value pointed to by the capability in r_2 provided it has the permission R and points within its bounds; `store` r ρ stores ρ to the location pointed to by the capability in r provided it has the W permission and points within bounds. The `jmp` and `jnz` instructions correspond to an unconditional and conditional jump respectively, thus loading the provided capability into pc. Using updatePcPerm, in the case of a sentry (E) capability, they unseal it into a RX capability first. Three instructions allow deriving new capabilities from existing ones. `restrict` r ρ allows restricting the permission of a capability (where ρ provides an integer encoding of the desired permission), provided it is less permissive than the current permission according to $\preccurlyeq$. `subseg` r ρ_1 ρ_2 restricts the range of authority of the capability stored in r, provided it is a subset of the current range of the capability. `lea` r ρ modifies the current address of the capability in r, by adding to it the integer offset ρ. As should be expected, `subseg` and `lea` fail for sentry capabilities. Arithmetic operations are provided by the `add`, `sub` and `lt` instructions, which implement addition, subtraction, and comparison on integers, respectively. Finally, a number of instructions allow inspecting machine words and capabilities. `isptr` can be used to query whether a machine word is an integer or a capability, and `getp`, `getb`, `gete`, and `geta` return the different parts of a capability (permission, bounds and address). (More precisely, `getp` returns an integer encoding the permission, as given by encodePerm.) If any of the capability checks for an instruction are not satisfied, the machine fails.

An important aspect of our operational semantics is how it explicitly accounts for errors: when a capability check fails (for instance when a program tries to use a capability outside of its range), the semantics does not get stuck (meaning that it would not be able to reduce): instead, it explicitly transitions to a state with the `Failed` execution state flag.

2.4 Program logic

The operational semantics presented in the previous section formally define the behavior of our machine when it runs and executes code. Based on that, we expect to be able to formally verify concrete programs running on the machine.

The most direct approach would be to manually establish properties of sequences of reduction steps, based on the sole definition of the operational semantics. We do not follow this approach, because it would quickly become very tedious even for simple programs.

Instead, we draw from previous research in program logics and separation logic, and define Cerise: a program logic which provides a convenient framework in which to modularly reason about programs running on our machine. Indeed:

- It is typically more convenient to devise a system of proof rules for programs, rather than work directly at the level of abstraction provided by the bare operational semantics. Such rules form a program logic, which can be proved sound according to the operational semantics, and then can be used to verify properties of concrete programs.
- Separation logic, a family of program logics, has been widely used to reason about programs manipulating shared mutable state (such as memory). On our capability machine, not only do all programs access a mutable shared memory, but programs are themselves represented as unstructured data in memory; so the use of separation logic is particularly called for. Separation logic enables modular reasoning about programs that operate only on a sub-part of the global state, allowing them to be freely composed with programs that operate on a disjoint part of the state.

The first step is to consider what part of the machine state should be described by

$$\begin{array}{lr}
P, Q \in \mathit{iProp} ::= & \\
\quad \mathrm{True} \mid \mathrm{False} \mid \forall x.\, P \mid \exists x.\, P \mid \ldots & \text{higher-order logic}\\
\quad \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ulcorner \phi \urcorner \mid \square P \mid \triangleright P & \text{separation logic}\\
\quad \mid \mathrm{a} \mapsto w \mid r \Mapsto w \mid \vec{a} \mapsto \vec{l} & \text{machine resources}\\
\quad \mid \boxed{P} & \text{invariants}\\
\quad \mid \langle P \rangle \rightarrow \langle s.\, Q \rangle \mid \{P\} \rightsquigarrow \{s.\, Q\} \mid \{P\} \rightsquigarrow \bullet & \text{program logic}
\end{array}$$

Figure 2.8: The syntax of our program logic.

separation logic assertions. Here, the machine state consists of both the machine memory and the machine registers. Indeed, it is useful to modularly reason about programs operating on both a subset of memory and a subset of the available registers.

Technically speaking, we build the Cerise program logic on top of the Iris framework [80], which provides us with additional useful features, such as invariants. In the following we introduce both the basic separation logic assertions describing the machine state and additional features inherited from Iris (Section 2.4.1). Then, we describe the rules that are used to specify the execution of machine instructions and programs (Section 2.4.2).

Note that the program logic is, in a sense, only a technical device. The end goal is to obtain theorems that only refer to reductions in the operational semantics of our machine. To that end, we present (Section 2.4.3) an Adequacy theorem for our logic, which allows us to "extract" a correctness theorem expressed in terms of the operational semantics of the machine from a proof established in the program logic.

2.4.1 Basic resources

Figure 2.8 shows the syntax of our Cerise program logic based on Iris. We write *iProp* for the universe of propositions. These feature the standard connectives of higher-order logic and separation logic, including the separating conjunction $*$ and the magic wand $-\!\!*$ (read as an implication). The proposition $\ulcorner \phi \urcorner$ asserts that the pure proposition ϕ holds, where ϕ is a proposition from the meta logic.

Iris assertions can be divided in two categories: *ephemeral* assertions and *persistent* assertions. Ephemeral assertions describe facts or resources that are available at a given point but might become false or unavailable later. Persistent assertions describe facts that never cease to be true. The assertion $\square P$, read "persistently P", is persistent, and asserts ownership over resources whose duplicable part satisfies P. In other words, $\square P$ is like P except that it does not assert any exclusive ownership over resources.

As the knowledge associated with a persistent assertion can never be invalidated, persistent assertions can be freely duplicated.

The modality $\triangleright P$ expresses (roughly) that the assertion P holds after one "logical step" of execution. In this paper, we mainly use it to define recursive predicates using guarded recursion. It is not necessary to understand how the modality behaves in detail and the reader can safely ignore it for the most part and just recall that it supports an abstract accounting of execution steps.

Our logic includes resources (predicates) that describe parts of the current state of the machine. The assertion $\mathrm{a} \mapsto w$ expresses that the memory cell at address a contains the machine word w. Furthermore, this assertion should be read as giving *unique ownership* over location a, giving the right to freely read and update the corresponding memory cell. Similarly, the assertion $r \Mapsto w$ asserts ownership of a CPU register r containing the word w. We write $\vec{a} \mapsto \vec{l}$ for the ownership of contiguous memory cells at addresses $\vec{a}$ containing $\vec{l}$.

A key feature of the logic is the notion of an invariant. The assertion $\boxed{P}$ asserts that P should hold at all times, now and for every future step of the execution (where P can be any separation logic assertion). An invariant is a persistent assertion. An invariant $\boxed{P}$ can be created (or "allocated") by handing over the resources for P, turning them into $\boxed{P}$. Then, whenever we know that $\boxed{P}$ holds, we can get access to the resources P held in the invariant, but only for the duration of one program step. Indeed, since the invariant must hold at every step of the execution, when accessing its resources, one needs to show that it holds again no later than one program step after. A more precise rule for accessing invariants is given next in Section 2.4.2 (rule Inv).

2.4.2 Program specifications

The predicates for machine resources we just presented allow describing the state of the machine. Our logic, moreover, includes assertions that can be used to specify machine executions, similar to *Hoare triples* used in program logics for high-level languages. Because we work with a low-level machine (where code is located in memory), we distinguish between three different types of program specifications:

$$\begin{array}{ll} \langle P \rangle \rightarrow \langle s.\, Q \rangle & \text{single instruction} \\ \{P\} \rightsquigarrow \{s.\, Q\} & \text{code fragment} \\ \{P\} \rightsquigarrow \bullet & \text{complete safe execution.} \end{array}$$

In each case, P and Q are separation logic assertions describing the state of the machine (registers and memory). P corresponds to a pre-condition, Q a post-condition, and s binds in Q the corresponding execution state (of type ExecState, see Figure 2.5).

Informally, $\langle P \rangle \rightarrow \langle s.\, Q \rangle$ holds if, starting from a machine state satisfying P, the machine can execute one step of computation, and reach a state satisfying Q in an execution state s. The predicate $\{P\} \rightsquigarrow \{s.\, Q\}$ holds if, starting from a state satisfying P, then the machine can diverge (i.e. loop) or reach a state satisfying Q in an execution state s. This is typically used to describe the execution of a code fragment. Finally, $\{P\} \rightsquigarrow \bullet$ holds if, starting from a machine state satisfying P, then the machine loops forever or runs until completion, ending in either a Halted or Failed state. In this case, we say that the initial configuration described by P is *safe*. (Not every configuration is safe: the resources in P describing registers and memory must suffice for the machine to run and execute the code pointed to by pc: we do not have $\{pc \Mapsto w\} \rightsquigarrow \bullet$ in general.)

Additionally, these three specifications *require the logical invariants to be preserved at every step of the execution*. This requirement is implicit in the definition of invariants, but it is a crucial reasoning principle that we will leverage.

Echoing back to Section 2.2.2, note that our program specification for a complete safe execution allows the program to fail (or diverge). Indeed, we will capture the preservation of security properties by preserving *invariants* throughout execution and having the machine fail is both fine (invariants are trivially preserved when the machine ends up in a failure state) and unavoidable (we cannot prevent unknown code from triggering a capability check failure). Similar considerations apply for divergence.

Notations In the rest of the paper, we will rely on a couple of additional notations when writing program specifications. Because we often want to reason about the case where an instruction (or program fragment) does not fail, we write $\langle P \rangle \rightarrow \langle Q \rangle$ (respectively $\{P\} \rightsquigarrow \{Q\}$) to denote a resulting execution state equal to Running:

$$\begin{aligned} \langle P \rangle \rightarrow \langle Q \rangle \quad &\triangleq \quad \langle P \rangle \rightarrow \left\langle s.\, \ulcorner s = \mathsf{Running} \urcorner * Q \right\rangle \\ \{P\} \rightsquigarrow \{Q\} \quad &\triangleq \quad \{P\} \rightsquigarrow \left\{ s.\, \ulcorner s = \mathsf{Running} \urcorner * Q \right\}. \end{aligned}$$

When writing pre- and post-conditions, we will often need to include a points-to resource describing the contents of the pc register. We introduce a short-hand notation for that purpose, and write $w; P$ to assert P and additionally that pc is set to w:

$$w; P \quad \triangleq \quad \mathrm{pc} \Mapsto w * P$$

Using these two notations, the specification for a single instruction, in a case where it does not fail, is written as $\langle w_0; P \rangle \rightarrow \langle w_1; Q \rangle$ (typically, we have $w_1 = w_0 + 1$, except in the case of the `jmp` and `jnz` instructions, or when explicitly writing to the pc register).

Properties Our program specifications satisfy the well-known "frame rule" of separation logic, which permits local reasoning, and asserts that it is always possible to extend a specification by adding arbitrary resources not accessed by the program.

$$\begin{array}{lll}
\textsc{FragFrame} & \textsc{StepFrame} & \textsc{FullFrame} \\
\dfrac{\{P\} \rightsquigarrow \{s.\, Q\}}{\{P * R\} \rightsquigarrow \{s.\, Q * R\}} &
\dfrac{\langle P\rangle \longrightarrow \langle s.\, Q\rangle}{\langle P * R\rangle \longrightarrow \langle s.\, Q * R\rangle} &
\dfrac{\{P\} \rightsquigarrow \bullet}{\{P * R\} \rightsquigarrow \bullet}
\end{array}$$

Program specifications can also be composed using sequencing rules. In order to establish a specification of the form $\{P\} \rightsquigarrow \{s.\, Q\}$, one typically uses single-instructions rules ($\langle R\rangle \longrightarrow \langle s.\, S\rangle$) in a sequence, one for each instruction of the relevant code block. Specifications for two program fragments that follow each other can also be combined to obtain a specification for the sequence of the two fragments. We prove general sequencing rules for our three kind of specifications; for simplicity, we only reproduce here restricted rules that deal with successful executions (relying on the notations introduced before):

$$\begin{array}{ll}
\textsc{SeqFrag} & \textsc{SeqFull} \\
\dfrac{\{P\} \rightsquigarrow \{Q\} \qquad \{Q\} \rightsquigarrow \{R\}}{\{P\} \rightsquigarrow \{R\}} &
\dfrac{\{P\} \rightsquigarrow \{Q\} \qquad \{Q\} \rightsquigarrow \bullet}{\{P\} \rightsquigarrow \bullet}
\end{array}$$

$$\begin{array}{ll}
\textsc{StepFull} & \textsc{StepFrag} \\
\dfrac{\langle P\rangle \longrightarrow \langle Q\rangle \qquad \{Q\} \rightsquigarrow \bullet}{\{P\} \rightsquigarrow \bullet} &
\dfrac{\langle P\rangle \longrightarrow \langle Q\rangle \qquad \{Q\} \rightsquigarrow \{R\}}{\{P\} \rightsquigarrow \{R\}}
\end{array}$$

When reasoning about a single execution step, one can additionally access resources held in known invariants. This is done using the Inv rule, given below: [2]

$$\begin{array}{l}
\textsc{Inv} \\
\dfrac{\langle P * \triangleright R\rangle \longrightarrow \langle s.\, Q * \triangleright R\rangle}{\boxed{R} \vdash \langle P\rangle \longrightarrow \langle s.\, Q\rangle}
\end{array}$$

Example specifications As illustrative examples, Figure 2.9 shows specifications for the `subseg`, `load` and `store` instructions, as well as the `rclear` macro which is used to clear the contents of a number of specified registers. The first rule shows a specification for the `subseg` instruction. It states that if the program counter contains a capability pointing to a memory location a_{pc}, if that location contains an integer n which decodes into `subseg` r z_1 z_2, and if the register r contains a capability, then assuming that the program counter is valid (ValidPC(...)) and that z_1 and z_2 are

[2] For clarity of the presentation, we choose to omit additional details related to Iris invariant namespaces and masks. We refer to the Coq development for the full details ②.

$$\frac{\begin{array}{c}\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \\ \text{ValidSubseg}(p, b, e, z_1, z_2) \qquad \text{decode}(n) = \texttt{subseg}\ r\ z_1\ z_2\end{array}}{\begin{array}{l}\big\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ; \quad a_{pc} \mapsto n * r \Mapsto (p, b, e, a) \big\rangle \longrightarrow \\ \big\langle (p_{pc}, b_{pc}, e_{pc}, \textcolor{red}{a_{pc} + 1})\ ;\ a_{pc} \mapsto n * r \Mapsto (p, \textcolor{red}{z_1}, \textcolor{red}{z_2}, a) \big\rangle\end{array}}$$

$$\frac{\begin{array}{c}\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \\ \neg\text{ValidSubseg}(p, b, e, z_1, z_2) \qquad \text{decode}(n) = \texttt{subseg}\ r\ z_1\ z_2\end{array}}{\begin{array}{r}\big\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\ a_{pc} \mapsto n * r \Mapsto (p, b, e, a) \big\rangle \longrightarrow \\ \big\langle s.\ \ulcorner s = \mathsf{Failed} \urcorner * \big((p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\ a_{pc} \mapsto n * r \Mapsto (p, b, e, a) \big) \big\rangle\end{array}}$$

$$\frac{\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \qquad \text{ValidLoad}(p, b, e, a) \qquad \text{decode}(n) = \texttt{load}\ \mathit{dst}\ \mathit{src}}{\begin{array}{l}\big\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ; \quad a_{pc} \mapsto n * \mathit{dst} \Mapsto - * \mathit{src} \Mapsto (p, b, e, a) * a \mapsto w \big\rangle \longrightarrow \\ \big\langle (p_{pc}, b_{pc}, e_{pc}, \textcolor{red}{a_{pc} + 1})\ ;\ a_{pc} \mapsto n * \mathit{dst} \Mapsto \textcolor{red}{w} * \mathit{src} \Mapsto (p, b, e, a) * a \mapsto w \big\rangle\end{array}}$$

$$\frac{\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \qquad \text{ValidStore}(p, b, e, a) \qquad \text{decode}(n) = \texttt{store}\ \mathit{dst}\ \mathit{src}}{\begin{array}{l}\big\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ; \quad a_{pc} \mapsto n * \mathit{dst} \Mapsto (p, b, e, a) * \mathit{src} \Mapsto w * a \mapsto - \big\rangle \longrightarrow \\ \big\langle (p_{pc}, b_{pc}, e_{pc}, \textcolor{red}{a_{pc} + 1})\ ;\ a_{pc} \mapsto n * \mathit{dst} \Mapsto (p, b, e, a) * \mathit{src} \Mapsto w * a \mapsto \textcolor{red}{w} \big\rangle\end{array}}$$

$$\frac{\forall i \in [0, n),\ \text{ValidPC}(p, b, e, a_i) \qquad n = \text{length}(\text{rclear\_instrs}\ l)}{\begin{array}{l}\big\{ (p, b, e, a_0);\ \mathop{\huge *}_{r \in l} r \Mapsto - * \mathop{\huge *}_{i \in [0,n)} a_i \mapsto (\text{rclear\_instrs}\ l)[i] \big\} \rightsquigarrow \\ \big\{ (p, b, e, \textcolor{red}{a_n});\ \mathop{\huge *}_{r \in l} r \Mapsto \textcolor{red}{0}\ * \mathop{\huge *}_{i \in [0,n)} a_i \mapsto (\text{rclear\_instrs}\ l)[i] \big\}\end{array}}$$

$$\begin{array}{lcl}\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) & \triangleq & \textsc{rx} \preccurlyeq p_{pc} \land b_{pc} \leq a_{pc} < e_{pc} \\ \text{ValidSubseg}(p, b, e, z_1, z_2) & \triangleq & p \neq \textsc{e} \land b \leq z_1 \land 0 \leq z_2 \leq e \\ \text{ValidLoad}(p, b, e, a) & \triangleq & \textsc{ro} \preccurlyeq p \land b \leq a < e \\ \text{ValidStore}(p, b, e, a) & \triangleq & \textsc{rw} \preccurlyeq p \land b \leq a < e \\ \text{rclear\_instrs}\ l & \triangleq & \text{map}\ (\lambda r.\ \text{encode}(\texttt{move}\ r\ 0))\ l\end{array}$$

Figure 2.9: Specifications for the machine instructions `subseg`, `load` and `store` and for the `rclear` macro that sets a given list of registers to zero. Changes to the machine state are highlighted in red.

valid new bounds (ValidSubseg(...)), the machine successfully increments the program counter and restricts the capability held in register r with new bounds z_1 and z_2.

The second rule is also a specification for `subseg`, but in a case where it fails a bound check, i.e. ValidSubseg(p, b, e, z_1, z_2) does not hold. (For instance, when the new bounds z_1 and z_2 would allow accessing more memory than what is available through the original capability.) Then, the rule does give us a specification for an execution step, but with a resulting execution state of `Failed`, meaning that the execution cannot continue afterwards.

The third and fourth rules give specifications for the `load` and `store` instructions (in non-failing cases). The specification for `load` states that `load` *dst src* loads a word from memory pointed to by a capability in register *src* and stores its contents in register *dst*. The specification for `store` states that `store` *dst src* reads a word from the *src* register and writes it into the memory location pointed to by the capability in *dst*.

Note that these specifications for `subseg`, `load` and `store` are not in fact the most general specifications for these instructions. They assume that some side-conditions hold, and specify the behavior of the instruction in the case of either a "normal" successful execution, or a failing one. These specifications are typically useful for reasoning about the correctness of a concrete program. We have also proved in Coq (e.g., ③ for the `subseg` instruction) "most general" specifications, covering in one lemma all possible cases for a given instructions. These are useful for deriving the more specific rules shown previously. Furthermore, we use them directly in the proof of the Fundamental Theorem (Theorem 2.2), for specifying the behavior of arbitrary instructions that might or might not fail.

The last rule of Figure 2.9 shows a derivable specification for a program composed of several instructions, the `rclear` macro. This macro (meaning, a small program that is typically inserted inline as part of a larger program) clears a number of registers by setting their content to 0. It is parameterized by a list l of register names, and its code consists of a sequence of instructions `move` r `0` for each register name r in l. We state `rclear`'s specification using the program specification for code fragments. This specification is provable using the basic reasoning rules for `move`. It requires that the body of the macro ("rclear_instrs l") is laid out contiguously in memory range $[a_0, a_n)$, while the program counter initially points to a_0. When the program counter eventually points to a_n, the address immediately after the macro's instructions, then all the registers in l have been cleared and now contain 0. (The "big star" $\ast$ denotes an iterated separating conjunction, here over the registers r in list l.)

2.4.3 Adequacy theorem

After establishing program specifications and properties at the level of our program logic, we ultimately want to transfer these results into properties of a program execution at the level of the operational semantics of the bare machine. Generally speaking, we prove using the rules of the Iris logic a statement of the form $\boxed{P} \vdash Q$, where P and Q are Iris propositions (read "Q holds assuming invariant P"). From this, we want to deduce that some mathematical proposition Φ holds (as a Coq proposition, in our case), where Φ describes some property of the machine execution expressed in terms of its operational semantics.

Because we are interested in establishing *invariants* about a program execution, we typically want to obtain in Φ that at every step of the execution, the state of the machine satisfies an invariant corresponding to the Iris assertion P.

Deriving mathematical facts from Iris proof derivations is made possible thanks to the so-called *adequacy* theorem of Iris [4]. This theorem has a very general but intricate statement. In this section, we describe a simpler but more specialized adequacy theorem for our capability machine, which we can use to reason about the examples introduced in Section 2.2. (We also describe in Section 2.7 a more advanced adequacy theorem, suitable for reasoning about programs such as the case study of Section 2.8.) This specialized adequacy theorem is itself established on top of the general Iris adequacy theorem. When it applies, it is more convenient to use; but in the general case, it is always possible to directly leverage the general adequacy theorem.

We now present our specialized adequacy theorem. We first define a notion of *memory invariant* (Definition 2.1), which corresponds to a predicate over a finite subset of the machine memory. Typically, we will consider predicates of the form: "the value at this specific memory address holds a positive integer" (for instance, the value of the counter of Section 2.2.4). A memory invariant is given by a predicate I over machine memory and a set of addresses D (the "domain" of the invariant); we then require that I is not impacted by changes outside of D.

Definition 2.1 (Memory invariant [5]). *We say that I is a memory invariant over D if I is a predicate over machine memory, D a finite set of addresses, and:*

$$\forall m\, m'.\ (\forall a \in D.\ m(a) = m'(a)) \implies I(m) \Leftrightarrow I(m').$$

We now present the statement of our specialized adequacy theorem; we explain the ingredients in the theorem statement below. Given a memory invariant I over a set D, our adequacy theorem (Theorem 2.1) can be used to show that I indeed holds of the memory at every step of the execution, provided we can show that it holds as an invariant in Iris.

Theorem 2.1 (Adequacy ⑥)**.** *Given a memory invariant I over D, a memory fragment $prog : [b, e) \longrightarrow \mathrm{Word}$, a memory fragment $adv : [b_{adv}, e_{adv}) \longrightarrow \mathrm{Word}$, an initial memory mem, and an initial register file reg, assuming that:*

1. *the initial state of memory mem satisfies:*

$$prog \uplus adv \subseteq mem \qquad D \subseteq \mathrm{dom}(prog) = [b, e)$$

2. *invariant I holds of the initial memory:*

$$I(mem)$$

3. *the adversary region contains no capabilities:*

$$\forall a \in \mathrm{dom}(adv).\ adv(a) \in \mathbb{Z}$$

4. *the initial state of registers reg satisfies:*

$$reg(\mathrm{pc}) = (\textsc{rwx}, b, e, b), \quad reg(\mathrm{r}_0) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}),$$
$$reg(r) \in \mathbb{Z} \ \textit{otherwise}$$

5. *the proof in Iris that the initial configuration is safe given invariant I:*

$$\forall reg,$$
$$\boxed{\exists m, \mathop{\ast}_{(a,w)\in m} a \mapsto w \ast \ulcorner \mathrm{dom}(m) = D \urcorner \ast \ulcorner I(m) \urcorner}$$
$$\vdash \left\{ (\textsc{rwx}, b, e, b);\ \begin{array}{l} \mathrm{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) \ast \\ \mathop{\ast}_{\substack{(r,v)\in reg,\\ r \notin \{\mathrm{pc}, \mathrm{r}_0\}}} r \Mapsto z \ast \ulcorner z \in \mathbb{Z} \urcorner \ast \\ \mathop{\ast}_{(a,z)\in adv} a \mapsto z \ast \ulcorner z \in \mathbb{Z} \urcorner \ast \\ \mathop{\ast}_{\substack{(a,w)\in prog,\\ a \notin D}} a \mapsto w \end{array} \right\} \rightsquigarrow \bullet$$

Then, for any reg', mem', if $(reg, mem) \longrightarrow^{*} (reg', mem')$, then $I(mem')$.

Theorem 2.1 establishes that, starting from an initial machine state (reg, mem), any subsequent machine state (reg', mem') satisfies $I(mem')$. This is subject to a number of conditions, in particular about the initial state of the machine.

First, the initial memory must be provisioned with relevant code and data. This means that the program that we wish to verify (both its code and data) given by memory fragment $prog : [b, e) \longrightarrow \mathrm{Word}$ should be included in the initial memory. Moreover, some additional "adversarial code" given by $adv : [b_{adv}, e_{adv}) \longrightarrow \mathrm{Word}$ should be included in the initial memory. Indeed, we are not only interested in reasoning about the execution of our verified program in isolation, but also its interaction with unverified, possibly adversarial code. The initial memory *mem* should therefore include *prog* and *adv*, in disjoint regions. Furthermore, the domain of the invariant I should be included in the program's region $[b, e)$. The intent is that I specifies an invariant about some private data of the verified program, and thus should not depend on other parts of memory.

Second, as should be expected, the invariant I must hold of the initial memory *mem*.

Third, the adversary memory *adv* is required not to contain any capabilities. This conservatively ensures that *adv* does not contain any "rogue" capability that would give undesired access to the verified program's private state. No further assumption is made about *adv*, which is thus free to contain arbitrary code (i.e. instructions encoded as integers). Furthermore, note that the absence of capabilities in *adv* does not mean that code in *adv* will not be able to access memory at all: at runtime, it will still get access to a capability to its own region through the program counter pc.

Then, the initial register file *reg* should be provided with a RWX capability to the verified program in pc (meaning that it executes first), and a capability to the unverified code in register r_0 (as we have seen in Section 2.2, by convention r_0 holds the pointer to a program's continuation). Other registers are conservatively required not to contain any capabilities.

Finally, one needs to establish at the level of the program logic that the program is safe to run under invariant I. Concretely, one needs to prove a specification for a complete safe execution (of the form $\{P\} \rightsquigarrow \bullet$), given "points-to" resources in the pre-condition that correspond to the initial state of registers and memory. In particular, we get access to points-to resources for the adversary region (along the fact that they contain integers) and points-to resources for the region containing the program to execute.

Note that no resources are given for the domain of I as part of the initial resources for the complete-execution specification. Instead, these resources are part of the logical invariant under which the specification must be established (inside $\boxed{\ldots}$). This corresponds to the intuition that these resources should only be modified in a way that preserves invariant I. This logical invariant therefore specifies that there exists a subset of memory m, which covers the memory region defined by D, such that the invariant holds the corresponding points-to resources and such that $I(m)$ holds, i.e. the memory invariant I holds of this memory subset. (Recall from Section 2.4.1 that $\ulcorner \phi \urcorner$ denotes an Iris proposition that asserts that the mathematical proposition ϕ holds.)

The reader may be surprised to notice that the region containing "adversarial" code has no special status. Indeed, it simply corresponds to a memory region containing (a priori unknown) integers. Nevertheless, remember that we ultimately want our program to be able to pass control to the unknown adversary code by jumping to the capability in r_0, as we have seen our example programs do. This means we need to have a way of reasoning about "what it will do", at least to ensure that it will not break our program's invariants.

$$
\begin{array}{ll}
\mathcal{V}(w) & \begin{cases}
\mathcal{V}(z), \mathcal{V}(\textsc{o}, -, -, -) & \triangleq \text{True} \\
\mathcal{V}(\textsc{e}, b, e, a) & \triangleq \triangleright \square\, \mathcal{E}(\textsc{rx}, b, e, a) \\
\mathcal{V}(\textsc{rw}/\textsc{rwx}, b, e, -) & \triangleq \mathop{\ast}_{a \in [b,e)} \boxed{\exists w,\ a \mapsto w \ast \mathcal{V}(w)} \\
\mathcal{V}(\textsc{ro}/\textsc{rx}, b, e, -) & \triangleq \mathop{\ast}_{a \in [b,e)} \exists P, \begin{array}{l} \boxed{\exists w,\ a \mapsto w \ast P(w)} \ast \\ \triangleright \square\, (\forall w,\ P(w) \mathrel{-\!\!\ast} \mathcal{V}(w)) \end{array}
\end{cases} \\
\mathcal{E}(w) & \triangleq \forall reg,\ \left\{ w; \mathop{\ast}_{(r,v) \in reg, r \neq \mathrm{pc}} r \Mapsto v \ast \mathcal{V}(v) \right\} \rightsquigarrow \bullet
\end{array}
$$

Figure 2.10: Logical relation defining "safe to share" and "safe to execute" values.

In the next section, we show how to reason about whether unknown code can be considered "safe to execute", so that we can pass control to it while preserving previously established invariants.

2.5 Reasoning about Untrusted Code in Cerise

Code running on a capability machine is constrained by the set of capabilities it has access to. This is a crucial idea for reasoning about adversarial code. Whatever code the machine is running, if this code does not have access to a capability for, e.g., writing to a memory region, then it will not be able to modify memory in that region. In other words, one can prove a theorem describing the behavior of arbitrary code depending only on the capabilities it has access to.

One major technical contribution of this work is to formulate and mechanize such a theorem. Specifically, we are concerned with the preservation of invariants established in the program logic. We will thus give a definition of which machine words that are "safe" to share with unknown code. Informally, a word is safe if it cannot be used to break any previously established logical invariants. We will then prove that, as long as some arbitrary code only has access to safe machine words, its execution indeed preserves logical invariants.

Interestingly, we can establish this result while staying within the framework of the Cerise program logic exposed in the previous section. This illustrates the generality of said program logic: verifying specifications for known programs or specifying the behavior of arbitrary code are only two of its possible applications.

2.5.1 Logical Relation

Our formal definition of what makes a machine word *safe*, meaning "safe to share with unknown code", appears in Figure 2.10. It takes the form of a unary logical relation, defining simultaneously the notions of a machine word that is "safe to share" ($\mathcal{V}$) and "safe to execute" ($\mathcal{E}$). The names $\mathcal{V}$ and $\mathcal{E}$ originate from the tradition of logical relations, corresponding respectively to the "value relation" and the "expression relation", although this interpretation is perhaps less obvious in the setting of low-level machine code. We explain the definition in detail below. The intuition is that:

- *A value which is safe to share* only gives transitive access to other values that are safe to share, or code that is safe to execute (in the case of a sentry capability).
- *A value which is safe to execute* allows the machine to run while preserving logical invariants (by definition of $\{\cdot;\cdot\} \rightsquigarrow \bullet$), provided the registers contain safe values.

Technically speaking, this informal definition is circular. Luckily, we can define it properly with the help of the "later" modality $\triangleright$. Iris provides us with a fixed-point operator that only requires recursive occurences to be guarded under a $\triangleright$, and we use that to formally define $\mathcal{V}$ and $\mathcal{E}$. Except for this technical requirement, the reader can in practice ignore the use of $\triangleright$ here.

Let us more closely examine the definition of $\mathcal{V}$, which is defined by case analysis on the shape of the given machine word w. If w is an integer (z), then it is always safe to share, since it cannot be used to access memory. Similarly, opaque capabilities with permission $\textsc{o}$ are always safe as they also do not give access to memory.

A sentry capability $\textsc{e}$ is safe to share if the code it encapsulates is safe to execute. Such a capability can be invoked at any moment and possibly several times: this is expressed through the use of the persistently modality $\Box$. Technically speaking, this means that the property $\mathcal{E}(\textsc{rx}, b, e, a)$ must be established by only relying on persistent resources (typically, logical invariants) that will remain "available" throughout the entire execution.

A read-write capability $\textsc{rw}$ or $\textsc{rwx}$ gives read and write access to the memory region in its range. It is therefore safe as long as the words stored in the corresponding memory region are safe, and continue to be so when the memory gets modified. We thus say that it is safe when we have an invariant for each memory cell in the capability's region, which asserts ownership over the corresponding memory points-to resource, and asserts validity of its contents.

Finally, a capability with permission $\textsc{ro}/\textsc{rx}$ cannot be used by unknown code to modify the memory words in its range. Therefore, these words can obey any property P as long as it entails safety ($\mathcal{V}$). Intuitively, the words in the interval have to be safe to share, because the adversary can read them. But since the adversary cannot modify them, it is possible to guarantee a stronger invariant about them. For instance, $P(w)$ could be the predicate "$w = 42$", describing that a value in the range stays equal to the integer 42.

Notice that this definition of safety does not distinguish between capabilities with permission RO and RX, or RW and RWX. This seems to strangely imply that permissions with the execute bit X have no additional expressive power over permissions without the execute bit. And indeed, *in terms of our model*—which "only" captures the ability to break memory invariants—their expressive power is the same![3] The crux of our main theorem (presented in the next sub-section) is that executing arbitrary code does not produce capabilities with more access to memory than was available before. Thus, being able to execute code within a memory region does not yield additional access to memory compared to what was available by simply reading the memory region (it only leads to additional machine behaviors).

Is this definition of safety trivial? One might wonder whether the definition in Figure 2.10 is trivial, meaning that any machine word w will in fact be considered safe. This is thankfully not the case; let us illustrate concrete cases where a memory word w is *not* considered safe to share with unknown code.

At a high level, $\mathcal{E}$ is not trivial because establishing $\mathcal{E}(w)$ requires proving that a full execution of the machine, starting from w, *preserves logical invariants.* This requirement is not explicit in the definition, but comes from the definition of the Cerise program logic. The definition of $\mathcal{V}(w)$ is also not trivial because, e.g., in the case of an RW capability, it requires the memory points-to $a \mapsto -$ predicate to be part of a specific invariant, $\boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)}$. Since the resource "$a \mapsto -$" is not duplicable, there can be only one resource $a \mapsto -$, which cannot be simultaneously part of two different invariants. Memory cells whose contents evolve according to an invariant more specific (less permissive) than the one above thus cannot be associated with a safe capability (according to $\mathcal{V}$).

What is a concrete example of a capability which is *not* safe? Let us consider a memory cell at address x initialized to 0. Let us assume the following Iris invariant: $\boxed{x \mapsto 0}$. This invariant expresses that x will contain the integer 0 for the rest of the execution. Then, a capability $(\text{RW}, x, x+1, x)$ is not safe to share with an adversary. Indeed, an adversary could use such a capability to write an arbitrary value at address x, thus invalidating the Iris invariant. (However, $(\text{RO}, x, x+1, x)$ would be safe.) A bit more formally speaking, it is not possible to prove $\mathcal{V}(\text{RW}, x, x+1, x)$, because it is not possible to create the invariant $\boxed{\exists w,\ x \mapsto w * \mathcal{V}(w)}$, as the resource for the memory cell x is already part of the invariant $\boxed{x \mapsto 0}$, and cannot be extracted to create a different invariant.

Similarly, one cannot prove $\mathcal{E}$ for a code fragment that writes another value than 0 at address x (after getting access to it through the pc register), because the proof would not be able to guarantee that the Iris invariant related to x is preserved at every step.

[3] Having read-only permission over a region also allows one to simply copy the contents of the region into any other read-execute region and execute them here.

2.5.2 Fundamental Theorem

The Fundamental Theorem of our Logical Relation (Theorem 2.2) (hereafter, FTLR) establishes that any capability that is "safe to share" (in $\mathcal{V}$) is in fact "safe to execute" (in $\mathcal{E}$). In other words, if a capability only gives transitive access to safe capabilities, then it is safe to use it as a program counter capability and execute it: it will not be able to gain extra authority over memory or break any invariants. Importantly, this theorem is independent of the code that the capability points to, even though it is this code that will be executed. Hence the result applies to arbitrary code and we sometimes refer to it as a *universal contract* because of this.

Theorem 2.2 (FTLR ⑦)**.** *Let* $p \in \text{Perm}$, $b, e, a \in \text{Addr}$. *If* $\mathcal{V}(p, b, e, a)$, *then* $\mathcal{E}(p, b, e, a)$.

This is a non-trivial theorem, the proof of which requires checking all the possible cases of the semantics of each instruction of the machine. Indeed, one needs to check that there is no way for some machine instruction to create capabilities with further authority than what was available. This could, for example, happen if some runtime checks were missing, making it possible to create a capability $(\textsc{rw}, b, e + 1, a)$ from a capability $(\textsc{rw}, b, e, a)$. One can imagine how this would break expected security guarantees, and reveal a design or implementation bug of the machine. Therefore, another informal interpretation of the fundamental theorem is that it expresses that the capability machine "works well" or that it is *capability safe*.

The fundamental theorem provides a universal security property satisfied by unknown code, and gives us a way of verifying the correctness of known code that includes calls to possibly malicious code. To sum up, our logical relation characterizes the interface between a piece of verified code which wishes to preserve invariants on some internal state, and "external" arbitrary code whose accessible, safe capabilities have been sufficiently restricted.

It is important to note that the distinction between "known" and "adversary" code only exists at the logical level: there is no such distinction at runtime. We can have two components that have been verified separately, and that do not mutually trust each other. In this case, from the point of view of each component, the other component is considered as being the adversary.

Rules for program verification. From the general statement of the FTLR, we can derive two corollaries, which can be used to instantiate our adequacy theorem (Theorem 2.1) with a program that passes control to an unknown adversarial code region.

Corollary 2.1 (Unknown integers are safe ⑧). *For $m : [b, e) \longrightarrow$ Word,*

$$\mathop{\Huge\ast}_{(a,z)\in m} a \mapsto z * \ulcorner z \in \mathbb{Z} \urcorner \mathrel{-\!\!*} \mathcal{V}(p, b, e, a)$$

Corollary 2.1 can be used to trade ownership over a memory region of integers to the knowledge that a capability over this region is safe.[4] Since integers can encode program instructions, we can typically use this rule to reason about a memory region containing an (unknown) program. The rule follows directly from the definition of $\mathcal{V}$ for values of p different from E; when $p = \text{E}$, an additional application of the FTLR (Theorem 2.2) is required.

Notice that the pre-condition of the rule matches the resources that one gets in the Adequacy theorem (Theorem 2.1) for the adversary region. When using the Adequacy theorem, we will thus be able to conclude that capabilities pointing to the adversary region are safe.

Corollary 2.2 (Jump to a safe word ⑨).

$$\begin{array}{l}\mathcal{V}(w) \mathrel{-\!\!*} \\ \triangleright \forall reg.\ \left\{\text{updatePcPerm}(w); \mathop{\ast}_{(r,v)\in reg, r\neq \text{pc}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet\end{array}$$

Corollary 2.2 gives us a specification for the execution of the machine after a jump to an unknown word w, assuming that w is safe. Recall that updatePcPerm(w) corresponds to the value of the program counter after jumping to w (see the machine semantics in Figure 2.6). The full execution specification in the conclusion of the rule requires that the machine registers contain safe values: indeed, we must only share safe words with unknown code.

An important application of Corollary 2.2 is to reason about the last instruction of a program encapsulated in a sentry (E) capability, where it “returns” and passes control to its caller by calling `jmp` on the (unknown but safe) return pointer held in r_0. In this scenario, the return pointer provided by the caller is unknown but safe, so Corollary 2.2 gives us a specification for the continuation of the program.

Additionally, Corollary 2.2 is typically used in combination with Corollary 2.1 when instantiating the Adequacy theorem. Indeed, in order to prove the complete safe execution specification required by the theorem, one typically needs to justify that one can `jmp` and pass control to an adversary region, given the resources granted by the Adequacy theorem.

[4] We simplify the presentation here a bit and omit a view shift from the statement of Corollary 2.1. See the Coq development for the exact formal statement ⑧.

2.5.3 Proving the fundamental theorem

To give a more in-depth perspective of the ideas behind the Fundamental Theorem, we detail in this sub-section one of the interesting cases of its proof. This sub-section can be safely skipped on a first read.

Proof. (FTLR) We begin by unfolding the definition of $\mathcal{E}$.

$$\forall reg.\ \left\{(p, b, e, a);\ \mathop{\ast}_{(r,v) \in reg, r \neq \mathrm{pc}} r \Mapsto v \ast \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

We proceed by Löb induction. The Löb rule is a powerful reasoning principle, which Cerise inherits from Iris, and which states that (in any context Q), if from $\triangleright P$ we can derive P, then we can also derive P without any assumptions.

$$\textsc{Löb}\qquad \frac{Q \land \triangleright P \vdash P}{Q \vdash P}$$

The idea of the rule is that "after we do some work", we will be able to remove the $\triangleright$ modality from the assumption, and reach the conclusion. In our case, this means reasoning about one step of execution, for one instruction. Intuitively, if we show that our property holds for the execution of one arbitrary instruction, then it must hold for a sequence of many instructions.

We thus let:

$$\begin{aligned} \mathrm{IH} \ \triangleq\ & \forall p, b, e, a.\ \mathcal{V}(p, b, e, a) \mathrel{-\!\!\ast} \\ & \quad \forall reg.\ \left\{(p, b, e, a);\ \mathop{\ast}_{(r,v) \in reg, r \neq \mathrm{pc}} r \Mapsto v \ast \mathcal{V}(v)\right\} \rightsquigarrow \bullet \end{aligned}$$

and assume $\triangleright$ IH; we then wish to show IH.

First, we consider the case where (p, b, e, a) is not a valid program counter (for instance, if it contains a non-executable capability, or an integer). Then the machine configuration will step into a Failed configuration. In that case, any full execution specification ($\{\cdot; \cdot\} \rightsquigarrow \bullet$) trivially holds, and we are done.

In the case where (p, b, e, a) is a valid program counter, we will have to execute the next instruction of the program, pointed to by a. For (p, b, e, a) to be a valid program counter, the following needs to hold:

$$p \in \{\textsc{rx}, \textsc{rwx}\} \tag{2.1}$$

$$b \leq a < e \tag{2.2}$$

From (2.1), we can infer that $\mathcal{V}(p, b, e, a)$ will unfold to (at least) the following:

$$\mathop{\ast}_{a \in [b,e)} \exists P,\ \boxed{\exists w,\ a \mapsto w \ast P(w)} \ast \triangleright \square\, \forall w,\ P(w) \mathrel{-\!\!\ast} \mathcal{V}(w)$$

Since we know that a is an address in the range $[b, e)$ (2.2), we can in particular infer that there exists a predicate P such that $\triangleright \square\, \forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w)$, for which the following invariant holds:

$$\boxed{\exists w,\ a \mapsto w * P(w)} \tag{2.3}$$

Ownership over $a \mapsto w$ is in fact required in order to apply any rule of the program logic (we need to be able to access memory for the instruction pointed to by pc). We will therefore first open the invariant (2.3) to get access to that resource.

Recall the invariant opening rule Inv (Section 2.4.2). According to that rule, we can get access to the resources held inside the invariant *now*, as long as we give them back *after one execution step*. Since we wish here to reason about the execution of a single instruction, this is a perfectly good deal.

Once the invariant has been opened, the following propositions are added to our assumptions, for some word w (technically speaking, the Iris context also tracks the fact that these facts come from an invariant and must be given back next, but we choose to hide these details):[5]

$$a \mapsto w \tag{2.4}$$

$$\triangleright\ P(w) \tag{2.5}$$

Because pc points to a, and address a contains the word w, w should correspond to the (encoding of the) instruction to execute now. We thus reason by case analysis on $\mathrm{decode}(w)$.

This leads to as many cases as there are instructions in the machine. We will now detail a sub-case for the `load` instruction, which is one of the interesting cases. Many of the other cases are similar in nature.

Case: $\mathrm{decode}(w) = \texttt{load}\ r_{dst}\ r_{src}$.

We consider here the case where r_{dst} and r_{src} are two different registers, both different from pc. We also only consider the case where r_{src} contains a capability, which we are permitted to load from. In other words, our goal is as follows:[6]

$$\begin{array}{l} \triangleright \mathrm{IH} * a \mapsto w * \triangleright P(w) \\ \vdash \left\{ (p, b, e, a);\ \begin{array}{l} \mathop{\Huge *}_{(r,v) \in reg, r \neq \mathrm{pc}, r_{dst}, r_{src}} r \Mapsto v * \mathcal{V}(v) \\ * \ r_{dst} \Mapsto w' * \mathcal{V}(w') \\ * \ r_{src} \Mapsto (p', b', e', a') * \mathcal{V}(p', b', e', a') \end{array} \right\} \rightsquigarrow \bullet \end{array}$$

[5] Notice that we directly get $a \mapsto w$ rather than $\triangleright\, a \mapsto w$, due to the fact that memory points-to are timeless.

[6] We again omit details involving masks and update modalities, and refer to the Coq formalization for the full details.

As stated, we assume that (p', b', e', a') permits us to load from a'. We can thus infer the following two properties:

$$p' \in \{\textsc{ro}, \textsc{rx}, \textsc{rw}, \textsc{rwx}\} \tag{2.6}$$

$$b' \leq a' < e' \tag{2.7}$$

Just like before, we can from $\mathcal{V}(p', b', e', a')$ conclude that the following invariant holds, where P' is a predicate such that $\triangleright \Box\, \forall w,\ P'(w) \mathrel{-\!\!*} \mathcal{V}(w)$:

$$\boxed{\exists w,\ a' \mapsto w * P'(w)} \tag{2.8}$$

We consider the (more interesting) case where $a \neq a'$. We can thus open the invariant (since it has not been opened already), meaning that we have for some word w_{src} the following (again, plus some invariant-tracking resources not shown here):

$$a' \mapsto w_{src} \tag{2.9}$$

$$\triangleright\ P'(w_{src}) \tag{2.10}$$

With these assumptions, we now have all the necessary resources to take a step in the program logic, using the rule for the `load` instruction (Figure 2.9). A feature of single-instruction rules of our program logic is that they include a built-in $\triangleright$ modality. In other words, after applying a single-instruction rule, we are "one execution step later", and we can remove one occurrence of $\triangleright$ for each assumption of our context. In particular, this means that we can turn $\triangleright$ IH into IH, and similarly for $P(w)$ and $P'(w_{src})$. We now have to show:

$$\begin{array}{l} \mathrm{IH} * a \mapsto w * P(w) * a' \mapsto w_{src} * P'(w_{src}) \\ \vdash \left\{ (p, b, e, a+1);\ \begin{array}{l} \mathop{\Large\ast}_{(r,v) \in reg, r \neq \mathrm{pc}, r_{dst}, r_{src}} r \Mapsto v * \mathcal{V}(v) \\ * \ r_{dst} \Mapsto w_{src} \\ * \ r_{src} \Mapsto (p', b', e', a') * \mathcal{V}(p', b', e', a') \end{array} \right\} \rightsquigarrow \bullet \end{array}$$

We now have direct access to IH (our initial goal) as an assumption, so the proof is nearly done. Before we can invoke IH and conclude the goal, we must do two things: (a) close all the open invariants (as required by the invariant opening rule), and (b) show that the contents of all registers satisfies $\mathcal{V}$ (required by the definition of IH). (We actually need to show (b) *before* addressing (a), as we will make use of resources from the open invariants.)

Addressing (b), we already know that the contents of registers satisfy $\mathcal{V}$ for all registers except for r_{dst}—the only register whose contents were changed by the instruction. We must thus prove $\mathcal{V}(w_{src})$. Luckily, w_{src} is not a completely arbitrary word: it was accessible from available memory, so it must be safe as well. More precisely, from the invariant about a' (previously opened), we know that $P'(w_{src})$ holds, and furthermore we know that:

$$\Box\, \forall w, P'(w) \mathrel{-\!\!*} \mathcal{V}(w)$$

Owing to the fact that $\mathcal{V}(\cdot)$ is persistent, we can shave off the $\square$ modality, and conclude that $\mathcal{V}(w_{src})$ holds, concluding the proof of (b).

Finally, addressing (a) is straightforward, since we did not change the contents of memory at either address a or a'. We can therefore close the invariants again, by giving up the same resources as we initially got from opening them, concluding the proof of (a) and thus the case of the proof for `load`.

In the proof sketch above, we followed one specific subcase of the proof for the `load` instruction. In the complete proof, we must go through all the possible cases of the semantics for the instruction. In some cases, the machine fails which terminates the proof easily (for instance, if the capability in r_{src} does not in fact allow reading memory, or if r_{src} does not in fact contain a capability). In some other cases, the machine does not fail, and the proof is similar to the case highlighted here but slightly different (for instance when r_{dst} and r_{src} are the same register).

The proofs for the other instructions of the machine follow a similar pattern. In particular, in the `store` case, the register state is not modified except for the pc register, but memory is modified. As such, closing the invariants is not as easy since we need to establish that the stored word is at least safe. This is established by using the fact that we assumed that the register only contains safe words. The case of the `restrict`, `subseg` and `lea` instructions require showing that a capability with smaller authority remains in the value relation $\mathcal{V}$, and the `jmp`, `jnz` and `mov` cases show that pc (or other registers) can be updated with arbitrary safe words. The other remaining cases are rather trivial, as they all only change a register state to an integer, which is always safe. $\square$

2.6 Reasoning with capabilities: two examples

In this section, we return to the motivational examples introduced in Section 2.2, and show how to prove that they enforce the desired properties, using Cerise's reasoning tools, laid out in the previous sections.

2.6.1 Sharing a sub-buffer with an unknown adversary

Let us recall (on the right) the code for our buffer-sharing program, previously introduced in Figure 2.3. The labels `code`, `data`, `secret` and `end` denote addresses in memory. We wish to prove formally that the program can share the data between addresses `data` and `secret`

```
code: mov r1 PC
      lea r1 [data-code]
      subseg r1 [data] [data+3]
      jmp r0
data:   'H', 'i', 0, ; public
secret: 42           ; secret
end:
```

(excluded), while protecting the integrity of
the data at address `secret`.

Using the reasoning rules from our program logic, we can first prove a specification for the program, specifying its behavior from its first instruction up until the final `jmp`. The corresponding specification is as follows, where code_instrs is the list of integers corresponding to the encoded instructions of the program, i.e., code_instrs = map encodeInstr [`mov` r_1 pc;...;`jmp` r_0].

Lemma 2.1 (Program specification ⑩).

$$\left\{ (\textit{RWX}, \text{code}, \text{end}, \text{code});\ \begin{array}{l} \mathsf{r}_0 \Mapsto w_{adv} * \mathsf{r}_1 \Mapsto - * \\ [\text{code}, \text{data}) \mapsto \text{code\_instrs} \end{array} \right\} \rightsquigarrow$$
$$\left\{ \text{updatePcPerm}(w_{adv});\ \begin{array}{l} \mathsf{r}_0 \Mapsto w_{adv} * \mathsf{r}_1 \Mapsto (\textit{RWX}, \text{data}, \text{secret}, \text{data}) * \\ [\text{code}, \text{data}) \mapsto \text{code\_instrs} \end{array} \right\}$$

One can read from the specification that executing the program will store in r_1 an RWX capability to the memory segment between addresses `data` and `secret` (our "buffer"), and pass control to the word w_{adv} found in register r_0.

Proving this specification is easy: it is enough to successively apply the program logic rule of each individual instruction found in the program.

This specification shows that the program ultimately jumps to the word initially passed in register r_0, but does not describe what happens after, in the case where this word points to a region containing unknown code. For this, we use the reasoning principles from Section 2.5.2 (built on top of the Fundamental Theorem), and derive a specification for a complete execution of the machine, see Lemma 2.2 below. The lemma specifies that, starting by executing our program, and given that r_0 contains a capability to a region containing unknown integers, then the machine is safe to run. Notice that we do not assume a points-to resource for the secret data: instead, this points-to will be part of an invariant—specifying that it contains the same secret value at every step—and we do not need to access that here.

Lemma 2.2 (Full execution specification ⑪).

$$\left\{ (\textit{RWX}, \text{code}, \text{end}, \text{code});\ \begin{array}{l} \mathsf{r}_0 \Mapsto (\textit{RWX}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathsf{r}_1 \Mapsto - * \\ \mathop{\Huge *}_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0,\mathsf{r}_1\}}} r \Mapsto z * \ulcorner z \in \mathbb{Z} \urcorner * \\ [\text{code}, \text{data}) \mapsto \text{code\_instrs} * \\ [\text{data}, \text{secret}) \mapsto [\texttt{'H'};\texttt{'i'};0] * \\ \mathop{\Huge *}_{(a,z)\in adv} a \mapsto z * \ulcorner z \in \mathbb{Z} \urcorner \end{array} \right\} \rightsquigarrow \bullet$$

Proof. By Lemma 2.1, the frame rule FragFrame and the sequence rule SeqFull, it suffices to show the following goal, which corresponds to a specification about the execution of the machine *after* the execution of the verified code:

$$\textit{Goal:}\qquad \left\{ (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv});\quad \begin{array}{l} \mathsf{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathsf{r}_1 \Mapsto (\textsc{rwx}, \mathrm{data}, \mathrm{secret}, \mathrm{data}) * \\ \bigast_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0,\mathsf{r}_1\}}} r \Mapsto z * \ulcorner z \in \mathbb{Z} \urcorner * \\ \bigast_{(a,z)\in adv}\, a \mapsto z * \ulcorner z \in \mathbb{Z} \urcorner * \\ [\mathrm{code}, \mathrm{data}) \mapsto \mathrm{code\_instrs} * \\ [\mathrm{data}, \mathrm{secret}) \mapsto [\texttt{'H'};\texttt{'i'};\texttt{0}] \end{array} \right\} \rightsquigarrow \bullet$$

We now rely on the reasoning rules derived from the Fundamental Theorem (Section 2.5.2). First, from the fact that the adversary region *adv* does not contain capabilities, we get using Corollary 2.1 that any capability on that region is safe, in particular we have $\mathcal{V}(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$. Then, from Corollary 2.2 we get a specification for the execution of the machine starting from $\mathcal{V}(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$ (recall that updatePcPerm is the identity on non-$\textsc{e}$ capabilities):

$$\textit{Fact:}\qquad \forall reg.\ \left\{(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv});\ \bigast_{(r,v)\in reg, r\neq \mathsf{pc}}\, r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

From this fact, we can prove our goal provided that we show that the contents of all machine registers satisfy $\mathcal{V}$. For registers other than r_0 and r_1, this holds by definition of $\mathcal{V}$, as we know they only contain integers. Register r_0 contains a capability to the adversary region, which we have already proved to be safe using Corollary 2.1. Finally, register r_1 contains the capability pointing to the public buffer. We can again leverage Corollary 2.1 to obtain $\mathcal{V}(\textsc{rwx}, \mathrm{data}, \mathrm{secret}, \mathrm{data})$ from the memory points-to for the buffer ($[\mathrm{data}, \mathrm{secret}) \mapsto [\texttt{'H'};\texttt{' i'};\texttt{0}]$), thus concluding the proof. □

Finally, from Lemma 2.2, established in the program logic, we wish to obtain a final result in terms of the operational semantics of the machine. The toplevel end-to-end theorem that we obtain is shown in Theorem 2.3. We consider a machine whose memory is initially loaded with our program and unknown adversarial code, and that starts by executing our verified code. The theorem establishes that the adversary will not be able to tamper with the value held at address `secret`: at every step of the execution, it will be unchanged and equal to 42.

Theorem 2.3 (End-to-end theorem: integrity of the secret data is preserved ⑫). *Starting from an initial state of the machine* (reg, mem) *where:*

- *prog* $\uplus$ *adv* $\subseteq$ *mem, for adv* : $[b_{adv}, e_{adv}) \to$ Word *and prog* : $[\mathrm{code}, \mathrm{end}) \to$ Word
- *the contents of prog correspond to the encoded instructions and program data;*
- *the adversary memory contains no capabilities:* $\forall a. adv(a) \in \mathbb{Z}$*;*

- *the initial state of registers satisfies:*
  $reg(\mathrm{pc}) = (\textsc{rwx}, \mathrm{code}, \mathrm{end}, \mathrm{code})$,
  $reg(\mathsf{r}_0) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$,
  $reg(r) \in \mathbb{Z}$ *otherwise;*

Then, for any reg'*,* mem'*, if* $(reg, mem) \longrightarrow^* (reg', mem')$*, then* $mem'(\mathrm{secret}) = 42$.

Proof. We first invoke Theorem 2.1, choosing the memory invariant I and its domain D to be the invariant I_{buf} and domain D_{buf} defined below, asserting that the value at address `secret` is equal to 42:

$$\begin{aligned} &I_{buf} \triangleq \lambda m.\ m(\mathrm{secret}) = 42 \\ &\text{and} \ \ D_{buf} = \{\mathrm{secret}\}. \end{aligned}$$

Most side-conditions of the adequacy theorem can be easily discharged. What remains is the following specification in Iris:

$$\begin{array}{l} \qquad\qquad\qquad \boxed{\exists m, \mathop{\ast}_{(a,w)\in m} a \mapsto w \ast \ulcorner \mathrm{dom}(m) = D_{buf} \urcorner \ast \ulcorner I_{buf}(m) \urcorner} \\ \textit{Goal:} \quad \vdash \left\{ (\textsc{rwx}, \mathrm{code}, \mathrm{end}, \mathrm{code});\ \begin{array}{l} \mathsf{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) \ast \\ \mathop{\ast}_{\substack{(r,v)\in reg, \\ r \notin \{\mathrm{pc}, \mathsf{r}_0\}}} r \Mapsto z \ast \ulcorner z \in \mathbb{Z} \urcorner \ast \\ \mathop{\ast}_{(a,z)\in adv} a \mapsto z \ast \ulcorner z \in \mathbb{Z} \urcorner \ast \\ \mathop{\ast}_{\substack{(a,w)\in prog, \\ a \notin D_{buf}}} a \mapsto w \end{array} \right\} \rightsquigarrow \bullet \end{array}$$

We can simplify this goal by unfolding the definition of I_{buf}, D_{buf}, *prog* and massaging the goal to extract relevant points-to resources. The goal then becomes:

$$\begin{array}{l} \qquad\qquad\qquad \boxed{\mathrm{secret} \mapsto 42} \\ \textit{Goal:} \quad \vdash \left\{ (\textsc{rwx}, \mathrm{code}, \mathrm{end}, \mathrm{code});\ \begin{array}{l} \mathsf{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) \ast \\ \mathsf{r}_1 \Mapsto - \ast \\ \mathop{\ast}_{\substack{(r,v)\in reg, \\ r \notin \{\mathrm{pc}, \mathsf{r}_0, \mathsf{r}_1\}}} r \Mapsto z \ast \ulcorner z \in \mathbb{Z} \urcorner \ast \\ [\mathrm{code}, \mathrm{data}) \mapsto \mathrm{code\_instrs} \ast \\ [\mathrm{data}, \mathrm{secret}) \mapsto [\texttt{'H'}; \texttt{'i'}; 0] \ast \\ \mathop{\ast}_{(a,z)\in adv} a \mapsto z \ast \ulcorner z \in \mathbb{Z} \urcorner \end{array} \right\} \rightsquigarrow \bullet \end{array}$$

Note how the points-to resource for the `secret` address is held as part of the invariant, asserting that it contains the value 42 at each step. This simplified goal now follows from the full execution specification established earlier in Lemma 2.2 by applying the rule FullFrame, which concludes the proof. □

2.6.2 Creating a closure around local state

Let us now come back to the example introduced in Section 2.2.4, whose code is reproduced below. In this example, the control flow is somewhat more involved, as we have two separate pieces of known code that run at different times. The initialization code between init and code runs first, and creates a sentry capability before passing control to the unknown code. The code and data located between code and end are encapsulated in the sentry capability created by the initialization code. Because the sentry capability is exposed to the unknown code, the code it encapsulates may be invoked several times, incrementing the value of the counter each time.

We wish to prove formally that the value of the counter is correctly encapsulated. We prove that it remains non-negative at every step: starting from zero, it can only get incremented by the code routine encapsulated in the sentry capability.

```
init:
  mov r1 PC
  lea r1 [data-init]
  mov r2 r1
  lea r2 1
  store r1 r2
  lea r1 [code-data]
  subseg r1 [code] [end]
  restrict r1 E
  mov r2 0
  jmp r0
```

```
code:
  mov r1 PC
  lea r1 [data-code]
  load r1 r1
  load r2 r1
  add r2 r2 1
  store r1 r2
  mov r1 0
  jmp r0
data:
  ; will be:
  ; (RWX, init, end, data+1)
  0xFFFF,
  0 ; counter value
end:
```

Using the rules of our program logic, we can first prove a specification for the initialization code, shown in Lemma 2.3. This specification describes the behavior of the code between init and code, where init_instrs denote the corresponding list of encoded instructions.

Lemma 2.3 (Specification for the initialization code ⑬)**.**

$$\left\{ (\textit{RWX}, \mathrm{init}, \mathrm{end}, \mathrm{init});\ \begin{array}{l} \mathrm{r}_0 \Mapsto w_{adv} * \mathrm{r}_1 \Mapsto - * \mathrm{r}_2 \Mapsto - * \\ \mathrm{data} \mapsto - * [\mathrm{init}, \mathrm{code}) \mapsto \mathrm{init\_instrs} \end{array} \right\} \rightsquigarrow$$

$$\left\{ \mathrm{updatePcPerm}(w_{adv});\ \begin{array}{l} \mathrm{r}_0 \Mapsto w_{adv} * \mathrm{r}_1 \Mapsto (\textit{E}, \mathrm{code}, \mathrm{end}, \mathrm{code}) * \mathrm{r}_2 \Mapsto 0 * \\ \mathrm{data} \mapsto (\textit{RWX}, \mathrm{init}, \mathrm{end}, \mathrm{data} + 1) * \\ [\mathrm{init}, \mathrm{code}) \mapsto \mathrm{init\_instrs} \end{array} \right\}$$

From this specification, one can read that running the initialization code will store in register r_1 a sentry capability to $[\mathrm{code}, \mathrm{end})$, and write at address data an RWX capability pointing to the location holding the counter value. The initialization code then passes control to the unknown word w_{adv} stored in r_0.

We can also use the program logic rules to prove a specification for the code routine in $[\mathrm{code}, \mathrm{data})$ which increments the counter, and which will run each time the sentry capability is invoked. The specification appears in Lemma 2.4, where code_instrs refers to the list of encoded instructions for the routine.

Lemma 2.4 (Specification for the increment routine ⑭).

$$
\begin{aligned}
&\boxed{[\mathrm{code}, \mathrm{data}) \mapsto \mathrm{code\_instrs}}\,,\\
&\boxed{\mathrm{data} \mapsto (\textit{RWX}, \mathrm{init}, \mathrm{end}, \mathrm{data}+1)}\,,\ \boxed{\exists n.\, (\mathrm{data}+1) \mapsto n * \ulcorner n \geq 0 \urcorner}\\
&\vdash \{(\textit{RX}, \mathrm{code}, \mathrm{end}, \mathrm{code}); \mathsf{r}_0 \Mapsto w_{cont} * \mathsf{r}_1 \Mapsto - * \mathsf{r}_2 \Mapsto -\} \rightsquigarrow\\
&\quad \{\mathrm{updatePcPerm}(w_{cont}); \exists n.\, \mathsf{r}_0 \Mapsto w_{cont} * \mathsf{r}_1 \Mapsto 0 * \mathsf{r}_2 \Mapsto n\}
\end{aligned}
$$

This specification assumes a number of Iris invariants, describing the contents of the $[\mathrm{code}, \mathrm{end})$ memory region. Indeed, because the increment routine is invoked by unknown code, it cannot make many assumptions about the state of the machine. The only thing that it can assume is that previously established invariants still hold (because, by definition, capability-safe unknown code has to preserve invariants).

The specification thus assumes, as invariants: 1) that the region $[\mathrm{code}, \mathrm{data})$ contains the code of the routine; 2) that data contains the RWX capability to the counter value previously stored there by the initialization code, and finally 3) that the counter value (at address $\mathrm{data} + 1$) is a non-negative integer.

The specification asserts that the routine can run, starting with pc containing an RX capability to the $[\mathrm{code}, \mathrm{end})$ region, while preserving the invariants. (In particular, this means that incrementing the counter indeed preserves the fact that it is a non-negative integer.) Recall that the RX permission in pc corresponds to what one gets after jumping to a sentry capability.

Finally, we prove as before a specification proving safety of complete executions, starting from the initialization code, then followed by the execution of unknown code, including its possible invocations of the sentry capability. This specification appears below in Lemma 2.5.

Lemma 2.5 (Full execution specification ⑮).

$$\boxed{\exists n.\, (\mathrm{data}+1) \mapsto n * \ulcorner n \geq 0 \urcorner}$$

$$\vdash \left\{ (\textsc{rwx}, \mathrm{init}, \mathrm{end}, \mathrm{init});\ \begin{array}{l} \mathrm{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \mathrm{r}_1 \Mapsto - * \\ \mathrm{r}_2 \Mapsto - * \mathop{\Large\ast}\limits_{\substack{(r,v)\in reg,\\ r\notin\{\mathrm{pc},\mathrm{r}_0..\mathrm{r}_2\}}} r \Mapsto z * \ulcorner z \in \mathbb{Z} \urcorner * \\ [\mathrm{init}, \mathrm{code}) \mapsto \mathrm{init\_instrs} * \\ [\mathrm{code}, \mathrm{data}) \mapsto \mathrm{code\_instrs} * \mathrm{data} \mapsto - * \\ \mathop{\Large\ast}_{(a,z)\in adv} a \mapsto z * \ulcorner z \in \mathbb{Z} \urcorner \end{array} \right\} \rightsquigarrow \bullet$$

Proof. By using Lemma 2.3 (the specification for the initialization code), the frame rule FragFrame and sequence rule SeqFull, it is enough to show the following goal, which specifies the execution of the machine after the initialization code has run:

$$\boxed{\exists n.\, (\mathrm{data}+1) \mapsto n * \ulcorner n \geq 0 \urcorner}$$

$$\textit{Goal:} \quad \vdash \left\{ (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv});\ \begin{array}{l} \mathrm{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathrm{r}_1 \Mapsto (\textsc{e}, \mathrm{code}, \mathrm{end}, \mathrm{code}) * \mathrm{r}_2 \Mapsto 0 * \\ \mathop{\Large\ast}\limits_{\substack{(r,v)\in reg,\\ r\notin\{\mathrm{pc},\mathrm{r}_0..\mathrm{r}_2\}}} r \Mapsto z * \ulcorner z \in \mathbb{Z} \urcorner * \\ [\mathrm{init}, \mathrm{code}) \mapsto \mathrm{init\_instrs} * \\ [\mathrm{code}, \mathrm{data}) \mapsto \mathrm{code\_instrs} * \\ \mathrm{data} \mapsto (\textsc{rwx}, \mathrm{init}, \mathrm{end}, \mathrm{data}+1) * \\ \mathop{\Large\ast}_{(a,z)\in adv} a \mapsto z * \ulcorner z \in \mathbb{Z} \urcorner \end{array} \right\} \rightsquigarrow \bullet$$

We then allocate two new invariants, one containing the code of the sentry capability, the other the points-to resource at address data.

$$\boxed{[\mathrm{code}, \mathrm{data}) \mapsto \mathrm{code\_instrs}},\ \boxed{\mathrm{data} \mapsto (\textsc{rwx}, \mathrm{init}, \mathrm{end}, \mathrm{data}+1)},$$
$$\boxed{\exists n.\, (\mathrm{data}+1) \mapsto n * \ulcorner n \geq 0 \urcorner}$$

$$\textit{Goal:} \quad \vdash \left\{ (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv});\ \begin{array}{l} \mathrm{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathrm{r}_1 \Mapsto (\textsc{e}, \mathrm{code}, \mathrm{end}, \mathrm{code}) * \mathrm{r}_2 \Mapsto 0 * \\ \mathop{\Large\ast}\limits_{\substack{(r,v)\in reg,\\ r\notin\{\mathrm{pc},\mathrm{r}_0..\mathrm{r}_2\}}} r \Mapsto z * \ulcorner z \in \mathbb{Z} \urcorner * \\ [\mathrm{init}, \mathrm{code}) \mapsto \mathrm{init\_instrs} * \\ \mathop{\Large\ast}_{(a,z)\in adv} a \mapsto z * \ulcorner z \in \mathbb{Z} \urcorner \end{array} \right\} \rightsquigarrow \bullet$$

From Corollary 2.1 and the fact that the adversary region *adv* does not contain capabilities, we get that any capability on that region is safe, and therefore that $\mathcal{V}(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$ holds. From Corollary 2.2, we get that a full execution starting from $(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$ is safe:

$$\textit{Fact:} \qquad \forall reg.\, \left\{ (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}); \mathop{\Large\ast}_{(r,v)\in reg, r\neq \mathrm{pc}} r \Mapsto v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet$$

In combination with rule FullFrame, this fact allows us to conclude the proof, *provided we can prove safety of values stored in all registers.* We have already proved the capability in r_0 to be safe. Registers r_2 to r_{31} contain integers, so they are safe by definition of $\mathcal{V}$. Safety of the sentry capability created by the initialization code and stored in r_1 remains to be proven.

$$\textit{Goal:}\qquad \begin{array}{l} \boxed{[\text{code}, \text{data}) \mapsto \text{code\_instrs}}\,,\ \boxed{\text{data} \mapsto (\textsc{rwx}, \text{init}, \text{end}, \text{data}+1)}\,, \\ \boxed{\exists n.\, (\text{data}+1) \mapsto n * \ulcorner n \geq 0 \urcorner} \\ \vdash \mathcal{V}(\textsc{e}, \text{code}, \text{end}, \text{code}) \end{array}$$

By definition of $\mathcal{V}$ and $\mathcal{E}$, this goals unfolds to the following:

$$\textit{Goal:}\qquad \begin{array}{l} \boxed{[\text{code}, \text{data}) \mapsto \text{code\_instrs}}\,,\ \boxed{\text{data} \mapsto (\textsc{rwx}, \text{init}, \text{end}, \text{data}+1)}\,, \\ \boxed{\exists n.\, (\text{data}+1) \mapsto n * \ulcorner n \geq 0 \urcorner} \\ \vdash \triangleright \Box\, \forall reg,\ \left\{(\textsc{rx}, \text{code}, \text{end}, \text{code});\ \mathop{\Large *}_{(r,v) \in reg, r \neq \mathrm{pc}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet \end{array}$$

For technical reasons, we can shave off the later modality ($\triangleright$) in front of the goal (we refer to the Coq formalization for more details). The persistent modality ($\Box$) is more interesting: it expresses the fact that safety of the callback should only depend on persistent assumptions. This corresponds to the fact that the callback may be invoked several times, in future execution states and because of this it cannot rely on non-persistent assumptions that only hold at the callback's creation time. Fortunately, invariants are persistent, so they remain available for proving the callback's safety.

Then, let us name w_0 the contents of register r_0: we get to assume $\mathcal{V}(w_0)$ (as for the contents of other registers). By using Lemma 2.4 (the specification for the increment routine) with rules FragFrame and SeqFull, it is enough to prove the following goal, which asserts safety of the execution *after* passing control back to unknown code by jumping to w_0:

$$\textit{Goal:}\qquad \vdash \left\{ \text{updatePcPerm}(w_0);\ \begin{array}{l} \exists n.\, \mathrm{r}_0 \Mapsto w_0 * \mathrm{r}_1 \Mapsto 0 * \mathrm{r}_2 \Mapsto n\, * \\ \mathop{\Large *}_{(r,v) \in reg, r \notin \{\mathrm{pc}, \mathrm{r}_0, \mathrm{r}_1, \mathrm{r}_2\}} r \Mapsto v * \mathcal{V}(v) \end{array} \right\} \rightsquigarrow \bullet$$

Informally, the increment routine returns to the unknown code by passing control to some unknown word provided in r_0: it is safe to do so, since such word can be assumed to be itself safe. Formally speaking, we know $\mathcal{V}(w_0)$, so we apply Corollary 2.2 which concludes the proof. □

Similarly to the previous example, we derive from Lemma 2.5 a toplevel theorem which only refers to the operational semantics of the machine, shown below in Theorem 2.4. We consider a machine initially loaded with our program and unknown adversarial code. The theorem establishes that the value of the counter is properly encapsulated: at every step of the execution, it will be a non-negative integer.

Theorem 2.4 (End-to-end theorem: integrity of the counter value is preserved (16)). *Starting from an initial state of the machine* (reg, mem) *where:*

- $prog \uplus adv \subseteq mem$, *for* $adv : [b_{adv}, e_{adv}) \rightarrow \mathsf{Word}$ *and* $prog : [\mathsf{init}, \mathsf{end}) \rightarrow \mathsf{Word}$
- *the contents of prog correspond to the encoded instructions and program data;*
- *the adversary memory contains no capabilities:* $\forall a. adv(a) \in \mathbb{Z}$*;*
- *the initial state of registers satisfies:*
  $reg(\mathsf{pc}) = (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{init})$,
  $reg(\mathsf{r}_0) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$,
  $reg(r) \in \mathbb{Z}$ *otherwise;*

Then, for any reg', mem', *if* $(reg, mem) \longrightarrow^* (reg', mem')$, *then* $mem'(\mathsf{data} + 1) \geq 0$.

Proof. We invoke Theorem 2.1, with invariant and domain I_{cnt} and D_{cnt} defined as follows:

$$I_{cnt} \triangleq \lambda m.\, m(\mathsf{data} + 1) \geq 0$$
$$\text{and } D_{cnt} = \{\mathsf{data} + 1\}$$

The main step of the proof is to show that the full execution specification for the initial machine configuration holds, as stated by the theorem. After some basic unfolding of definitions, it is easy to show that it follows from the specification we previously established in Lemma 2.5. □

2.7 Dynamic Memory Allocation and Closures

In the previous sections, we have shown how to use capabilities for memory protection and compartmentalization in the setting of relatively simple scenarios. In particular, the examples that we have presented so far only relied on memory allocated statically as part of the initial program region.

We now investigate how we can use and reason about more complicated programming patterns. More precisely, we show how we can implement features found in higher-level languages, such as dynamic memory allocation and function calls which guarantee encapsulation of local variables. Additionally, we implement an `assert` routine which we use to formally express properties about dynamically allocated memory.

This section focuses on presenting the aforementioned higher-level building blocks (§2.7.1–2.7.3), an updated adequacy theorem that incorporates the use of these components (§2.7.4), then followed by a simple illustrative example (§2.7.5). In

Section 2.8, we then apply them to build a larger, more significant case study, demonstrating how these building blocks can work at scale.

2.7.1 Dynamic memory allocation as a library routine

We show how dynamic memory allocation can be implemented as a library, for which: 1) we prove an Iris specification making it usable from verified code, and 2) we show that it is safe to share with untrusted code, so that an adversary can also use the library to allocate memory for its own uses.

Note that this task is made easier by the fact that we do not attempt to provide a way of deallocating memory. As such, memory provided by the allocation routine is never reclaimed. We leave deallocation for future work, as it likely requires a significantly more complex runtime mechanism to ensure that no dangling capabilities remain pointing to previously allocated memory regions [182, 55].

Concretely, we implement our allocator library as a simple bump-pointer allocator. The library provides a `malloc` entry point, to be called with an integer argument n, which works as follows:

1. the routine encapsulates a contiguous region of memory $[b, e)$, as well as a capability $(\textsc{rwx}, b, e, a)$ where the interval $[b, a)$ represents already allocated memory, and $[a, e)$ represents memory that can still be allocated;
2. the routine checks that the input size n is strictly positive;
3. if $a+n$ is greater than e, the routine fails (there is not enough memory available);
4. otherwise, it then records that memory has been allocated by updating its internal capability to $(\textsc{rwx}, b, e, a + n)$, and returns to the caller the capability $(\textsc{rwx}, a, a + n, a)$.

Figure 2.11 outlines the code for our simple `malloc` implementation. The code assumes that it is stored in memory in an interval $[b_m, b_{mid})$ and that b_{mid} points to a capability $(\textsc{rwx}, b_{mid}, e_m, a)$ giving access to: itself (so it can be updated), and the memory pool (between address $b_{mid} + 1$ and e_m). For simplicity, we assume that the non-allocated memory is already initialized to 0. These requirements are represented by the following invariant [17]:

$$\mathrm{mallocInv}(b_m, e_m) \triangleq \boxed{\begin{array}{ll} \exists b_{mid}, a, & [b_m, b_{mid}) \mapsto \texttt{malloc\_instrs} * \\ & b_{mid} \mapsto (\textsc{rwx}, b_{mid}, e_m, a) * \\ & [a, e_m) \mapsto [0 \cdots 0] * \\ & \ulcorner b_{mid} < a \leq e_m \urcorner \end{array}}$$

The core property of our safe `malloc` is that is does not hand out the same addresses across multiple dynamic allocations. This can be expressed elegantly in separation logic, by specifying that `malloc` hands out points-to resources for the allocated memory. Indeed, points-to resources ($a \mapsto w$) express full ownership over the data at address a: possessing a resource $a \mapsto w$ guarantees that one is the only owner of address a.

Consequently, remark that the invariant holds memory points-to for the region corresponding to non-allocated memory (between a and e_m), but not for the memory that has already been allocated (between $b_{mid} + 1$ and a): these resources have been handed out to previous callers of the library.

We show below the specification for `malloc` [18]. First, note that because `malloc` can fail if it runs out of memory or is given a wrong size, the specification documents that the resulting execution state is either Running or Failed. In the case where it does not fail, we can read that `malloc` hands out points-to resources for the allocated range in its post-condition: this expresses the fact that no piece of code but the caller of `malloc` can access the newly allocated memory.

$$\boxed{\text{mallocInv}(b_m, e_m)}$$

$$\vdash \left\{ (\textsc{rx}, b_m, e_m, b_m);\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto n * \\ r_2, r_3, r_4 \Mapsto - \end{array} \right\} \leadsto$$

$$\left\{ s.\ \begin{array}{ll} \ulcorner s = \text{Running} \urcorner * & \text{pc} \Mapsto \text{updatePcPerm}(w_0) * \\ & \exists b_a, e_a, \ulcorner b_a + n = e_a \urcorner * \\ & \quad \text{r}_0 \Mapsto w_0 * \\ & \quad \text{r}_1 \Mapsto (\textsc{rwx}, b_a, e_a, b_a) * \\ & \quad \mathop{\Huge *}_{a \in [b_a, e_a)} a \mapsto 0 * \\ & \quad \text{r}_2, \text{r}_3, \text{r}_4 \Mapsto 0 \\ \lor \ulcorner s = \text{Failed} \urcorner & \end{array} \right\}$$

The `malloc` routine can furthermore be encapsulated using a sentry capability, which can be shown to be safe to share with an adversary (Lemma 2.6).

Lemma 2.6 (`malloc` is safe [19]). $\text{mallocInv}(b_m, e_m) \mathrel{-\!\!*} \mathcal{V}(\textsc{e}, b_m, e_m, b_m)$

The proof is comparable to the proof that $\mathcal{V}(\textsc{e}, \text{code}, \text{end}, \text{code})$ on page 68. It relies on the `malloc` specification and the fundamental theorem.

2.7.2 Runtime checks: an `assert` routine

The final end-to-end theorems presented so far in Section 2.6 rely on establishing that a certain memory location satisfies a given invariant. This requires the relevant

```
;; r1: integer determining the number
;; of words to allocate
;;
;; malloc fails if n <= 0 or if it
;; does not have enough space left
;;
;; returns in r1 a capability to the
;; allocated memory
bm:
  lt r3 0 r1 ;; check that n > 0
  mov r2 pc  ;; jmp after fail if
  lea r2 4   ;; yes; continue and
  jnz r2 r3  ;; fail if not
  fail
xm:
  mov r2 pc
  lea r2 [bmid - xm]
  ;; r2 = (RWX, bm, em, bmid)
  load r2 r2 ;; r2 = (RWX, bmid, em, a)
  geta r3 r2
  lea r2 r1
  ;; r2 = (RWX, bmid, em, a+n)
```

```
  geta r1 r2
  mov r4 r2
  subseg r4 r3 r1
  sub r3 r3 r1
  lea r4 r3
  mov r3 r2
  sub r1 0 r1
  lea r3 r1
  getb r1 r3
  lea r3 r1   ;; r3 = (RWX, bmid, em, bmid)
  store r3 r2
  ;; bmid <- (RWX, bmid, em, a+n)
  mov r1 r4   ;; r1 = (RWX, a, a+n, a)
  mov r2 0
  mov r3 0
  mov r4 0
  jmp r0
bmid: (RWX, bmid, em, a)
;; ... already allocated memory ...
a:
;; ... free memory ...
em:
```

Figure 2.11: A simple `malloc` subroutine.

location is statically allocated in memory and thus known in advance, thus making it easy to tie it to an Iris invariant.

However, when using our `malloc` routine, we typically wish to enforce properties about the contents of dynamically allocated memory locations, whose address is, by definition, not known in advance. To address this issue, we implement an `assert` routine, to be linked alongside programs relying on `malloc`. One can invoke `assert` to dynamically test whether the contents of two registers are equal; if the test fails, `assert` sets a flag "assert has failed" at a fixed location in memory.

The idea is then that, to assert that some property holds about a piece of dynamically allocated memory, one can check dynamically whether it holds using `assert`. Then, one can *prove* that each `assert` check succeeds (meaning that the property indeed holds). Consequently, as a property of the whole execution, one gets that, at every step, the assert flag (initialized at 0) remains at 0 and is never set to 1 by `assert`.

The private memory of the `assert` routine is described by the following invariant [20]:

$$\mathrm{assertInv}(b_a, e_a, a_{flag}) \triangleq \boxed{\begin{array}{l} \exists a_{cap},\ \ [b_a, a_{cap}) \mapsto \texttt{assert\_instrs} \ast \\ \qquad a_{cap} \mapsto (\textsc{rw}, a_{flag}, a_{flag}+1, a_{flag}) \ast \\ \qquad \ulcorner a_{cap} + 1 = a_{flag} \land a_{flag} + 1 = e_a \urcorner \end{array}}$$

The address a_{flag} denotes the address of the "assert flag", which is initialized to 0 and set to 1 by the routine in case of failure. As we are interested in using `assert` in programs where we can prove that the equality check succeeds, we establish the following specification (21), which asserts in a separate invariant that a_{flag} remains at 0. Registers r_4 and r_5 contain the two integers which are compared by the routine; we thus require that they are equal.

$$\boxed{\mathrm{assertInv}(b_a, e_a, a_{flag})}, \boxed{a_{flag} \mapsto 0}$$
$$\vdash \left\{ (\mathrm{RX}, b_a, e_a, b_a); \begin{array}{c} \mathsf{r}_0 \Mapsto w_0 * \\ \mathsf{r}_4 \Mapsto n * \\ \mathsf{r}_5 \Mapsto n \end{array} \right\} \rightsquigarrow \left\{ \mathrm{updatePcPerm}(w_0); \begin{array}{c} \mathsf{r}_0 \Mapsto w_0 * \\ \mathsf{r}_4, \mathsf{r}_5 \Mapsto 0 \end{array} \right\}$$

Note that, as opposed to `malloc`, the `assert` routine should only be shared with *verified* code, which calls it according to the specification above. Were `assert` shared with an unknown adversary, the adversary could simply call the routine with two different integers, setting the flag to 1, thus invalidating any guarantees established by verified code. Technically speaking, we cannot prove safety of the `assert` routine from the specification above: if we try to prove $\mathcal{V}(\mathrm{E}, b_a, e_a, b_a)$, then we get that registers r_4 and r_5 contain two unknown (valid) words, which could be two different integers. In that case, we cannot use the specification above, as we would violate the invariant specifying that a_{flag} stays at 0.

2.7.3 A secure heap-based calling convention

We define a heap-based calling convention that uses `malloc` to dynamically allocate activation records. An activation record is encapsulated in a closure that reinstates its caller's local state, and continues execution from its point of creation. Conceptually, our heap-based calling convention can be seen as a continuation-passing style calling convention (one passes control to the callee, giving it a continuation for returning to the caller). This is similar to the calling convention that was used for instance in the SML/NJ compiler to implement an extension of Standard ML with call/cc [10] (in the setting of a traditional computer architecture).

In the setting of a capability machine, our calling convention is furthermore *secure* in the sense that it enforces local state encapsulation. In other words, one can use it to pass control to unknown adversarial code, while protecting local data of the caller, thanks to the use of sentry capabilities to implement the continuation. Note that this calling convention does not enforce well-bracketed control flow (another desirable property); see [62, 143, 144] for stack-based calling conventions that do.

We provide a `call` macro implementing the calling convention, invoked as `call` *target locals params*, where *target* is the name of the register containing a pointer to

```
; initially, PC = (RWX, code, end, a)
;            target = register containing the address to jump to
;            locals, params = lists of register names
; locals, params and target are parameters of the macro;
; they are in practice instantiated with concrete values
code:
  ...
a:
  malloc (length locals)        ; 1. allocate and store local state
  store_locals r1 locals
  mov r6 r1
  malloc 7                      ; 2. allocate region for activation record
  mov r0 r1
  store act_instr1              ;    store the activation code
  lea r0 1
  ...
  store act_instr5
  lea r0 1
  store r0 r6                   ;    store the capability to locals
  lea r0 1
x:
  mov r1 pc                     ;    prepare and store the continuation
  lea r1 [cont - x]
  store r0 r1
  lea r0 -6                     ; 3. create the return capability
  restrict r0 E
  rclear RegName\{PC,r0,r1} ∪ params ; 4. clear all registers except parameters
  jmp target                    ; 5. jump to target
cont:
  restore_locals r1 locals      ; 6. reinstate local state
  ...
data:
  (RO, table, end, table)       ; environment table
table:
  (E, bm, em, bm)               ; entry point to the malloc subroutine
  ...                           ; possibly other routines
end:
```

Figure 2.12: Heap-based calling convention, with a the first instruction in the call macro

the code to invoke, *locals* is the list of registers whose content corresponds to the local state to reinstate upon return, and *params* is the list of registers containing the parameters to the call (passed to the callee). Its implementation appears in Figure 2.12, and a representation of the corresponding memory layout in Figure 2.13. (Because `call` is defined as a macro, its code is used inline as part of a bigger program, here stored between addresses code and end.)

Before passing control to the callee, the call macro does the following:

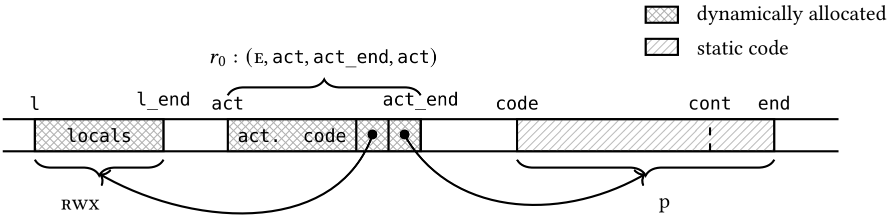


Figure 2.13: Memory layout dynamically created by the calling convention.

1. Invoke `malloc` to dynamically allocate a region of memory $[l, l_{end})$ to store the local state from the registers specified in *locals*.
2. Allocate a region of memory $[act, act_{end})$ to store the activation record, composed of: activation code, a capability to the region $[l, l_{end})$, and a capability to the instruction of the program following the call.
3. Create a sentry capability (E, act, act_{end}, act) encapsulating the activation record; this is capability for returning to the caller which is passed to the callee.
4. Clear all registers except those in *params*.
5. Jump to *target*.

When the callee passes back control to the caller by jumping to the continuation, the code stored in the activation runs first. It loads the capability pointing to local state, and returns to the old program counter set up by the call macro. As the last step, the macro will finally:

6. Restore the local state into the relevant registers from the activation record.

We show below the specification for the code of the macro up to step 5 (the jump to the target address) (22). Since the `malloc` routine is invoked by the macro, the specification relies on the corresponding invariant for `malloc`. The parameters of the macro are *params*, *locals* and *target*, respectively denoting the list of registers containing the parameters to the call, the list of registers containing local state, and the register containing the capability to jump to. The list of (encoded) instructions `act_instrs` denote the concrete instructions making up the activation code (in Figure 2.12 they are written as `act_instr1`...`act_instr5` (23)), which are not shown here for simplicity.

The post-condition of the specification describes the state immediately after the jump, where: the activation record has been allocated and initialized in $[act, act_{end})$; register r_0 contains an enter capability pointing to the activation record, and the local data has been copied to a newly allocated region $[l, l_{end})$.

$$\boxed{\text{mallocInv}(b_m, e_m)} \vdash \left\{ (p, \text{code}, \text{end}, \text{a});\; \begin{array}{l} [\text{a}, \text{cont}) \mapsto \texttt{call\_instrs} * \\ \text{data} \mapsto (\text{\textsc{ro}}, \text{table}, \text{end}, \text{table}) * \\ \text{table} \mapsto (\text{\textsc{e}}, b_m, e_m, b_m) * \\ \mathit{params} \Mapsto \mathit{pws} * \mathit{locals} \Mapsto \mathit{lws} * \mathit{target} \Mapsto w_{adv} * \\ \mathop{\Large\ast}_{\substack{(r,v)\in reg,\\ r\notin\{\text{pc},target\}\\ r\notin params\,\cup\, locals}} r \Mapsto v \end{array} \right\} \rightsquigarrow$$

$$\left\{ \text{updatePcPerm}(w_{adv});\; \begin{array}{l} \exists act, act_{end}, l, l_{end}, reg', \text{r}_0 \Mapsto (\text{\textsc{e}}, act, act_{end}, act) * \\ \text{data} \mapsto (\text{\textsc{ro}}, \text{table}, \text{end}, \text{table}) * \\ \text{table} \mapsto (\text{\textsc{e}}, b_m, e_m, b_m) * \\ \mathit{params} \Mapsto \mathit{pws} * \mathit{target} \Mapsto w_{adv} * [l, l_{end}) \mapsto \mathit{lws} * \\ [act, act_{end}) \mapsto \texttt{act\_instrs} \mathbin{+\!\!+} \\ \qquad\qquad [(\text{\textsc{rwx}}, l, l_{end}, l_{end}); (p, \text{code}, \text{end}, \text{cont})] * \\ \mathop{\Large\ast}_{\substack{(r,v)\in reg',\\ r\notin\{\text{pc},target,\text{r}_0\}\\ r\notin params}} r \Mapsto v \end{array} \right\}$$

It is then up to the user of the `call` macro to establish that the capability in r_0 is safe to share with the (possibly unknown) callee. This can be done with the help of the specification for the activation code (24), shown next:

$$\vdash \left\{ (\text{\textsc{rx}}, act, act_{end}, act);\; \begin{array}{l} \text{r}_1 \Mapsto - * \text{r}_2 \Mapsto - * \\ [act, act_{end}) \mapsto \texttt{act\_instrs} \mathbin{+\!\!+} \\ \qquad\qquad [(\text{\textsc{rwx}}, l, l_{end}, l_{end}); \\ \qquad\qquad\; (p, \text{code}, \text{end}, \text{cont})] \end{array} \right\} \rightsquigarrow$$

$$\left\{ (p, \text{code}, \text{end}, \text{cont});\; \begin{array}{l} \text{r}_1 \Mapsto - * \text{r}_2 \Mapsto (\text{\textsc{rwx}}, l, l_{end}, l) * \\ [act, act_{end}) \mapsto \texttt{act\_instrs} \mathbin{+\!\!+} \\ \qquad\qquad [(\text{\textsc{rwx}}, l, l_{end}, l_{end}); \\ \qquad\qquad\; (p, \text{code}, \text{end}, \text{cont})] \end{array} \right\}$$

One can read from this specification that the activation code passes control back to the caller (at address cont), while loading in register r_2 a capability to the region holding the local state, which can be then loaded back into the corresponding registers by the `restore_locals` macro (step 6, which we do not detail here).

To sum up, the calling convention presented here allows one to make a “function call” as one would do in a higher-level language, while protecting local data of the caller. The code invoked this way can be completely untrusted: in particular, it does not need to implement the calling convention itself for the local state encapsulation guarantees to hold. (But of course it might never “return” and pass control back to the caller.)

In Section 2.7.5, we demonstrate the use of this heap-based calling convention on a simple example, showing the interaction of its local state encapsulation guarantees with read-only capabilities.

2.7.4 Adequacy in the Presence of Dynamically Allocated Memory

We can now provide an updated version of the adequacy theorem (Theorem 2.1) which directly incorporates the `malloc` and `assert` library routines. Instead of establishing that a memory invariant is always preserved at each step, the new adequacy theorem establishes that the flag held by `assert` is never modified.

Theorem 2.5 assumes that the `malloc` and `assert` routines are loaded in memory disjoint from both *prog* and *adv*. Furthermore, the `assert` routine must have its flag initialized to 0. The verified program *prog* is given access to both the `malloc` and `assert` routines. The adversary program *adv* is given access to `malloc`. We assume that *prog* contains the code and a table that has been filled by a linker with capabilities giving access to the two routines. Likewise, we assume that *adv* contains its program (arbitrary integers) and a table filled by the linker with the capability to the `malloc` routine. Similarly to the first adequacy theorem, the theorem states that if the capability machine starts with the capability pointing to *prog* in the program counter, and if it has been proved in the program logic that the machine can run until completion, then the assertion flag is *never* modified.

In what follows, Lemma 2.5 will thus allow us to prove end-to-end theorems saying that the assertion flag will still be unset after a full execution. This corresponds to the end-to-end theorems of Swasey et al. [152] which are also phrased in terms of an assert primitive (albeit in a high-level language) that untrusted code does not get access to. Of course, such results remain a bit artificial: ultimately, in real systems, we are not directly interested in the contents of assertion flags in the system's memory, but rather in the system's interaction with the outside world: network communication, the content of displays etc. Our approach can be extended to reason about such properties, but we don't go into details here. Instead, we refer to Van Strydonck et al. [160] (Chapter 3 of this thesis), where we have done exactly this extension, by adding MMIO and external event traces to our operational semantics and using Iris invariants and ghost state to reason about them. This results in end-to-end theorems that prove security properties about the external event traces of a system, which we regard as a more realistic end goal of a verification effort.

Theorem 2.5 (Updated adequacy [25]). *Given memory fragments* $prog : [b, e) \rightarrow \mathrm{Word}$, $malloc : [b_m, e_m) \rightarrow \mathrm{Word}$, $assert : [b_a, e_a) \rightarrow \mathrm{Word}$, *and for any memory fragment* $adv : [b_{adv}, e_{adv}) \rightarrow \mathrm{Word}$, *assuming that:*

1. *the initial state of memory mem satisfies:*

$$prog \uplus malloc \uplus assert \uplus adv \subseteq mem$$

2. $[b_m, e_m)$ *contains the* `malloc` *routine;*
3. $[b_a, e_a)$ *contains the* `assert` *routine and its flag at address* a_{flag}*;*
4. *the assertion flag is initially set to 0:*

$$mem(a_{flag}) = 0$$

5. *prog contains a table linking to malloc and assert:*

$$\begin{gathered}\exists data, table, mem(data) = (\textsc{ro}, table, table + 2, table)\\ mem(table) = (\textsc{e}, b_m, e_m, b_m)\\ mem(table + 1) = (\textsc{e}, b_a, e_a, b_a)\end{gathered}$$

6. *the* only *capability in the adversary region is a table linking to malloc :*

$$\begin{aligned}&\exists data_{adv}, table_{adv}, \forall a \in \mathrm{dom}(adv) \backslash \{data_{adv}, table_{adv}\}, adv(a) \in \mathbb{Z}\\ &\qquad adv(data_{adv}) = (\textsc{ro}, table_{adv}, table_{adv} + 1, table_{adv})\\ &\qquad adv(table_{adv}) = (\textsc{e}, b_m, e_m, b_m)\end{aligned}$$

7. *the initial state of registers reg satisfies:*

$$\begin{gathered}reg(\mathrm{pc}) = (\textsc{rwx}, b, e, b), \quad reg(\mathrm{r}_0) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}),\\ reg(r) \in \mathbb{Z} \text{ otherwise}\end{gathered}$$

8. *the proof in the program logic that the initial configuration is safe given the invariants:*

$$\forall reg, \boxed{\mathrm{mallocInv}(b_m, e_m)}, \boxed{\mathrm{assertInv}(b_a, e_a, a_{flag})}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (\textsc{rwx}, b, e, b); \begin{array}{l} \mathrm{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) \ast \\ \mathop{\Large\ast}_{\substack{(r,v) \in reg,\\ r \notin \{\mathrm{pc}, \mathrm{r}_0\}}} r \Mapsto z \ast \ulcorner z \in \mathbb{Z} \urcorner \ast \\ \mathop{\Large\ast}_{\substack{(a,w) \in prog,\\ a \notin \{data, table, table+1\}}} a \mapsto w \ast \\ data \mapsto (\textsc{ro}, table, table + 2, table) \ast \\ table \mapsto (\textsc{e}, b_m, e_m, b_m) \ast \\ table + 1 \mapsto (\textsc{e}, b_a, e_a, b_a) \ast \\ \mathop{\Large\ast}_{\substack{(a,z) \in adv\\ a \notin \{data_{adv}, table_{adv}\}}} a \mapsto z \ast \ulcorner z \in \mathbb{Z} \urcorner \ast \\ data_{adv} \mapsto (\textsc{ro}, table_{adv}, table_{adv} + 1, table_{adv}) \\ \ast\ table_{adv} \mapsto (\textsc{e}, b_m, e_m, b_m) \end{array} \right\} \rightsquigarrow \bullet$$

Then, for any reg', mem', *if* $(reg, mem) \longrightarrow^{*} (reg', mem')$, *then* $mem'(a_{flag}) = 0$.

```
; initially, PC = (RWX, code, end, code)
;            r1 = (unknown) pointer to adversary function
code:
  malloc 1                ; r1 = (RWX, b, b+1, b) where b is fresh
  mov r3 r1               ; r3 = (RWX, b, b+1, b)
  mov r4 r1               ; r4 = (RWX, b, b+1, b)
  store r3 1              ; b <- 1
  restrict r4 RO          ; r4 = (RO, b, b+1, b)
  call r1 [r3] [r4]       ; call macro that jumps to r1, keeps r3 as local
                          ; state and passes r4 as parameter
  load r1 r3              ; r1 = 1, as long as b was not changed during call
  mov r2 1
  assert r1 r2            ; assert (r1 = 1)
  halt
data:
  (RO, table, end, table) ; environment table
table:
  (E, bm, em, bm)         ; entry point to the malloc subroutine
  (E, ba, ea, ba)         ; entry point to the assert subroutine
end:
```

Figure 2.14: Program passing a read-only capability to unknown callee.

2.7.5 Application: read-only sharing of dynamically allocated memory

We now present an example program sharing a read-only capability with adversary code, showcasing the combined use of the `malloc` (Section 2.7.1) and `assert` (Section 2.7.2) routines, the secure calling convention (Section 2.7.3), and exercising our updated adequacy theorem (Section 2.7.4).

Figure 2.14 shows the implementation of our program of interest. The program dynamically allocates a region of size 1, into which it stores the integer 1. Next, it creates a copy of the newly created capability, which is then restricted to read-only (RO). This restricted capability is shared with an unknown callee, while the original copy is kept as local state. Upon return, an `assert` statement checks that the region indeed still contains 1. We then wish to prove that the final assertion always succeeds.

Notice that in this example, control is passed to untrusted code, corresponding to the first scenario in Figure 2.2a. However, we also allow the callee to return, i.e. jump to a callback. This is achieved using our calling convention to create a secure two-way boundary between known code and the unknown callee.

In order to prove that the `assert` statement succeeds, we rely on two facts. First, the heap-based calling convention guarantees the encapsulation of $(\textsc{rwx}, b, b+1, b)$. Second, sharing $(\textsc{ro}, b, b+1, b)$ with unknown code does not threaten the integrity of b, since RO capabilities cannot be used to write to memory. These two facts are key when proving the following specification:

Lemma 2.7 (Full execution specification (26)).

$$\boxed{\mathrm{mallocInv}(b_m, e_m)}, \boxed{\mathrm{assertInv}(b_a, e_a, a_{flag})}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (\textit{RWX}, \mathrm{data}, \mathrm{end}, \mathrm{code}); \begin{array}{l} \mathsf{r}_1 \Mapsto w_{adv} * \mathcal{V}(w_{adv}) * \\ \bigstar_{(r,v) \in reg, r \notin \{\mathrm{pc}, \mathsf{r}_1\}}\, r \Mapsto w * \\ [\mathrm{code}, \mathrm{end}) \mapsto \texttt{code\_instrs} * \\ \mathrm{data} \mapsto (\textit{RO}, \mathrm{table}, \mathrm{table}+2, \mathrm{table}) * \\ [\mathrm{table}, \mathrm{table}+2) \mapsto \\ \qquad [(\textit{E}, b_m, e_m, b_m); (\textit{E}, b_a, e_a, b_a)] \end{array} \right\} \rightsquigarrow \bullet$$

Proof. We begin by applying program logic rules until we make it to the call to unknown code. At that point, a (fresh) region has been dynamically allocated and initialized to 1, and thus we have the following Separation Logic resources:

$$r_2 \Mapsto (\textsc{rwx}, b, b+1, b) * b \mapsto 1$$

At the call site, the calling convention creates an activation record, and sets up a sentry capability as the return pointer in r_0. (The "..." on the second line below stands for the address of the continuation after the call.)

$$r_0 \Mapsto (\textsc{e}, act, act_{end}, act) * \tag{2.11}$$
$$[act, act_{end}) \mapsto \texttt{act\_instrs} +\!\!+ [(\textsc{rwx}, l, l+1, l); (\textsc{rwx}, \mathrm{code}, \mathrm{end}, \ldots)] *$$
$$l \mapsto (\textsc{rwx}, b, b+1, b) *$$
$$\mathsf{r}_2 \Mapsto 0 *$$
$$\mathsf{r}_3 \Mapsto (\textsc{ro}, b, b+1, b) \tag{2.12}$$

Note in particular how the RWX capability pointing to b (part of the "local state") is only reachable from the capability (pointing to l) stored in the activation record, while the RO copy is available in register r_3.

The `call` macro then passes control to the adversary by jumping to w_{adv}. To reason about this jump, we apply Corollary 2.2 (assuming w_{adv} is safe). This requires us to show that all parameters in the current register state are valid. In particular, we must show that the sentry capability set up by the calling convention (2.11) is safe to execute, and that the read-only capability (2.12) is safe to share.

The latter is done by allocating an appropriate invariant, which is allowed to be *stronger* than the value relation itself, since the capability in question is read-only. To this end, we will allocate an invariant that remembers the current integer pointed to by `b`, namely 1.

$$\boxed{\exists w, b \mapsto w * w = 1}$$

That same invariant is then used to prove that (2.11) is safe to execute, in particular to show that the assert statement succeeds, and hence does not change the assert flag. □

From this functional specification, we can instantiate our updated adequacy theorem (Theorem 2.5) to then derive the following end-to-end theorem about our program.

Theorem 2.6 (End-to-end theorem: the read-only permission guarantees integrity (27)). *Starting from an initial state of the machine (reg, mem) assuming that:*

- *prog* $\uplus$ *adv* $\uplus$ *malloc* $\uplus$ *assert* $\subseteq$ *mem, where:*
  $adv : [b_{adv}, e_{adv}) \longrightarrow \text{Word}$, $prog : [\text{code}, \text{end}) \longrightarrow \text{Word}$
  $malloc : [b_m, e_m) \longrightarrow \text{Word}$ *and* $assert : [b_a, e_a) \longrightarrow \text{Word}$*;*
- *the contents of prog correspond to the encoded instructions and program data (i.e. table with capabilities to the malloc and assert subroutines);*
- *the adversary memory contains no capabilities except a table with a capability to the malloc subroutine;*
- *malloc contains the implementation of the* `malloc` *subroutine;*
- *assert contains the implementation of the* `assert` *subroutine, with its flag at address* a_{flag}*, initialized to 0;*
- *the initial state of registers satisfies:*
  $reg(\text{pc}) = (\textsc{rx}, \text{code}, \text{end}, \text{code})$,
  $reg(\mathsf{r}_1) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$.

Then, for any reg', mem'*, if* $(reg, mem) \longrightarrow^* (reg', mem')$*, then* $mem'(a_{flag}) = 0$.

Proof. We apply the updated adequacy theorem (Theorem 2.5), using the specification proved in Lemma 2.7. All that remains is to prove the validity of the adversary capability: $\mathcal{V}(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$. This is done in two steps. First, the adversary linking table is proved valid by applying validity of the `malloc` subroutine (Lemma 2.6). Next, the rest of the adversary region is proved valid through the assumption that it does not contain any other capabilities. The full proof can be found in the Coq mechanisation. □

2.8 Case study: a Library Implementing Dynamic Sealing and a Client

We have presented so far a variety of smaller examples enforcing interesting encapsulation properties while interacting with adversarial code. In this section, we demonstrate that our approach scales up to the verification of a larger case study, involving not only the building blocks of Section 2.7, but using them to build and modularly verify a number of libraries built on top of each other.

We take inspiration from the literature on *object capability patterns* (OCPs) from high-level languages, a technique that enables programmers to protect the private state of their objects from corruption by untrusted code. More precisely, we consider the *dynamic sealing* OCP as presented by [152]. Dynamic sealing enforces a form of data abstraction in the absence of static types. It can be implemented as a library providing pairs of seal/unseal functions, allowing their clients to "seal" private data into opaque objects which can be safely shared with untrusted code, and later unsealed in order to get back the original data.

In the context of a high-level language, [152] present a formally verified implementation of dynamic sealing, equipped with a specification that captures the abstraction guarantees it provides. The authors then use this dynamic sealing library to build and verify a library of abstract integer intervals, where the integrity of an interval value (representing a range $[i, j)$ with $i \leq j$) is protected using dynamic sealing. Finally, the authors use their verified integer library to establish *robust safety* of a simple client program checking integrity of intervals, establishing that an untrusted context cannot violate the internal invariants of the program and its underlying libraries.

We implement and verify low-level variants of the dynamic sealing OCP, interval library, and their robustly safe client. This represents a non-trivial amount of code: our implementation of those three components adds up to 632 machine instructions. Nevertheless, despite the fact that those libraries are implemented in low-level assembly code, we are able to give them specifications at a level of abstraction similar to their high-level counterparts.

For ease of reading, we will keep the explanations fairly high-level. We will first show high-level pseudo-code for the implementation of the interval library and its client, and informally discuss what kind of properties should be enforced. Then, we will present the key ideas for implementing dynamic sealing on a capability machine, and then for reasoning about it, in particular how to instantiate its specification to be able to verify the interval library.

```
interval㉘ = λ_,  let (seal, unseal) = makeseal() in
                 let makeint = λ z₁ z₂,  let x = malloc(2) in
                                         x ← {min(z₁, z₂); max(z₁, z₂)};
                                         seal(x)
                 in
                 let imin = λ i,  unseal(i)[0] in
                 let imax = λ i,  unseal(i)[1] in
                 (makeint, imin, imax)


      client㉙ =  let (makeint, imin, imax) = interval() in
                 let checkint = λ i, assert(imin(i) ≤ imax(i)) in
                 (checkint, makeint, imin, imax)
```

Figure 2.15: High-level pseudo-code for the implementation of the interval library and its client.

2.8.1 Interval Library and Client

The interval library implements an abstract data type representing intervals. An interval has a lower and upper bound, which can be extracted via two functions; imin and imax. An interval is created via a function makeint that takes as input two integers, and chooses the smallest input as the lower bound, and the largest input as the upper bound. Crucially, the internal representation of an interval must stay hidden so as to guarantee its integrity.

We thus use *dynamic sealing* ([149]) to dynamically enforce data abstraction for the intervals representation. We detail our implementation of seals in Section 2.8.2. For now, it suffices to know that a seal is a pair of functions, seal and unseal, where the former hides the internal representation of some value, such that only the latter can expose it.

An interval can be represented as an ordered pair of integers. On the capability machine, we implement such a pair as a dynamically allocated region of size two, storing the lower and upper bound of the interval. Then, an interval itself consists of a capability with read/write authority over the corresponding region of size two. In Figure 2.15, we depict the high-level implementation of our interval library. Note that the library implements closures around a fresh seal-unseal pair, used to seal the aforementioned internal representation of intervals. The low-level implementation that we formally reason about can be thought of as the result of compiling the high-level implementation shown in Figure 2.15.

The same figure also depicts a client of the interval library. The client exposes four entry points to the environment: in addition to entries to `makeint`, `imin` and `imax` from a fresh instance of the interval library, the client also exposes an encapsulated `checkint` function that, given an interval, dynamically asserts that the expected representation invariant holds for the interval, that is, that the minimum of the interval is indeed smaller than or equal to the maximum of the interval.

When formally verifying the interval library and its client, we will need an invariant to keep track of each interval created by `makeint`. The invariant should capture the properties enforced by the implementation of the interval library. We can already list the internal properties of an interval intuitively. First and foremost, the lower bound of an interval must be less than or equal to its upper bound. A perhaps more subtle property is that intervals are immutable. Thus, we will need to define an invariant that represents each interval as a dynamically allocated region of size two, which stores the lower and upper bound, and is immutable. The `seal-unseal` pair encapsulated by the library will be used only to seal intervals that adhere to this representation (satisfy this invariant). Keeping this intuition in mind, let us now explore the technical implementation of seals.

2.8.2 Dynamic Sealing

Dynamic sealing makes it possible to support data abstraction dynamically. The function `makeseal` creates a pair of functions, `seal` and `unseal`, where `seal` is used to seal a word w into a fresh sealed word σ. We will also refer to σ as the key to w. The only way to extract the word w from σ is with `unseal`. The key point is that this `seal-unseal` pair supports data abstraction by *sealing away* or *hiding* the internal representation of some value, only known and available to the owner of the associated `unseal` function.

Although capability machines such as CHERI include seals as a language primitive, we show here how we can implement seals in software, as a low-level library. The library is implemented via a data structure that stores each word sealed through `seal`, associating each sealed word with a key. A key in itself does not reveal any details about the word it is hiding. However, it can provide access to that word, granted one has the proper authority to unseal it. Only a valid key should grant access to a sealed word. Keys, and the data structure that uses them, should intuitively satisfy two properties; (1) the unforgeable nature of keys and (2) the unique association between a key and the word it seals.

The `seal` and `unseal` subroutines respectively perform insertions and lookups in this underlying data structure. `seal` takes a word as input, generates a fresh key, and adds the key value association to the data structure. `unseal` takes a key as input, checks

SEAL SPEC ㉚

$$\left\{\begin{array}{ll} & [b_s, e_s) \mapsto \texttt{seal} \ast \\ (-, b_s, e_s, -); & \mathsf{sealInv}\ ds\ \Phi \ast \\ & \mathsf{r}_1 \Mapsto v \ast \Phi(v) \ast \cdots \end{array}\right\} \rightsquigarrow \left\{ s\ k. \begin{array}{l} \ulcorner s = \mathsf{Running} \urcorner \ast \\ \quad \mathsf{isSealedWord}\ k\ v \ast \\ \quad \mathsf{r}_1 \Mapsto\ k \ast \cdots \\ \lor \ulcorner s = \mathsf{Failed} \urcorner \end{array} \right\}$$

UNSEAL SPEC ㉛

$$\left\{\begin{array}{ll} & [b_u, e_u) \mapsto \texttt{unseal} \ast \\ (-, b_u, e_u, -); & \mathsf{sealInv}\ ds\ \Phi \ast \\ & \mathsf{r}_1 \Mapsto k \ast \cdots \end{array}\right\} \rightsquigarrow \left\{ s\ v. \begin{array}{l} \ulcorner s = \mathsf{Running} \urcorner \ast \\ \quad \mathsf{isSealedWord}\ k\ v \ast \\ \quad \mathsf{r}_1 \Mapsto\ v \ast \Phi(v) \ast \cdots \\ \lor \ulcorner s = \mathsf{Failed} \urcorner \end{array} \right\}$$

Figure 2.16: Specifications of `seal` and `unseal`.

that the key is associated to a value in the data structure, in which case it returns the value.

Reasoning about dynamic sealing

A shared `seal-unseal` pair can be used to seal any word. In practice, one typically encapsulates a `seal-unseal` pair within a library, performing additional checks and thus ensuring that words that are sealed always satisfy a specific property. Then, whenever one successfully unseals a given key, one gets that the corresponding word satisfies the chosen property. For instance, the interval library enforces that each sealed word is a region of size 2, storing the ordered bounds of an interval.

When reasoning about code invoking the dynamic sealing library, one will need to pick, for each `seal-unseal` pair, a *representation invariant* $\Phi : \mathrm{Word} \to iProp$ describing the values to be sealed/unsealed by the pair[7]. Then, each `seal-unseal` pair maintains an Iris invariant sealInv describing the state of the pair itself, namely the data structure storing the key-values for all sealed entries. Additionally, this invariant stores the information that each sealed value satisfies Φ.

$$\mathsf{sealInv}\ ds\ \Phi\ ㉜ \quad \triangleq \quad \boxed{\begin{array}{l} \exists wvals, \mathsf{dataStructure}\ ds\ wvals \\ \ast \bigast_{(-,w) \in wvals}\ \Phi(w) \end{array}}$$

We require that Φ is persistent, since the representation invariant of a sealed word should always hold once sealed. The dataStructure predicate describes the state of the data structure internal to the seal library (see Section 2.8.2 for a formal definition). It asserts that *ds* can be used to access a data structure storing the key value pairs denoted by *wvals* (a sequence of pairs in Addr × Word). In other words, *wvals* is

[7] An analogous representation invariant is used in the [152]

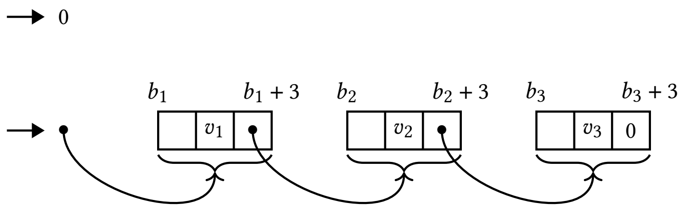


Figure 2.17: In-memory representation of an empty dictionary linked list and a dictionary linked list with three values v_1, v_2 and v_3.

the complete list of all words that have been sealed so far, each paired with their associated key.

A sealed word is sealed forever. It is thus possible to persistently remember that a particular word is an element of *wvals*. The predicate isSealedWord k v states that the key k is uniquely associated with the sealed word v. We present the formal definition of isSealedWord in Section 2.8.2.

The functional specifications of the `seal` and `unseal` subroutines depend on an instance of the seal invariant sealInv, for a specific user-provided predicate Φ. Then, `seal` can only be applied to words for which the representation predicate Φ holds. `unseal` can fail if a given key is not valid, or if it is not associated with any sealed word, however if it succeeds, it will return a word for which Φ holds. The specification of `makeseal` allocates a fresh sealInv instance, for any Φ chosen by the client of the library. Figure 2.16 shows specifications for `seal` and `unseal` (where we omit low-level administrative details).

Implementing a low-level seal library

We now present the data structure used to implement the low-level seal library. We implement it as a linked associative list with a twist, next referred to as a *linked list dictionary*. The trick is to take advantage of the unforgeable nature of capabilities, and use the capability to (a subrange of) a list node as a key to that node; the corresponding value being then stored in the node.

Figure 2.17 shows the in-memory representation of a linked list dictionary storing three key-value pairs. Each node is implemented as a region of size three, where the bottom address acts as the key address. To avoid access to sealed values, it is important that a key does not provide authority over the other parts of a node (the value and the next pointer). For instance, the value v_1 is uniquely associated to the capability $(\textsc{rwx}, b_1, b_1 + 1, -)$.

The linked list dictionary library contains two subroutines, `findB` (33) and `append` (34). `findB` expects as input an integer b, searches the linked list for a node of the form $(\textsc{rwx}, b, b + 3, -)$ and returns the value that the associated node stores. It fails if no such node exists. `append` expects a word as input, invokes `malloc` to dynamically allocate a new node of size three, stores the input word in the second position of that node, and then stores that node as the new tail of the linked list. Finally a key can then be derived from the newly created node; we now explain in more detail how that is done.

A fresh instance of a `seal`-`unseal` pair is created by calling the `makeseal` subroutine, which returns a pair of closures encapsulating a new empty linked list dictionary. Sealing a word w adds it to the dictionary, and returns a *restricted* capability representing the key to the linked list dictionary entry. Say for instance that the input word w is appended to the list in a fresh node $(\textsc{rwx}, b, b + 3, b)$. The `seal` subroutine will then return the key $(\textsc{rwx}, b, b + 1, -)$ (the address pointed to does not matter, and is here omitted for clarity).

Recall that in the enclosed linked list dictionary, w will be stored at address $b + 1$, for which the returned sealed value, or key, does not have authority. This sealed value is *unforgeable*. The only way to create it would be to derive it from a capability $(\textsc{rwx}, b', e', \_)$ where $[b, b + 1) \subseteq [b', e')$. However, this is impossible since the appended node is freshly allocated using a safe `malloc` subroutine, which is guaranteed to hand out fresh regions upon invocation. Only `seal` has access to such a capability, and thus sealed values cannot be forged.

In turn, the `unseal` subroutine expects a $\textsc{rwx}$ capability of range 1 as input. It reads its lower bound, searches the enclosed linked list for a node with matching lower bound, and returns the associated word. Let us consider a continuation of the previous example. Say that `unseal` receives $(\textsc{rwx}, b, b + 1, -)$ as input. It begins by authenticating the key by dynamically verifying its permission to be $\textsc{rwx}$, and its size to be 1. Upon validating its permission and range, it then runs `findB` on the enclosed linked list dictionary with the integer b, and returns the word stored within the node $(\textsc{rwx}, b, b + 3, -)$ at address $b + 1$, namely the previously sealed word w. The authentication guarantees that a key has the same unforgeable authority as when it was created.

In summary, the `seal` and `unseal` subroutines are implemented as follows:

- `seal`:
  1. append the input to the enclosed linked list dictionary
  2. restrict the range of the fresh node capability to bottom address of node

3. return resulting restricted capability

- `unseal`:
  1. check that permission of input is RWX
  2. check that the range of input is 1
  3. get the lower bound of input
  4. find the node in the linked list dictionary with same lower bound
  5. return the stored word at that node (fail if no such node exists)

We now have enough ingredients to revisit the predicates used in the previous section to define the seal invariant. Recall that the dataStructure predicate represents the state of the data structure internal to the seal library (now defined to be a linked list dictionary), and that the isSealedWord predicate describes a persistently known association between a sealed word and its key.

$$\begin{array}{rcl} \text{dataStructure } ds\ wvals & \triangleq & \exists hd,\ ds \mapsto hd \\ & & * \text{ isList } hd\ wvals \\ & & * \text{ Exact } wvals \\ \text{isSealedWord } k\ v & \triangleq & \exists wvals,\ \text{Pref } wvals * \ulcorner (k, v) \in wvals \urcorner{}^{8} \\ & & * \mathcal{V}(\textsc{rwx}, k, k+1, -) \end{array}$$

The head of the linked list dictionary is stored in location *ds*. isList corresponds to a standard inductive separation logic predicate for linked lists, with one caveat: it does not take ownership of the first location, i.e. the key address, depicted in each node of Figure 2.17. An invariant owning this key address will be passed to the adversary as part of the $\mathcal{V}$ relation in the isSealedWord predicate. Since the list monotonically grows, it is useful to persistently remember any prefix of the linked list dictionary. Exact *wvals* (the authoritative view of the list state) roughly states that *wvals* is the full state of the data structure. Pref *wvals* (the local fragment view) states that *wvals* is a prefix of the data structure. isSealedWord k v, a persistent predicate, states that the word v has been sealed with a key; a capability with lower bound k. This key is safe to share, hence $\mathcal{V}(\textsc{rwx}, k, k+1, -)$ holds.

In the next section, we describe how we use the reasoning principles about `seal-unseal` to verify our interval library.

2.8.3 Verifying the Interval Library and its Client

The first key step is to formally define the representation invariant for an interval. Recall the intuitive description given in Section 2.8.1: an interval is a capability with

[8] In the Coq mechanization, *wvals* associates the word w to $k + 1$ rather than k, for technical reasons. This small discrepancy has otherwise no impact on the rest of the proof.

authority over a region of size 2, storing the lower and upper bounds of an interval, and which is immutable.

A first thought might be that one can define the representation invariant using two points-to predicates for the region. However, this does not capture the immutability of intervals, nor is it persistent. Instead, we use *persistent* points-to predicates ([167]). A persistent points-to predicate $a \hookrightarrow w$ asserts that address a stores the word w. It can be used to read from address a, but not write to it, and as such, is a persistent resource. This is exactly what we need for our immutable invariants. We formally define the representation invariant isInterval as follows:

$$\begin{aligned} \text{isIntervalInt}\ z_1\ z_2\ w\ ㉟ &\triangleq \exists a, \ulcorner w = (\textsc{rwx}, a, a+2, a) \urcorner * \\ & \quad a \hookrightarrow z_1 * (a+1) \hookrightarrow z_2 * \ulcorner z_1 \leq z_2 \urcorner \\ \text{isInterval}\ ㊱ &\triangleq \lambda w, \exists z_1\ z_2, \text{isIntervalInt}\ z_1\ z_2\ w \end{aligned}$$

(Note, in particular, that the invariant also captures the property that the lower bound is less than or equal to the upper bound.) Using properties of persistent points-to predicates, we can prove the following lemma:

Lemma 2.8 (㊲).

$$\text{isIntervalInt}\ z_1\ z_2\ w \longrightarrow \text{isIntervalInt}\ z_3\ z_4\ w \longrightarrow \ulcorner z_1 = z_3 \land z_2 = z_4 \urcorner.$$

Because isInterval is persistent, we can use it as the representation predicate for a `seal-unseal` pair, which will thus operate over the following invariant:

$$\text{sealInv}\ ll\ \text{isInterval}$$

This seal invariant is allocated by the specification for `makeseal`, which is invoked during the creation of an interval library closure.

When sealing a new interval using `makeint`, we must establish isInterval for the newly created interval. This requires us to transform the regular points-to predicates handed out by the `malloc` specification into persistent points-to predicates, and assert that indeed $min(z_1, z_2) \leq max(z_1, z_2)$.

Specifications for `imin` and `imax` return the respective lower and upper bound of a sealed interval. The seal invariant guarantees that the sealed word is an interval according to the representation invariant isInterval. In other words, if `imin` or `imax` succeeds for some word w, we know that w is the key to some associated capability pointing to the bounds of an interval $[l, r]$; specifically that isIntervalInt $l\ r\ w$ holds.

During the verification of `checkint`, the specification for `imin` gives us some value l and predicate isIntervalInt $l\ r\ w$. Similarly, the specification for `imax` gives us some value r' and predicate isIntervalInt $l'\ r'\ w$. Notice that the bounds may be different, but the sealed word w is the same in each instance. We can thus apply Lemma 2.8

on the two given instances of isIntervalInt, and use the definition of isInterval to conclude that the given `assert` statement succeeds, namely that $l \leq r$.

Finally, all that remains is to apply adequacy and prove the following final end-to-end theorem:

Theorem 2.7 (End-to-end theorem: the interval client does not trigger an assertion failure (38)). *Starting from an initial state of the machine* (reg, mem) *in which regions reserved for the interval library, the seal library,* `malloc`*, the assert flag, the client and the adversary are all disjoint, and initialized as expected, we have that, for any* reg'*,* mem'*, if* $(reg, mem) \longrightarrow^* (reg', mem')$ *then* $mem'(a_{flag}) = 0$*.*

2.9 Discussion and Perspectives

In this paper we have introduced Cerise, a program logic for reasoning about a low-level capability machine. Moreoever, we have shown how Cerise can be used to define a logical relation for reasoning about unknown code. Thanks to the logical relation and the fundamental theorem from Section 2.5, Cerise can be used for *robust verification* [152, 136], i.e., to verify correctness of software that interacts with unverified components. The Cerise program logic is the culmination of ideas used in a sequence of earlier papers [142, 143, 62, 160] and this paper is intended to give an accessible and didactic introduction to Cerise and the application of Cerise to program verification in the presence of untrusted code, accompanied with new results on a heap-based calling convention and implementations of sophisticated object-capability patterns.

Throughout the paper we have introduced increasingly complex examples, which demonstrate how fine-grained abstractions can be implemented on a capability machine and reasoned about using Cerise. Our examples from Section 2.7 and Section 2.8 are modeled after examples from a paper about a high-level object capability language [152]. Because of the more low-level nature of our capability machine, we had to implement some abstractions ourselves (such as the calling convention in Section 2.7.3) but we think it is otherwise fair to say that our examples faithfully represent the examples used by Swasey et al., using the same granularity of encapsulation and attacker interaction. As such, this paper demonstrates that the low-level security primitives offered by our capability machine are expressive enough to implement high-level language abstractions, despite the stronger attacker model of a low-level adversary. At the same time, the examples show that Cerise is expressive enough to reason about these abstractions.

Cerise is the first instantiation of the Iris framework to such a low-level language and thus this work also demonstrates that the key features of Iris (such as guarded recursion, ghost state, and invariants) are equally applicable in this low-level setting as in the high-level settings they were originally intended for.

Of course, while we implement and reason about our examples directly in the capability machine assembly language, we are not proposing that real software should all be developed in that way. On the contrary, we think this is only realistic for low-level code in compiler back-ends [143, 62], operating systems and low-level security measures [160]. Other software should be developed and reasoned about in a more abstract setting, which suggests the need for a secure compiler that preserves high-level security guarantees in a low-level environment. In the context of capability machines, such compilers have been investigated already, both formally [162, 51], and practically [32, 131]. While we in this work have shown how to implement and reason about some high-level programming patterns at a low level, much interesting work remains to be done to further explore the design of a high-level language whose security abstractions map well to those offered by a capability machine.

An important aspect of the universal contract provided by our logical relation and fundamental theorem is that it formalizes the security guarantee of our capability machine without overspecifying implementations of the ISA. The contract specifies an authority bound that suffices to reason about adversarial code, but does not overly constrain future extensions or optimized implementations of the ISA. This is similar to how the ISA itself is designed to specify expected behavior that is sufficient for software authors to reason about their code without preventing CPU designers from constructing optimized or extended implementations. In fact, we believe universal contracts offer a general and powerful approach for formalizing ISA security guarantees. Such security guarantees are informally stated in informal ISA specifications but they have not yet been incorporated in formal definitions of ISAs [14, 24]. As such, a promising application of universal contracts like the one from Section 2.5 is to incorporate them into the ISA definition to formalize intended ISA security guarantees.

Finally, it is worth acknowledging that in this paper, we only describe a minimal capability machine that lacks many features from realistic capability machine ISAs. Our approach has been extended to support some additional features in the literature (e.g., local and uninitialized capabilities [62], and MMIO [160]), but other features are still missing for now (e.g. sealing, interrupts, virtual memory, etc.). In terms of reasoning, the unary model we have described only supports reasoning about integrity properties. However, we have implemented a binary model in our Coq development which can be used to reason about relational properties (e.g., confidentiality).

2.10 Related work

We now discuss several lines of work related to ours. First, we discuss earlier variants of Cerise by the authors and colleagues. Then, we discuss work on verifying object capability patterns in *high-level* languages, verification of ISA properties *in CHERI*, and other applications of *universal contracts* in the literature.

2.10.1 Earlier variants of Cerise

Earlier variants of Cerise focused on showing how capabilities can be used to implement a *secure, stack-based calling convention* [143, 144, 62] and *nested security wrappers* (Chapter 3 of this thesis) [160].

Skorstengaard et al. [143] were the first to show that capabilities can be used to implement a secure stack-based calling convention, i.e., a calling convention where the security guarantees of function calls at the machine code level are faithful to the high-level notion of a function call. They employed an additional kind of "local" capabilities and stack clearing to achieve security. Their work follows a similar methodology as the one described here, that is, they define a logical relation which characterizes a notion of safety. However, their proofs were not mechanized and the logical relation was defined using a non-trivial concrete model; in contrast we use the Cerise program logic to define and prove properties about our logical relation, which means that our development is done at a higher-level of abstraction and thus we, e.g., do not have to solve any recursive domain equations. In follow-up non-mechanized work, Skorstengaard et al. [144] achieved similar security guarantees with a novel calling convention based on so-called "linear" capabilities; capabilities that can never be duplicated. Although this calling convention avoids the stack clearing required in the previous work, linear capabilities come with certain architectural restrictions [see e.g. 144, §6.2]. An efficient implementation of linear capabilities has so far not been demonstrated.

The subsequent work by Georges et al. [62] introduced a new type of capabilities (called "uninitialized") to avoid most of the stack clearing from Skorstengaard et al.'s first calling convention, thereby improving runtime efficiency. Importantly, uninitialized capabilities do not come with the same architectural hurdles as linear capabilities. As a second contribution, Georges et al. used Iris to formulate safety as a logical relation and mechanized their proofs of security.

The aforementioned logical relations of both Skorstengaard et al. and Georges et al. are more expressive and therefore significantly more complicated than the one presented here: they permit reasoning about revocation of local/linear/uninitialized capabilities and well-bracketedness properties of machine-code "function calls", on top of local-state encapsulation. In our present work, object capabilities ensure local state encapsulation, but we do not enforce calls and returns to be well-bracketed. In particular, we do not prevent an adversary from invoking a return pointer several times, or storing return pointers for later use. In other words, our calling convention implements the kind of function calls one has in a high-level language with control operators (e.g., call/cc), where calls and returns are not necessarily well-bracketed. (It is well-known that models of well-bracketed function calls are more involved than models of not-necessarily-well-bracketed function calls, see, e.g., [6, 50], and here we opted for the latter, to present a more accessible model, which suffices for a heap-based calling convention and for studying low-level implementations of object-capability patterns.)

In a different line of work, Van Strydonck et al. [160] employed a capability machine and logical relations model similar to the one presented here, but with additional support for MMIO, to verify safety properties for small, nestable wrappers around security-critical devices on a capability architecture. As part of the verification effort, multiple end-to-end security theorems were proven, which state that safety predicates of interest hold over the trace of IO events admitted by the machine. Here we have instead focused on demonstrating how a core model (without MMIO support) can be used to reason about low-level implementations of object-capability patterns.

2.10.2 Verifying object capability patterns in high-level languages

A number of high-level programming languages allow for programming patterns similar to object capabilities, that enable preserving local state while interacting with unknown code. Examples are closures, and high-level objects in capability safe languages.

Devriese et al. [44] pioneered the use of a logical relation to give a semantic characterization of capability safety (earlier work used a more conservative syntactic approach based on whether or not objects contain references to each other and ignored the behaviour of objects). Devriese et al. focused on capability safety for a core calculus of Javascript, including a notion of observable effects, and used an explicit construction of their logical relation (not a program logic), which was the inspiration for the capability model by Skorstengaard et al. [143] mentioned above and for the work by Swasey et al. [152], who presented a program logic which allows reasoning modularly about object capability patterns in a high-level language. The methodology of Swasey et al. is close to the one presented here, but in contrast to Swasey et al. we reason about object capabilities on a low-level machine. For instance, Swasey et al. define two predicates to describe a reference: a predicate for "high integrity" locations ($\ell \hookrightarrow v$), and one for "low integrity" locations ($\mathsf{lowloc}\ \ell$).

The first predicate grants exclusive access to the corresponding reference, and is therefore not safely shareable with an adversary. The second is shareable with an adversary, but can only be used to read and write "low integrity" values. In our setting, "high integrity" directly corresponds to the predicate $a \mapsto w$ for a memory location, and "low integrity" corresponds to the invariant used in the definition of $\mathcal{V}$: $\boxed{\exists w, a \mapsto w * \mathcal{V}(w)}$. Correspondingly, our definitions satisfy similar reasoning rules to the ones established by Swasey et al.. In particular, we believe that the various object capability patterns they verify can be implemented and verified in a similar way in the setting of a capability machine, using the principles presented in this paper. We demonstrated one such implementation by adapting their dynamic sealing example in Section 2.8. Additionally, the robust safety theorem of Swasey et al. is related to our template adequacy theorem with malloc and assert (Theorem 2.5); our assert flag plays a role similar to their OK flag.

2.10.3 Verifying ISA properties in CHERI

Nienhuis et al. [110] formally verify a number of "architectural" properties of CHERI capability machines. This constitutes a significant mechanization effort: the authors tackle the full generality of a realistic operational semantics for CHERI, which is significantly more complex than the minimal machine we consider here. The approach followed by Nienhuis et al. is different from ours: they state the properties they establish as trace properties, over a trace of "abstract actions" describing the various capabilities transiting through the machine during the execution. This approach makes it possible to state the desired properties in a very explicit and concrete fashion. For instance, the authors state and prove a property of "capability monotonicity": during the execution, the authority of available capabilities cannot increase (in other words, the machine does not allow forging new authority). Intuitively, this seems like a very reasonable property, required for proper operation of the capability machine. However, in practice it is more subtle: calls between components (in our case, jumping to an E-capability) do allow for some restricted form of non-monotonicity. The property proved by Nienhuis et al. is thus restricted to trace fragments that do not include calls to a different component. Our methodology is less explicit, but more expressive. In our setting, the fundamental theorem can be understood as expressing that "the machine works well". Its very extensional statement is admittedly harder to understand in terms of the operational semantics of the machine, but it enables deriving correctness statements in terms of the operational semantics that do apply to a full execution of the machine, including calls between an arbitrary number of components.

2.10.4 Other applications of universal contracts

As mentioned, our fundamental theorem constitutes a universal contract for arbitrary code, i.e., it allows deriving the guarantee that *any* adversarial capability is safe to execute, given validity of said capability. This safety is typically obtained by syntactically restricting the adversarial capability; e.g., requiring that the adressed memory only contains integers.[9] Similar notions of universal contracts have been used for high-level languages (explicitly or implicitly) in the literature. The aforementioned work of Skorstengaard et al. [143, 144], and Swasey et al. [152] all used a version of universal contracts, and placed varying syntactic restrictions on adversaries. The semantic type systems of Jung et al. [78] and Sammler et al. [136] permit similar reasoning about untrusted code based on a syntactic well-typedness restriction. The back-translation in the full-abstraction proof by Van Strydonck et al. [162] (Chapter 5 of this thesis) involved an explicit, universal separation logic contract for a C-like language with capabilities. Generally, whenever a semantic model is used to describe semantic guarantees satisfied by arbitrary code (possibly subject to syntactic restrictions), and when these guarantees are used in the manual verification of other code, this can be regarded as an application of a universal contract.

Acknowledgements Thanks to Léon Gondelman and Pierre Pradic for feedback on earlier drafts of this document.

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation; by the Research Foundation - Flanders (FWO); and by DFF project 6108-00363 from The Danish Council for Independent Research for the Natural Sciences (FNU). Thomas Van Strydonck holds a Research Fellowship of the Research Foundation - Flanders (FWO). Amin Timany was postdoctoral fellow of the Flemish Research Foundation (FWO) during parts of this project.

[9] Note that instructions are encoded in memory as integers.

Chapter 3

Proving full-system security properties under multiple attacker models on capability machines

Publication Data

This chapter contains the following paper:

> Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. Proving full-system security properties under multiple attacker models on capability machines. Accepted for publication at CSF22, 2022

It explores the addition of MMIO to the Cerise model. Most of the implementation work was done by me, with help from my co-authors to finalize everything in a timely fashion. The same holds true of the writing, where my co-authors made important contributions and improvements.

Abstract

Assembly-level protection mechanisms (virtual memory, trusted execution environments, virtualization) make it possible to guarantee security properties of a full system in the presence of arbitrary attacker provided code. However, they typically only support a single trust boundary: code is either trusted or untrusted, and protection cannot be nested. Capability machines provide protection mechanisms that are more fine-grained and that do support arbitrary nesting of protection. We show in this paper how this enables the formal verification of full-system security properties under multiple attacker models: different security objectives of the full system can be verified under a different choice of trust boundary (i.e. under a different attacker model). The verification approach we propose is modular, and is *robust*: code outside the trust boundary for a given security objective can be arbitrary, unverified attacker-provided code. It is based on the use of *universal contracts* for untrusted adversarial code: sound, conservative contracts which can be combined with manual verification of trusted components in a compositional program logic. Compositionality of the program logic also allows us to reuse common parts in the analyses for different attacker models. We instantiate the approach concretely by extending an existing capability machine model with support for memory-mapped I/O and we obtain full system, machine-verified security properties about external effect traces while limiting the manual verification effort to a small trusted computing base relevant for the specific property under study.

3.1 Introduction

Assembly-level security primitives are a cornerstone of secure systems, and they come in many forms. CPU-supported security mechanisms like virtual memory, trusted execution environments, virtualization, micro-policies or capability machines all offer a form of encapsulation, which supports the execution of untrusted code with restricted authority. Some architectural security mechanisms, such as virtual memory, virtualization or trusted execution environments, are carefully optimized for specific security abstractions, such as processes, virtual machines or enclaves, and provide poor support for features which fall outside of these abstractions. In this paper, we are interested in such a feature, which is poorly supported by the most-used security mechanisms: *nested encapsulation*. This refers to scenarios where an encapsulated piece of code (e.g. a user process or virtual machine) further encapsulates a subset of its own code from the rest of its code. When nested encapsulation is poorly supported by the architectural primitives, it can only be supported at the cost of additional effort, complexity and a certain performance loss. For example, library OSs like Graphene require host OS cooperation to implement process isolation [155] and running virtual machines inside virtual machines requires additional context switches with additional overhead [19].

Contrary to many other primitives, micro-policies [41] and capability machines [94, 28, 176] are designed explicitly for generality and flexibility. In particular, capability machines offer good support for nested encapsulation, as we now explain. On a capability machine, capabilities are used to represent authority explicitly. Different forms of capabilities represent authority over memory, the authority to invoke other code, etc. Unprivileged code can easily set up an encapsulation boundary by constructing an *object capability*: an opaque capability that can be invoked by other code and only makes private state available after invocation. This private state can in turn include other capabilities representing additional authority. An object capability invocation hence represents a context switch between security domains.

Building on related work in high-level languages [44, 152, 162], Skorstengaard et al. and Georges et al. have developed a methodology for *robust modular verification* of software running on capability machines [142, 62] that supports proving (security) properties in the presence of untrusted code. The idea is to formalize the hardware-provided security guarantees in the form of a *universal contract*: a separation logic contract that holds for arbitrary, untrusted code on the machine. This universal contract expresses a form of capability safety: the untrusted code's authority is effectively bounded by the authority of the capabilities it is given access to. In robust modular verification, the program logic allows combining manual verification of a property for certain components with this universal contract for untrusted code to obtain a full-system proof of the property.

For now, this work has remained restricted to proving artificial security properties: for example the fact that an assertion failure flag will never be set [62, 143] or equi-termination of programs [144]. In this paper, we extend this robust modular verification approach to a capability machine with memory-mapped I/O (MMIO). Although this is not technically the most complex feature to add, it does allow us to prove end-to-end system properties that are more interesting and realistic. Contrary to the artificial properties of previous work, our results specify that a security property holds for the system's trace of external effects, which we believe is the ultimate goal of verification in many practical settings.

Additionally, we extend the approach to reasoning about nested encapsulation. To formally verify intended security properties in the presence of nested encapsulation, we analyze the system several times using different attacker models, corresponding to different scenarios that the encapsulations are (explictly or implicitly) designed for. [1]

[1] Although we use terms like attacker model and adversary in this paper, we use the terms as synonyms for trust model and untrusted code, as we don't necessarily mean that the corresponding components are malicious, but perhaps simply faulty or vulnerable to security exploits. We do not distinguish such

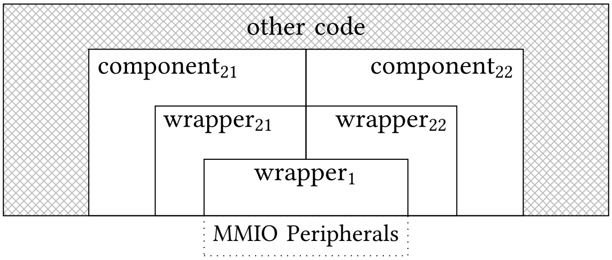


Figure 3.1: An example architecture of nested parapass-through wrappers around the peripherals.

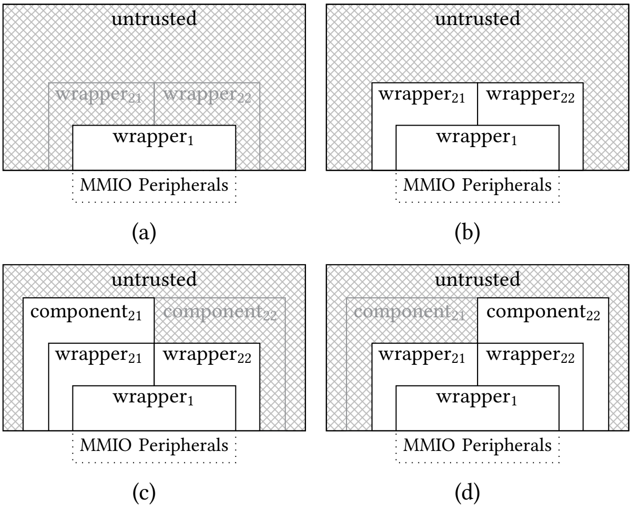


Figure 3.2: Four example attacker models, which could be used for analysing security of the system in Figure 3.1.

To make our approach concrete, we study the equivalent of BitVisor's parapass-through virtualization [140], where small wrappers enforce security policies on the interaction with peripherals. Such wrappers rely on a very small TCB and lend themselves well to verification. Particularly, we consider scenarios like the one depicted in Figure 3.1 where different parties install wrappers in a nested way. For example, a hardware manufacturer could install a wrapper with exclusive access to certain peripherals and read and write callbacks which other code on the system can use to interact with the peripherals. The wrapper could restrict configuration parameters (perhaps depending on the options purchased from the device manufacturer) simply by applying a bounds check on values written to a particular MMIO address. On top of this, a device manufacturer could install a second wrapper that keeps a counter to enforce a maximum amount of interactions with a particular device or rate-limit the interactions with a device (by consulting an external timer device before allowing an interaction). Another realistic policy could enable an LED whenever a camera device is set to capture mode (for privacy reasons). More generally, we believe that many interesting policies could be enforced using nested, unsophisticated low-TCB wrappers. Such policies are directly supported in our model in a way that appears quite realistic for embedded processors (e.g. the SAM D5x/E5x [104]), if they were extended with capability machine security primitives. Figure 3.1 shows a possible architecture with a bottom-level wrapper wrapper$_1$ around all peripherals, as well as two nested wrappers (wrapper$_{21}$ and wrapper$_{22}$), each consisting of a read and write closure around different peripherals. On top of these wrappers is the remainder of the code base, which can remain untrusted and largely unmodified, except that direct writes to peripheral MMIO addresses need to be replaced with invocations of the wrappers.

A security analysis of nested wrappers should consider different attacker models, for example the four models depicted in Figure 3.2. In these diagrams, gray components are treated as untrusted: white components are manually verified for correctness and security (i.e. proper encapsulation towards the gray components). For example, a security analysis of wrapper$_1$ could use the leftmost model: only wrapper$_1$ is trusted and all other code on the system is treated as arbitrary. Because of the machine's capability safety, it suffices to manually verify the wrapper and its encapsulation to verify the intended property. A second analysis could use the attacker model of Figure 3.2b: in addition to wrapper$_1$, it relies on the secure wrappers wrapper$_{21}$ and wrapper$_{22}$. This second analysis has a larger TCB but can prove the stronger properties that wrapper$_{21}$ and wrapper$_{22}$ enforce. The analysis does not inspect the code of wrapper$_1$ for this, but relies on a functional contract for it (perhaps provided by the hardware manufacturer together with wrapper$_1$). Further analyses could use additional attacker models as depicted in Figure 3.2c and Figure 3.2d.

scenarios and we believe that nested encapsulation is useful for enforcing correctness, as well as security properties.

Note that while the system is analyzed several times, we hasten to point out that (1) only wrapper code is manually verified and (2) no wrapper is verified twice: wrapper$_1$ is verified manually, resulting in a functional correctness contract, which is then used when manually verifying wrapper$_{21}$ and wrapper$_{22}$. The full system properties are obtained by combining the verification results of each of the wrappers with our general capability safety theorem. Note also that although wrapper$_{21}$ and wrapper$_{22}$ are both trusted in the second attacker model, compositionality of our program logic still allows us to verify them separately, relying on a contract for each other's behavior. This avoids creating unnecessary dependencies when verifying wrappers.

In this paper, we present and explain our approach by considering representative example wrappers on a model capability machine. A first example corresponds roughly to the first two layers of Figure 3.1, with wrappers enforcing a simple bounds check and a maximum bound on the amount of accesses respectively (i.e. a stateful property). In this first example, we demonstrate that we can modularly reason about independent wrappers and that the verification effort can be shared when they are implemented according to a fixed code structure. Our second example shows that our approach also supports more complex properties that consider interactions with several peripherals. It considers a setup where wrapper$_{22}$ is replaced with a wrapper wrapper$_{22}$_bis that limits the rate at which a certain peripheral is accessed. Wrapper$_{22}$_bis will only allow accesses to its peripheral after an external timer device indicates that sufficient time has passed. It does not follow the same fixed structure as the first wrappers and is verified separately.

To summarize, our contributions are the following:

- We extend the capability machine model, program logic and logical relation of Georges et al. [62] with memory-mapped I/O and secure enforcement of I/O properties and reprove their universal contract for arbitrary code on the machine.
- We demonstrate universal contracts for proving full-system security properties on effect traces in the presence of untrusted code, thus obtaining more interesting and realistic end-to-end properties.
- We extend the approach to systems with nested encapsulation by analyzing the same system several times with different attacker models.
- We apply this approach to nested policy enforcement wrappers around peripherals and obtain machine-checked full-system proofs of the different stakeholders' intended properties. Custom separation logic resources and wrapper specifications in so-called HOCAP style allow us to accurately specify the contracts between wrappers and verify them modularly. We demonstrate that verification effort can be shared for wrappers sharing a fixed code structure.

$$
\begin{array}{llll}
a & \in & \mathrm{Addr} & \triangleq\ [0, \mathrm{AddrMax}] \\
p & \in & \mathrm{Perm} & \triangleq\ \textsc{o} \mid \textsc{e} \mid \textsc{ro} \mid \textsc{rx} \mid \textsc{rw} \mid \textsc{rwx} \\
c & \in & \mathrm{Cap} & \triangleq\ \{(p, b, e, a) \mid b, e, a \in \mathrm{Addr}\} \\
w & \in & \mathrm{Word} & \triangleq\ \mathbb{Z} + \mathrm{Cap} \\
r & \in & \mathrm{RegName} & \triangleq\ \mathsf{pc} \mid \mathsf{r}_0 \mid \mathsf{r}_1 \mid \ldots \mid \mathsf{r}_{31} \\
reg & \in & \mathrm{Reg} & \triangleq\ \mathrm{RegName} \longrightarrow \mathrm{Word} \\
m & \in & \mathrm{Mem} & \triangleq\ \mathrm{Addr} \longrightarrow \mathrm{Word}
\end{array}
$$

$$
\begin{array}{llll}
 & & \mathrm{EventTy} & \triangleq\ \mathrm{IOWrite} \mid \mathrm{IORead} \\
e & \in & \mathrm{Event} & \triangleq\ \mathrm{EventTy} \times \mathrm{Addr} \times \mathbb{Z} \\
t & \in & \mathrm{Trace} & \triangleq\ \mathrm{list\ Event} \\
\varphi & \in & \mathrm{ExecConf} & \triangleq\ \mathrm{Reg} \times \mathrm{Mem} \times \mathrm{State} \times \mathrm{Trace} \\
\delta & \in & \mathrm{DoneState} & \triangleq\ \mathsf{Standby} \mid \mathsf{Halted} \mid \mathsf{Failed} \\
\mu & \in & \mathrm{ExecMode} & \triangleq\ \mathsf{SingleStep} \mid \mathsf{Repeat}\ \mu \mid \mathsf{Done}\ \delta \\
\rho & \in & \mathbb{Z} + \mathrm{RegName} &
\end{array}
$$

$$
\begin{array}{lll}
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ \rho \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \\
 & & \texttt{eq}\ r\ \rho\ \rho \mid \texttt{lt}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{restrict}\ r\ \rho \mid \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{isptr}\ r\ r \mid \\
 & & \texttt{getp}\ r\ r \mid \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Figure 3.3: Machine words, machine state and instructions.

All of our proofs have been machine-verified in Coq, using the Iris program logic, and are available online [161].

3.2 A simple capability machine with MMIO support

First, we present the operational semantics of our capability machine. Our work builds upon the work of Georges et al. [62] and, transitively, on that of Skorstengaard et al. [142, 143]. As such, our presentation here is similar to theirs, though simplified to fit our purposes. Specifically, we do not consider so-called local and uninitialized capabilities.

Section 3.2.1 presents the operational semantics for a simple capability machine and Section 3.2.2 explains how we add memory-mapped I/O. The semantics is summarized in Figures 3.3 to 3.6 with additions for memory-mapped I/O in blue.

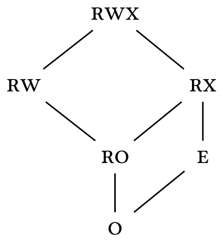


Figure 3.4: Permission lattice.

3.2.1 A simple capability machine

The syntax of capability machine programs is given in Figure 3.3. We consider a machine with finite memory bounded by AddrMax. A machine word $w \in \text{Word}$ is either an unbounded integer or a capability. A capability is a quadruple (p, b, e, a) representing a permission p with authority over range $[b, e)$ and currently pointing to address a. There are six different permissions: opaque (O), enter (E), read-only (RO), read/execute (RX), read/write (RW) and read/write/execute (RWX). These permissions are standard, except for the opaque permission which provides no privilege and the enter permission, inspired by the M-Machine [28], that can be used to build opaque closures or object capabilities. Enter capabilities are "unsealed" into read/execute capabilities when jumped to. The capability is then available in the pc register and can be copied into another register to restore environment variables in the case of a closure.

Figure 3.6 defines the small-step operational semantics of the machine. The machine's state consists of an execution mode μ and an execution configuration φ. The mode μ models the machine's instruction cycle, which loops infinitely (expressed by Repeat μ) until it reaches a successful done state Done Halted through RepeatHalt or a failed state Done Failed through RepeatFail. The RepeatSingle rule allows for the execution of single instructions through the ExecSingle rule. If the execution of the instruction is successful, i.e. execution in ExecSingle does not fail or halt and results in a Done SingleStep state, then RepeatStandby allows for another iteration of the processor's instruction cycle.

An execution step (ExecSingle) requires an executable, in-range capability (p, b, e, a) in the pc register. The word z at address a is then read and decoded into an instruction $decode(z)$ which is executed on the current configuration φ to result in a new machine state $\llbracket decode(z) \rrbracket(\varphi)$. Machine instructions i operate over registers r or either integers or registers ρ. The behavior of instruction i in configuration φ is specified by $\llbracket i \rrbracket(\varphi)$ defined in Figure 3.5. Most instructions use the auxiliary function updPC to increment

$$\mathrm{updPC}(\varphi) = \begin{cases} (\mathrm{Done\ Standby}, \\ \quad \varphi[\mathrm{reg.pc} \mapsto (p, b, e, a+1)]) & \text{if } \varphi.\mathrm{reg}(\mathrm{pc}) = (p, b, e, a) \\ (\mathrm{Done\ Failed}, \varphi) & \text{otherwise} \end{cases}$$

$$\mathrm{getWord}(\varphi, \rho) = \begin{cases} \rho & \text{if } \rho \in \mathbb{Z} \\ \varphi.\mathrm{reg}(\rho) & \text{if } \rho \in \mathrm{RegName} \end{cases} \qquad \begin{aligned} \mathrm{updST}(\varphi, e, s) = \varphi&[\mathrm{state} \mapsto s] \\ &[\mathrm{trace} \mapsto \varphi.\mathrm{trace} +\!\!\!+ [e]] \end{aligned}$$

i	$[\![i]\!](\varphi)$	Conditions
$\texttt{fail}$	$(\mathrm{Done\ Failed}, \varphi)$	
$\texttt{halt}$	$(\mathrm{Done\ Halted}, \varphi)$	
$\texttt{move}\ r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$w = \mathrm{getWord}(\varphi, \rho)$
$\texttt{load}\ r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto w])$	$\varphi.\mathrm{reg}(r_2) = (p, b, e, a)$ and $w = \varphi.\mathrm{mem}(a)$ and $b \leq a < e$ and $p \in \{\textsc{ro}, \textsc{rx}, \textsc{rw}, \textsc{rwx}\}$ and $a \notin \texttt{MMIO}$
$\texttt{load}\ r_1\ r_2$	$\mathrm{updPC}(\mathrm{updST}($ $\varphi[\mathrm{reg}.r_1 \mapsto z],$ $(\mathrm{IORead}, a, z), s))$	$\varphi.\mathrm{reg}(r_2) = (p, b, e, a)$ and $\mathrm{mmioLoad}(\varphi.\mathrm{state}, a) = (s, z)$ and $b \leq a < e$ and $p \in \{\textsc{ro}, \textsc{rx}, \textsc{rw}, \textsc{rwx}\}$ and $a \in \texttt{MMIO}$
$\texttt{store}\ r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{mem}.a \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p, b, e, a)$ and $b \leq a < e$ and $p \in \{\textsc{rw}, \textsc{rwx}\}$ and $w = \mathrm{getWord}(\varphi, \rho)$ and $a \notin \texttt{MMIO}$
$\texttt{store}\ r\ \rho$	$\mathrm{updPC}(\mathrm{updST}($ $\varphi, (\mathrm{IOWrite}, a, z), s))$	$\varphi.\mathrm{reg}(r) = (p, b, e, a)$ and $\mathrm{mmioStore}(\varphi.\mathrm{state}, a, z) = s$ and $b \leq a < e$ and $p \in \{\textsc{rw}, \textsc{rwx}\}$ and $z = \mathrm{getWord}(\varphi, \rho)$ and $z \in \mathbb{Z}$ and $a \in \texttt{MMIO}$
$\texttt{jmp}\ r$	$(\mathrm{Done\ Standby},$ $\varphi[\mathrm{reg.pc} \mapsto \mathit{newPc}])$	if $\varphi.\mathrm{reg}(r) = (\textsc{e}, b, e, a)$ then $\mathit{newPc} = (\textsc{rx}, b, e, a)$ otherwise $\mathit{newPc} = \varphi.\mathrm{reg}(r)$
$\texttt{restrict}\ r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p, b, e, a)$ and $p' = \mathrm{decodePerm}(\mathrm{getWord}(\varphi, \rho))$ and $p' \preccurlyeq p$ and $w = (p', b, e, a)$
$\texttt{subseg}\ r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p, b, e, a)$ and for $i \in \{1, 2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and $b \leq z_1$ and $0 \leq z_2 \leq e$ and $p \neq \textsc{e}$ and $w = (p, z_1, z_2, a)$
$\texttt{lea}\ r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p, b, e, a)$ and $z = \mathrm{getWord}(\varphi, \rho)$ and $p \neq \textsc{e}$ and $w = (p, b, e, a + z)$
$\texttt{geta}\ r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto a])$	$\varphi.\mathrm{reg}(r_2) = (\_, \_, \_, a)$
$\ldots$		
$\_$	$(\mathrm{Done\ Failed}, \varphi)$	otherwise

Figure 3.5: Operational semantics: instruction semantics.

$$\textsc{RepeatSingle}\quad \frac{(\mathsf{SingleStep},\ \varphi) \rightarrow (\mathsf{Done}\ \delta,\ \varphi')}{(\mathsf{Repeat\ SingleStep},\ \varphi) \rightarrow (\mathsf{Repeat}\ (\mathsf{Done}\ \delta),\ \varphi')}$$

$$\textsc{RepeatStandby}\quad (\mathsf{Repeat}\ (\mathsf{Done\ Standby}),\ \varphi) \rightarrow (\mathsf{Repeat\ SingleStep},\ \varphi)$$

$$\textsc{RepeatHalt}\quad (\mathsf{Repeat}\ (\mathsf{Done\ Halted}),\ \varphi) \rightarrow (\mathsf{Done\ Halted},\ \varphi)$$

$$\textsc{RepeatFail}\quad (\mathsf{Repeat}\ (\mathsf{Done\ Failed}),\ \varphi) \rightarrow (\mathsf{Done\ Failed},\ \varphi)$$

$$\textsc{ExecSingle}\quad (\mathsf{SingleStep},\ \varphi) \rightarrow \begin{cases} [\![\mathit{decode}(z)]\!](\varphi) & \text{if } \varphi.\mathrm{reg}(\mathrm{pc}) = (p, b, e, a) \wedge b \leq a < e \wedge \\ & \quad p \in \{\textsc{rx}, \textsc{rwx}\} \wedge \varphi.\mathrm{mem}(a) = z \\ (\mathsf{Done\ Failed},\ \varphi) & \text{otherwise} \end{cases}$$

Figure 3.6: Operational semantics: reduction steps.

the pc register at the end of their executions. Since the address space is finite, pointer arithmetic such as $a + 1$ can fail. For ease of reading, we write $a + k$ to indicate successful pointer arithmetic.

Instructions `fail` and `halt` respectively terminate the execution in a Failed or Halted state. `move` r ρ copies ρ (its value if it's an integer, or its contents if it's a register) into r. Memory can be manipulated using the `load` and `store` instructions: `load` r_1 r_2 reads the value at the address pointed to by the capability in r_2 assuming it has read permission and is within bounds, and copies the value into r_1. Similarly, `store` r ρ stores ρ at the address pointed to by the capability in r assuming it has write permission and is within bounds. The `jmp` instruction jumps to a capability, by writing it into the pc register. As explained earlier, the `jmp` instruction unseals E capabilities into RX capabilities before jumping to them. Capabilities can be modified using the `restrict`, `subseg` and `lea` instructions. `restrict` can be used to *decrease* the permission of a capability according to the permission lattice's partial order $\preccurlyeq$. `subseg` can be used to *decrease* the range of authority of a capability, while `lea` can be used to modify where a capability points to. As E capabilities are used to encapsulate code and data, they cannot be modified until they are unsealed, hence the instructions `subseg` and `lea` fail when used with E capabilities. Indeed, e.g. changing the address of an E-capability could enable Return Oriented Programming (ROP) flavored attacks [138]. Instructions to read capabilities' fields are also provided: `geta`, `getp`, `getb` and `gete` respectively read the a, p, b and e fields of a capability (p, b, e, a). Not shown in Figure 3.5 are instructions `jnz` (conditional jump), `add`, `sub` (addition and subtraction), `eq` (equality), `lt` (comparison) and `isptr` to check whether a register contains a capability. Finally, if none of the above cases apply, the execution falls through to a failed state as shown on the last row in Figure 3.5.

3.2.2 Adding support for memory-mapped I/O

To allow communication between the CPU and devices, we add support for MMIO. Those additions are indicated in blue in Figures 3.3 and 3.5. For simplicity, we do not yet support interrupts; the CPU and devices must poll memory for updates. We discuss adding interrupts in Section 3.6.

To model MMIO, we assume a set `MMIO` of addresses reserved for MMIO and augment execution configurations φ with a trace t of MMIO events e, and an environmental state s drawn from a set State as shown in Figure 3.3. An event e is a triple of a mode e.type (read or write), an address e.addr, and an integer e.value that is read or written. The operational semantics of our machine in Figure 3.5 is parameterized by this set State, `MMIO` and two operations mmioLoad : $(\text{State} \times \text{Addr}) \longrightarrow (\text{State} \times \mathbb{Z})$ and mmioStore : $(\text{State} \times \text{Addr} \times \mathbb{Z}) \longrightarrow \text{State}$, that model how the state of the devices reacts to MMIO loads and stores.

Figure 3.5 shows the new behavior of `load` and `store` for MMIO addresses. `load` will use mmioLoad to get the value at address a from the environment and load it into the register. Similarly, `store` uses mmioStore to indicate that a value is written at some address. In both cases, the operations transition the environment's state and an IORead or IOWrite event is recorded in the trace.

3.3 Example wrappers

This section details the two examples with nested parapass-through security wrappers that were described in the introduction. The goal is to illustrate how stakeholders such as the ones in Figure 3.1 can set up the capability machine to achieve their security objectives.

3.3.1 Three-layer stateful example with orthogonal wrappers

The first system we consider is set up as in Figure 3.1, with an additional bottom-most wrapper wrapper$_0$. There are hence 3 layers of simple wrappers, where the third layer contains two disjointly operating wrappers. The proof effort for this example can be shared, since the wrappers share a common structure, as will be discussed in section 3.4.4. The different wrappers enforce the following concrete invariants, modelling the kind of real security properties we discussed in the introduction:

- Wrapper$_0$ encapsulates all of MMIO, and creates a read and write closure for MMIO that higher-level wrappers use. It does not enforce its proper predicate, i.e. it enforces the trivially true predicate P_0 on the trace t in Figure 3.7. We

$$
\begin{aligned}
&c\_ls_i && \triangleq (\textsc{rwx}, d\_ls_i, d\_e_i, d\_ls_i)\\
&c\_r_i && \triangleq (\textsc{e}, d\_r_i, d\_e_i, d\_r_i)\\
&c\_w_i && \triangleq (\textsc{e}, d\_e_i, d\_e_i, d\_w_i)\\[1em]
&\upharpoonleft^t_e : (\text{Event} \longrightarrow \text{Prop}) \longrightarrow (\text{Trace} \longrightarrow \text{Prop}) && \triangleq \lambda P_{\text{e}}\ t.\,(\forall e.\, e \in t \Rightarrow P_{\text{e}}(e))\\
&\text{F}_{21} : \text{Trace} \longrightarrow \text{Trace} && \triangleq \lambda t.\, \text{filter}(\lambda e.\, e.\text{addr} = a_1, t)\\
&\text{F}_{22} : \text{Trace} \longrightarrow \text{Trace} && \triangleq \lambda t.\, \text{filter}(\lambda e.\, e.\text{addr} = a_2, t)\\
&\overline{\text{F}}_2 : \text{Trace} \longrightarrow \text{Trace} && \triangleq \lambda t.\, \text{filter}(\lambda e.\, e.\text{addr} \notin \{a_1, a_2\}, t)\\[1em]
&\text{P}_0 : \text{Trace} \longrightarrow \text{Prop} && \triangleq \lambda\_.\, \text{True}\\
&\text{P}_1 : \text{Trace} \longrightarrow \text{Prop} && \triangleq \lambda t.\, \text{length}(t) < 1000\\
&\text{P}_{21} : \text{Trace} \longrightarrow \text{Prop} && \triangleq \lambda t.\, \upharpoonleft^t_e(\lambda e, e.\text{value} > 0)\\
&\text{P}_{22} : \text{Trace} \longrightarrow \text{Prop} && \triangleq \lambda t.\, \upharpoonleft^t_e(\lambda e, e.\text{value} < 0)\\[1em]
&\text{LS\_gen}_x : \text{list Word} \longrightarrow \text{list Word} && \triangleq\\
&\qquad\qquad \lambda\overline{v_{\text{cust}}}.\, [c\_ls_x, c\_r_{\text{pr}(x)}, c\_w_{\text{pr}(x)}] +\!\!+ \overline{v_{\text{cust}}} &&\\
&\text{LS}_0 : \text{Trace} \longrightarrow \text{list Word} && \triangleq \lambda\_.\, [(\textsc{rw}, \texttt{MMIO}_b, \texttt{MMIO}_e, \texttt{MMIO}_b)]\\
&\text{LS}_1 : \text{Trace} \longrightarrow \text{list Word} && \triangleq \lambda t.\, \text{LS\_gen}_1([\,\text{length}(t)])\\
&\text{LS}_{21} : \text{Trace} \longrightarrow \text{list Word} && \triangleq \lambda\_.\, \text{LS\_gen}_{21}([\,])\\
&\text{LS}_{22} : \text{Trace} \longrightarrow \text{list Word} && \triangleq \lambda\_.\, \text{LS\_gen}_{22}([\,])
\end{aligned}
$$

Figure 3.7: Definitions involved in the first example.

$$
\textsc{Obj-1}\quad
\frac{\text{init\_config\_1}(r_0, m_0) \qquad (r_0, m_0, \emptyset, s_0) \longrightarrow^* (r, m, t, s)}{\text{P}_1(t)}
$$

$$
\textsc{Obj-21}\quad
\frac{\text{init\_config\_21}(r_0, m_0) \qquad (r_0, m_0, \emptyset, s_0) \longrightarrow^* (r, m, t, s)}{\text{P}_{21}(F_{21}(t)) \wedge \overline{\text{F}}_2(t) = [\,]}
$$

$$
\textsc{Obj-22}\quad
\frac{\text{init\_config\_22}(r_0, m_0) \qquad (r_0, m_0, \emptyset, s_0) \longrightarrow^* (r, m, t, s)}{\text{P}_{22}(F_{22}(t)) \wedge \overline{\text{F}}_2(t) = [\,]}
$$

Figure 3.8: The different security objectives that the stakeholders wish to enforce in the first example

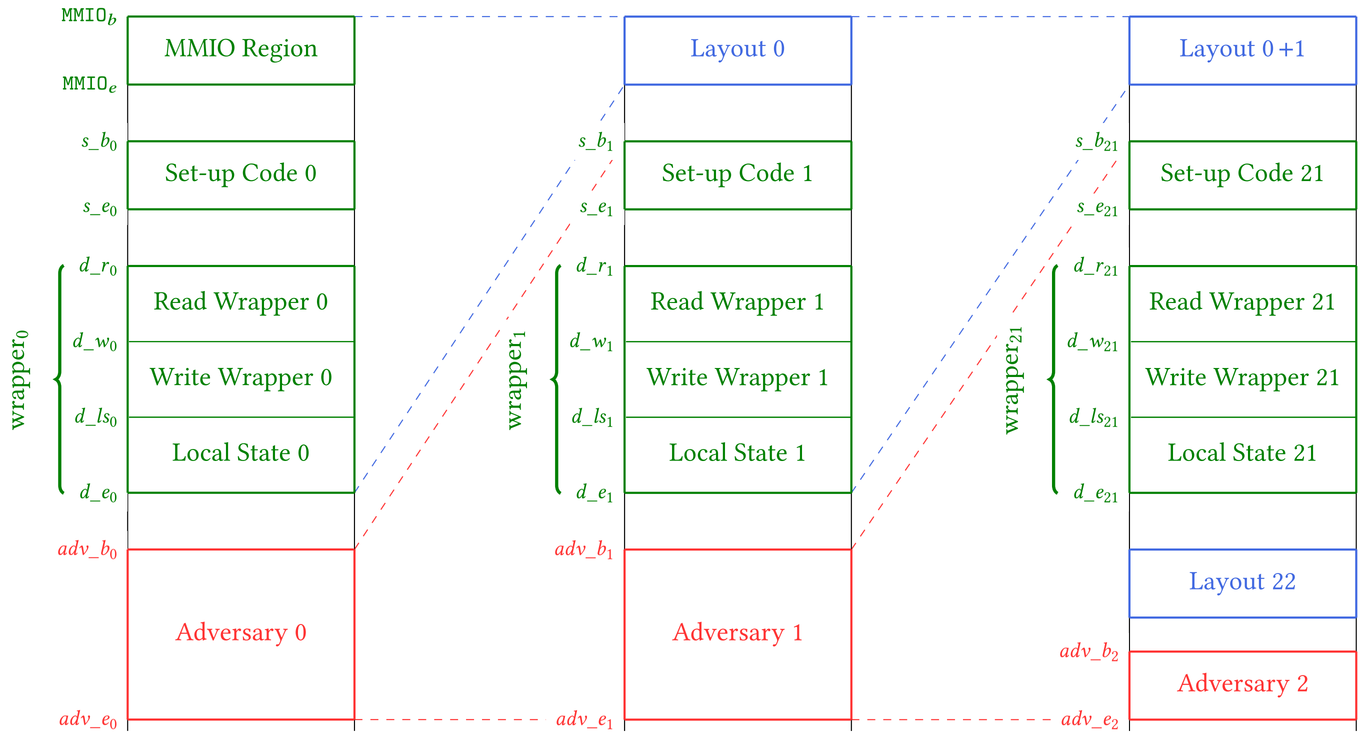
separate this wrapper from the others, since its implementation deviates: it is the only wrapper that does not recursively call another wrapper, but rather accesses MMIO directly.

- Wrapper$_1$ ensures that no more than 1000 MMIO-events (read and write combined) occur once the capability machine boots. This invariant is expressed by the predicate P_1 on the trace t in Figure 3.7.
- To ensure safety of values sent to their respective peripherals, wrapper$_{21}$ solely allows positive values, whereas wrapper$_{22}$ solely allows negative values. Note that it would be trivial to extend this to an arbitrary bounds check. This is expressed by P_{21} and P_{22} in Figure 3.7, where $\upharpoonright^{t}_{e}(P_e)$ is a predicate on traces that holds on a trace t iff $P_e(e)$ holds for all events e in t.
- We model two different peripherals (e.g. network and display) situated at the memory-mapped addresses a_1 and a_2, respectively. Wrapper$_{21}$ only allows events destined for address a_1, i.e., wrapper$_{21}$ enforces predicate P_{21} on a *filtered* view of the general MMIO trace. The filtering is represented by F_{21} in Figure 3.7. Thus wrapper$_{21}$ enforces $\mathrm{P}_{21} \circ \mathrm{F}_{21}$ on the MMIO trace. Analogously, wrapper$_{22}$ enforces $\mathrm{P}_{22} \circ \mathrm{F}_{22}$. The complement filter $\overline{\mathrm{F}}_2$ will be used to prove that no addresses other than a_1 and a_2 receive `MMIO` events, by requiring that $\overline{\mathrm{F}}_2(t) = [\,]$.

The previous description has provided us with sufficient details to define the verification goals of each stakeholder. These goals are formulated as security objectives, and summarized in Figure 3.8. Note that wrapper$_0$ does not have any security objective of its own, since it solely encapsulates MMIO. Obj-1 specifies the guarantees that wrapper$_1$ hopes to achieve from verification. Obj-1 states that if the initial memory m_0 and registers r_0 constitute a valid configuration init_config_1(r_0, m_0) (further explained below), then any execution starting from the empty trace $\emptyset$ and an arbitrary state s_0 and taking an arbitrary number of steps (denoted by $\longrightarrow^*$), will result in a configuration that has a trace t' satisfying P_1. In other words, wrapper$_1$ can be sure that P_1 will hold on any trace of execution, as long as the capability machine boots into a configuration satisfying init_config_1.

For the second layer, we get two separate security objectives; Obj-21 and Obj-22. This models the situation where the two drivers are developed and verified independently (perhaps by separate developer teams in the same company), relying on a contract for the other wrapper, rather than its exact code. Otherwise, Obj-21 and Obj-22 are analogous to Obj-1. They enforce that if the initial memory and register configuration is satisfactory, the respective predicates P_{21} and P_{22} hold over the network and display parts of the final trace t, i.e. $\mathrm{P}_{21}(\mathrm{F}_{21}(t))$ and $\mathrm{P}_{22}(\mathrm{F}_{22}(t))$ hold. Additionally, the complement filter $\overline{\mathrm{F}}_2$ guarantees that no events to addresses other than a_1 or a_2 can ever happen in the system.

The init_config predicate above defines the assumptions each wrapper makes on the initial state of memory and registers to make its security objective provable. It ensures

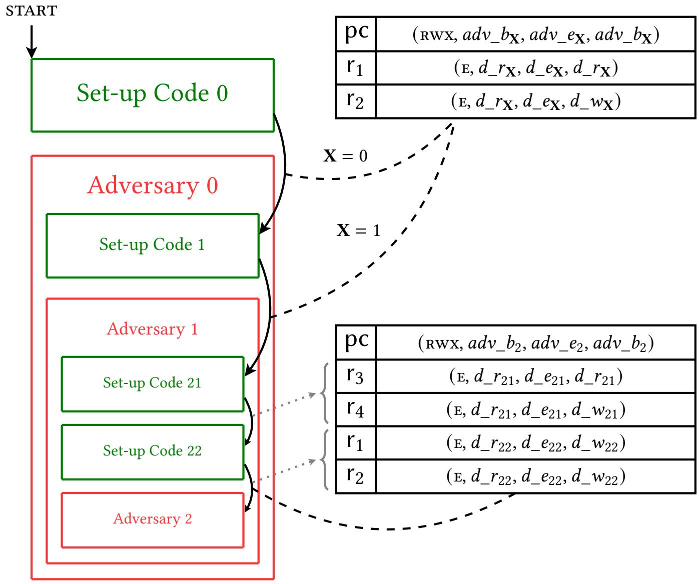


(a) wrapper$_0$ layout

(b) wrapper$_1$ layout

(c) wrapper$_{21}$ layout

Figure 3.9: The memory layout from the point of view of wrappers 0, 1 and 21. Verified parts of memory are shown in green, regions that the wrapper's set-up code assumes correctness contracts for are shown in blue, and adversarial code is shown in red. Red dashed lines illustrate how each following wrapper is situated in the adversarial region of the previous wrapper. Blue dashed lines show how an abstract view of the previous memory layout is assumed in the following wrapper.



pc	$(\textsc{rwx}, adv\_b_{\mathbf{X}}, adv\_e_{\mathbf{X}}, adv\_b_{\mathbf{X}})$
r_1	$(\textsc{e}, d\_r_{\mathbf{X}}, d\_e_{\mathbf{X}}, d\_r_{\mathbf{X}})$
r_2	$(\textsc{e}, d\_r_{\mathbf{X}}, d\_e_{\mathbf{X}}, d\_w_{\mathbf{X}})$

pc	$(\textsc{rwx}, adv\_b_2, adv\_e_2, adv\_b_2)$
r_3	$(\textsc{e}, d\_r_{21}, d\_e_{21}, d\_r_{21})$
r_4	$(\textsc{e}, d\_r_{21}, d\_e_{21}, d\_w_{21})$
r_1	$(\textsc{e}, d\_r_{22}, d\_e_{22}, d\_w_{22})$
r_2	$(\textsc{e}, d\_r_{22}, d\_e_{22}, d\_w_{22})$

Figure 3.10: The flow of control and a few representative register states when jumping to the adversary in the execution of the motivating example. **Black**, dotted connectors indicate register state during the indicated transition. Gray connectors indicate that the pointed-to registers received the indicated value in the previously executed set-up code.

that the pc register is initialized correctly, and disallows the adversary's memory from containing *any* capabilities; a conservative assumption made for simplicity reasons, to avoid a trivial bypass of the encapsulation of trusted components. Additionally, init_config makes assumptions on the initial layout of memory; Figure 3.9 graphically illustrates these assumptions for wrappers 1 and 21 (22 is analogous). The figure also summarizes the different components involved in correctly setting up the wrappers in each layer before passing control to the adversary. Note that these assumptions imply different attacker models for the different security objectives, as sketched earlier in Figure 3.2: adversaries are allowed to arbitrarily instantiate the untrusted parts of the system, depicted in red, with code.

We now discuss each subfigure in order, with the aid of Figure 3.10, which illustrates the control flow of the running example from machine start-up (at START), until control is passed to an adversary. The contents of important registers are shown at key points in execution.

First, Figure 3.9a presents the memory layout from the point of view of wrapper$_0$. As

```
#1: Load MMIO capability
reqint r2 r25
move r25 pc
lea_a r25 d_ls0 r26
load r25 r25
lea_a r25 r2 r26
#2: Write MMIO value
store r25 r1
#3: Clear and return
rclear Raux
jmp r0
```

Figure 3.11: The code for wrapper$_0$'s write closure (Write Wrapper 0 in Figure 3.9a).

for all wrappers in our system, the memory layout contains 3 major parts; code for the wrapper itself, set-up code to initialize the wrapper code, and adversarial code. The MMIO region represents all of `MMIO`, and is unique to wrapper$_0$, since no other wrappers access `MMIO` directly.

The code of wrapper$_0$ itself consists of a read closure from address d_r_0 to d_w_0, a write closure from d_w_0 to d_ls_0 and local state used by the closures, from d_ls_0 to d_e_0. As mentioned before, the read and write closures do not enforce any predicate on the trace; they simply provide read and write functionality to `MMIO` memory. The local state consists of a single address, which the set-up code will store a capability $(\text{RW}, \texttt{MMIO}_b, \texttt{MMIO}_e, \texttt{MMIO}_b)$ for `MMIO` into. It is hence independent of the current trace, and given by $\mathrm{LS}_0(\_)$ in Figure 3.7. This capability provides the read and write closures of wrapper$_0$ access to all of `MMIO`. Figure 3.11 demonstrates the code for the write closure. It makes use of the following macros:

- `reqint` r r_{aux}: succeeds iff r contains an integer
- `lea_a` r ρ r_{aux}: absolute version of `lea`, that makes the capability in r point to the address corresponding to ρ.
- `rclear` $\overline{r}$: clears the set of registers $\overline{r}$ by overwriting them with the value 0.

In order to simplify wrapper invocations, we settled on the following common calling convention in our examples:

- r_0 contains the return address
- r_1 contains the value to write in case of a write event, and the value that is read in case of a read event
- r_2 contains the request's destination MMIO address
- r_{25}-r_{31} are caller-save auxiliary registers, jointly denoted by the set R_{aux}

Figure 3.11 starts by loading the `MMIO` capability from d_ls_0 into r_{25}, and making it point to the MMIO address in r_2. Next, the value in r_1 is stored through r_{25}, thereby writing it to `MMIO` memory. Note that if r_1 is not an integer, or r_2 is not an `MMIO` address, this operation will fail. Finally, all auxiliary registers are cleared, and the write closure jumps to its return address in r_0.

```
#0: Machine boots here
#1: MMIO capability
move r0 pc
lea_a r2 d_ls0 r1
move r1 pc
subseg r1 MMIOb MMIOe
store r2 r1
#2: Wrapper closures
move r1 pc
subseg r1 d_r0 d_e0
move r2 r1
lea_a r1 d_r0 r3
lea_a r2 d_w0 r3
restrict r1 e
restrict r2 e
#3: Adv capability
move r0 pc
subseg r0 adv_b0 adv_e0
lea_a r0 adv_b0 r3
#4: Clear, jump to adv
rclear Rclr*
jmp r0
```

$^{*}R_{\mathrm{clr}} = \mathrm{RegName} \setminus \{\mathrm{pc}, r_0, r_1, r_2\}$

Figure 3.12: The set-up code for wrapper$_0$ (Set-up Code 0 in Figure 3.9a).

The purpose of the set-up code is to create the previously discussed read and write closures, and to pass these to Adversary 0 securely. Figure 3.12 lists the concrete set-up code for wrapper$_0$. When the machine boots, the pc register is assumed to point to Set-up Code 0 and contains an omnipotent capability granting access to all of memory. For simplicity reasons, all wrapper *code* is assumed pre-loaded in memory, but initial memory cannot contain any *capabilities*. The set-up code starts by deriving the previously discussed `MMIO` capability from the omnipotent pc, and storing it at address d_ls_0. Next, it derives the read and write closures from the pc and stores them in r_1 and r_2. Lastly, it restricts the omnipotent pc to the adversary region, clears all auxiliary registers and jumps to the adversary, thereby loading r_0 into the pc. Figure 3.10 demonstrates how execution starts in Set-up Code 0, and how, when jumping to Adversary 0, pc, r_1, and r_2 are set up as previously described. From the point of view of wrapper$_0$, the concrete code stored inside Adversary 0 is irrelevant, as capability safety ensures that no adversary will be able to bypass its read and write closures.

Let us now consider the memory layout for wrapper$_1$ in Figure 3.9b. The layout is similar to Figure 3.9a, except for the topmost region. Since Set-up Code 0 gets to execute before passing control to Set-up Code 1, wrapper$_1$ assumes the existence of a region Layout 0, whose instructions satisfy a contract that captures the behavior of Set-up Code 0. Consequently, init_config_1 in Obj-1 requires the machine to boot inside the Layout 0 region and pass control to Set-up Code 1 at the end, with the register state as specified in Figure 3.10. Requiring a contract rather than precise code makes the different layers more independent, and will, e.g., allow the hardware vendor to optimize network packet handling without affecting any proofs in higher-up layers.

The code for wrapper$_1$ itself is similar in layout, but makes use of more extensive local state than wrapper$_0$. Concretely, the read and write closures ensure that the local state always satisfies LS$_1$ in Figure 3.7. LS$_1$ is defined in terms of a general local state LS_gen$_x$. LS_gen$_x$ describes the layout of local state for wrapper x (with $x \neq 0$). It specifies that the first three addresses contain a rwx capability c_ls_x for all of local

```
#Template code
read_wrapper(check_read) ≜
  is_addr r2 r25 r26
  check_read
  move r25 pc
  lea_a r25 d_ls1 r26
  load r25 r25
  lea r25 1
  load r26 r25
  jmp r26

read_wrapper_1 ≜
  read_wrapper(check_1)

#Check: < 1000 events
check_1 ≜
  move r25 pc
  lea_a r25 d_ls1 r26
  load r25 r25
  lea r25 3
  load r26 r25
  add r26 r26 1
  lt r26 r26 1000
  lea pc r26
  fail
  load r26 r25
  add r26 r26 1
  store r25 r26
```

Figure 3.13: The code for wrapper$_1$'s read closure (Read Wrapper 1 in Figure 3.9b).

state, and the read and write closures $c\_r_{\mathrm{pr}(x)}$ and $c\_w_{\mathrm{pr}(x)}$, where $\mathrm{pr}(x)$ represents the layer-below wrapper, which x will call. For example, $\mathrm{pr}(22) = 1$. Having c_ls_x is required because jumping to an E capability in our capability machine results in a pc with RX permission, which disallows updating the local state. Lastly, $\mathrm{LS\_gen}_x$ takes a list of custom values $\overline{v_{\mathrm{cust}}}$ as an argument. This allow wrappers to specify additional local state to aid in enforcing their invariants. In this case, Figure 3.7 defines $\mathrm{LS}_1 \triangleq \lambda t.\,\mathrm{LS\_gen}_1([\,\mathrm{length}(t)])$, where $x = 1$ because wrapper$_1$ requires access to its own local state, and $\mathrm{pr}(x) = 0$ because wrapper$_1$ will call wrapper$_0$ to have it perform MMIO. Additionally, $\overline{v_{\mathrm{cust}}} = [\,\mathrm{length}(t)]$, because wrapper$_1$ requires one extra address to store local state; a counter $\mathrm{length}(t)$ that corresponds to the number of MMIO events performed so far, to compare it to 1000.

Figure 3.13 demonstrates how the read closure of wrapper$_1$ uses local state. It makes use of a generic `read_wrapper` template, which we use for any non-bottom-level wrapper in this example. The template uses a list of instructions `check_read` to check whether the wrapper's predicate (e.g., P_1 in this case) would still hold after the next MMIO event, and update the local state if this is the case. First, the template verifies whether r_2 is a valid address, and not just any integer. This is done using the `is_addr` r r_{aux1} r_{aux2} macro, which succeeds iff r contains an integer that corresponds to a memory address. Then, it calls upon the checking instructions. Lastly, it loads c_r_0 from address $d\_ls_1 + 1$ and jumps to it. For wrapper$_1$ the template is instantiated with `check_1`, which checks that P_1 holds. These instructions first load c_ls_1 from d_ls_1, then load $\mathrm{length}(t)$ from $d\_ls_1 + 3$, check whether $\mathrm{length}(t) + 1 < 1000$ still holds, and `fail` if this is not the case. If the check passes, $\mathrm{length}(t) + 1$ is stored to $d\_ls_0 + 3$, ensuring that LS_1 holds again after the call to wrapper$_0$.

The set-up code for wrapper$_1$ is very similar to the code we discussed in Figure 3.12. The only difference is that, rather than ensuring that $\mathrm{LS}_0([\,])$ holds, the set-up code needs to satisfy $\mathrm{LS}_1([\,])$ before jumping to Adversary 1. Figure 3.10 again shows the state of key registers at that point.

Finally, Figure 3.9c summarizes wrapper$_{21}$. Wrapper$_{21}$ again assumes a contract for lower-level wrappers, that is satisfied by the region Layout 0 +1. This contract captures the behavior of Set-up Code 0 and 1 combined, i.e., the first two steps in Figure 3.10. Similarly, a new region Layout 22 satisfies a second contract that captures the behavior of Set-up Code 22. Both contracts appear in the definition of init_config_21 in Obj-21 and enable more modular code development. We do not discuss wrapper$_{22}$ separately, since its layout is the dual of wrapper$_{21}$. Both wrappers share the same adversary.

The code for wrapper$_{21}$ itself is similar to the code for wrapper$_1$. It enforces LS_{21} in Figure 3.7, which is again defined in terms of $\mathrm{LS\_gen}_x$. No custom local state is needed to check P_{21}. The set-up code is also similar, but it jumps to wrapper$_{22}$ before control is passed to Adversary 2, as shown in Figure 3.10. This Figure demonstrates how Set-up Code 21 sets up the closures for wrapper$_{21}$ in r_3 and r_4, whereas Set-up Code 22 (or in this case the assumed contract for Layout 22) sets up its closures in r_1 and r_2. Set-up Code 21 uses r_1 and r_2 to pass the closures for wrapper$_1$ to Set-up Code 22.

3.3.2 Rate limiting

To demonstrate how the previous example generalizes to support a more complex property that requires interaction with multiple devices, we implemented and verified a second example where wrapper$_{22}$ is replaced by wrapper$_{22}$_bis; a wrapper that implements rate limiting. The other wrappers and security objectives remain unchanged. Concretely, wrapper$_{22}$_bis relies on a trusted, memory-mapped timer device, and only allows an IO-event to or from its peripheral (at the same address a_2 that wrapper$_{22}$ used) to occur when a value of 1 has been read from the timer address a_{timer} beforehand. In other words, wrapper$_{22}$_bis enforces the predicate P_{22}_bis in Figure 3.14 on a version of the MMIO trace that has been filtered through F_{22}_bis. Here, the function last returns the most recent event in t, if any. In summary, security objective Obj-22 is replaced by the following (where $\overline{\mathrm{F}}_2$ now also disallows events to a_{timer}):

$$\textsc{Obj-22-bis}\quad \frac{\mathrm{init\_config\_22}(r_0, m_0) \qquad (r_0, m_0, \emptyset, s_0) \longrightarrow^* (r, m, t, s)}{\mathrm{P}_{22}\mathrm{\_bis}(\mathrm{F}_{22}\mathrm{\_bis}(t)) \land \overline{\mathrm{F}}_2(t) = [\,]}$$

To correctly enforce $\mathrm{P}_{22}\mathrm{\_bis} \circ \mathrm{F}_{22}\mathrm{\_bis}$, wrapper$_{22}$_bis consists of 2 different types of closures. First, a read and write closure similar to the ones demonstrated in Figure 3.9, which clients can use to respectively read from and write to a_2 if the last event in

$$\begin{array}{l}
\mathrm{P}_{22}\text{\_bis} : \mathrm{Trace} \longrightarrow \mathrm{Prop} \triangleq \\
\quad \lambda t.\, \texttt{match}\ t\ \texttt{with} \\
\quad \mid [\,] : \mathrm{True} \\
\quad \mid t' \mathbin{+\!\!+} [e] : \mathrm{P}_{22}\text{\_bis}(t') \land \\
\quad\; (e.\mathrm{addr} \neq a_{\mathrm{timer}} \Rightarrow \mathrm{last}(t') = \mathrm{Some}(\mathrm{IORead}, a_{\mathrm{timer}}, 1)) \\
\mathrm{F}_{22}\text{\_bis} : \mathrm{Trace} \longrightarrow \mathrm{Trace} \triangleq \\
\quad \lambda t.\, \mathrm{filter}(\lambda e.\, e.\mathrm{addr} = a_2 \lor e.\mathrm{addr} = a_{\mathrm{timer}}, t) \\
\mathrm{LS}_{22}\text{\_bis} : \mathrm{Trace} \longrightarrow \mathrm{list\ Word} \triangleq \\
\quad \lambda t.\, \mathrm{LS\_gen}_{22}([\texttt{if}\ \mathrm{last}(t) = \mathrm{Some}(\mathrm{IORead}, a_{\mathrm{timer}}, 1) \\
\qquad \texttt{then}\ 1\ \texttt{else}\ 0)])
\end{array}$$

Figure 3.14: Definitions involved in the second example.

F_{22}_bis(t) is a timer event that returned 1. Second, a read-only timer closure, which allows reading from a_{timer} and returns the read value. To coordinate between these closures, they share local state that satisfies LS_{22}_bis in Figure 3.14. The three closures uphold LS_{22}_bis as follows:

- Whenever the timer closure is called, it writes a 1 to the local state if it read a 1, and 0 otherwise.
- Whenever the regular read or write closure is called, if the event is destined for address a_2, it checks the local state to see if the stored value is 1. If not, the wrapper fails. If the value is 1, it is consumed and set to 0, and the call is passed on to $\mathrm{wrapper}_1$.

Note that in our current set-up, $\mathrm{wrapper}_{22}$_bis is the sole wrapper that can read (and write) the timer address, since it requires a view of *all* MMIO events to a_{timer} and a_2 in F_{22}_bis. Section 3.4.1 will discuss how our approach can be generalized to a setting where a closure is shared between multiple wrappers, e.g. $\mathrm{wrapper}_{21}$ and $\mathrm{wrapper}_{22}$_bis.

3.4 Proving The Security Objectives

This section outlines the high-level technical ideas that underlie the proofs of security objectives such as the ones in Figure 3.8. Intuitively, the proof of each wrapper's security objective employs an invariant to state that the objective holds at each step of execution. Section 3.4.1 discusses how each wrapper's invariants are formalized in Iris.

$$\begin{array}{lcl} \text{valid}(F) & \triangleq & \exists P_e : \text{Event} \to \text{Prop}.\, (\forall t.\, F(t) = \text{filter}(P_e, t)) \land \\ & & \qquad (\forall e : \text{Event}.\, \text{decidable}(P_e(e))) \\ \text{orthogonal}(F_1, F_2) & \triangleq & \forall t.\, F_1(F_2(t)) = F_2(F_1(t)) = [\,] \end{array}$$

$$\text{①}\ (\text{filter\_full}\ \gamma\ t * \text{filter\_val}\ \gamma\ F\ t') \twoheadrightarrow F(t) = t'$$

$$\text{②}\ (\text{filter\_val}\ \gamma\ F\ t * \text{filter\_val}\ \gamma\ F'\ t') \twoheadrightarrow \text{orthogonal}(F, F')$$

$$\text{③}\ \begin{array}{l} (\forall F'.\, \text{valid}(F') \land \text{orthogonal}(F, F') \to F'(t) = F'(t')) \to \\ (\text{filter\_full}\ \gamma\ t * \text{filter\_val}\ \gamma\ F\ \_\,) \twoheadrightarrow (\text{filter\_full}\ \gamma\ t' * \text{filter\_val}\ \gamma\ F\ F(t')) \end{array}$$

Figure 3.15: Three main properties of the filter_full and filter_val abstractions built on top of the filter resource algebra in Iris, and definition of the auxiliary notions of validity and orthogonality they require.

Proving that the invariant always holds requires reasoning about 2 different phases of execution:

1. The wrapper's concrete closures that we hand-verify should enforce the invariant, as Section 3.4.2 further explains. Additionally, the concrete Set-up Code of the wrapper and (specifications for) the set-up code of all layer-below wrappers should respect the invariant. The latter is simple to prove, since set-up code does not perform IO itself.
2. The arbitrary adversarial code in the Adversary region should be *safe* to execute, i.e. have no way of bypassing the wrapper's closures and breaking the invariant. Section 3.4.3 discusses how we employ a semantic model to reason about the safety of unknown code in the capability machine.

In Section 3.4.4 we finally discuss an approach to sharing the verification effort for wrappers that have a common structure. This is not required for our verification approach, but it allowed us to reduce the verification effort involved in proving the first example.

3.4.1 Invariants to enforce security objectives

In this section, we detail how each wrapper x employs invariants *invs*(x) to prove *modularly* that its security objective Obj-**X** holds at each step of execution. It is insufficient to have an invariant that simply states that the security objective holds continuously. To be of general use, each wrapper requires three additional types of reasoning to be possible with its invariant.

First, each wrapper has a *view* of the physical trace, which is the part of the physical trace that it knows about. In our first example, wrappers 0 and 1 view the entire trace, whereas the views of wrappers 21 and 22 consist of all events addressed to a_1 and a_2, respectively. A wrapper x will only ever see a subview of the layer-below wrapper $\mathrm{pr}(x)$'s view. By expressing the view of x in terms of the view of $\mathrm{pr}(x)$, the invariant ensures that all wrappers' views are indeed recursively views of the actual physical event trace, obtained through repeated filtering.

Secondly, in case multiple wrappers $x_1, \ldots, x_n$ have a common layer-below wrapper $\mathrm{pr}(x_1)$, it should be possible to modularly reason about events they admit. For example, we should not have to know anything about the view of x_n on the trace, to reason about the view that x_1 has on the trace.

Lastly, the invariant needs to ensure that any local state that the wrapper requires for its correct operation is enforced on the wrapper's *current* view of the trace. For example, to prove correct operation of wrapper$_1$ in our first example, the invariant must be able to guarantee that the number of MMIO events that wrapper$_1$ keeps track of, is the number of events in the most up-to-date view of the trace.

In the remainder of this section, we flesh out these three types of reasoning by means of Figure 3.16, which demonstrates how different aspects of the wrappers' invariants help us achieve the desired reasoning. At the end, we showcase some concrete invariants used in our first example. Note that the enforcement of the security objective itself is trivially expressed by adding the condition $P_x(t_x)$ to the invariant, as Figure 3.16 shows.

Connecting x_1 to $\mathrm{pr}(x_1)$

In general, a wrapper x_1's view of the trace is a filtered view, a subsequence of the layer-below wrapper $\mathrm{pr}(x_1)$'s view. We can connect different layers this way: the invariant for wrapper x_1 in Figure 3.16 owns a resource filter_val $\gamma_{\mathrm{pr}(x_1)}\ F_{x_1}\ t_{x_1}$, which states that x_1's view on the trace is t_{x_1} and that t_{x_1} is obtained by applying a filter F_{x_1} to the view $t_{\mathrm{pr}(x_1)}$ that $\mathrm{pr}(x_1)$ has. As shown in the bottom of the figure, this resource connects to the resource filter_full $\gamma_{\mathrm{pr}(x_1)}\ t_{\mathrm{pr}(x_1)}$ in wrapper $\mathrm{pr}(x_1)$, which represents the full view. Here, $\gamma_{\mathrm{pr}(x)}$ is simply a name used to distinguish different filtering systems. The root wrapper has a view on the entire physical trace, i.e. its invariant owns a resource filter_val $\gamma\ \mathit{id}\ t$ where id is the identity filter and t is the physical trace. Note that top-level wrappers do not require a resource filter_full $\gamma\ t$, since they do not provide a view on the trace to a higher-level wrapper. When proving Obj-**X** for a wrapper x_1, x_1 and its siblings are the top level wrappers.

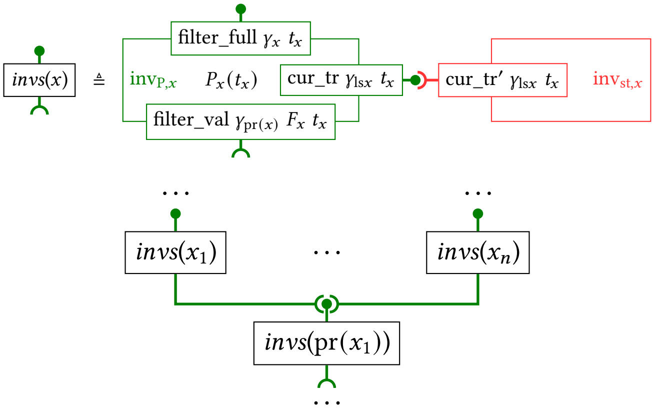

$invs(x)$
≜
filter_full γ_x t_x
$\mathrm{inv}_{\mathrm{P},x}$
$P_x(t_x)$
cur_tr γ_{lsx} t_x
filter_val $\gamma_{\mathrm{pr}(x)}$ F_x t_x
cur_tr′ γ_{lsx} t_x
$\mathrm{inv}_{\mathrm{st},x}$
. . .
. . .
$invs(x_1)$
. . .
$invs(x_n)$
$invs(\mathrm{pr}(x_1))$
. . .

Figure 3.16: Figure illustrating the interaction of the different resources in the invariants $invs(x)$ (consisting of two invariants $\mathrm{inv}_{\mathrm{P},x}$ and $\mathrm{inv}_{\mathrm{st},x}$) of an arbitrary wrapper x. The bottom of the figure shows how the invariants of a wrapper $\mathrm{pr}(x_1)$ are connected to its layer-above wrappers $x_1, \ldots, x_n$. Circular connectors -● represent an authoritative (i.e. "full") view of a resource, that one or more fragmentary (i.e. "partial") views, represented by claw-shaped connectors -<, can be connected to. The connector -< represents a *unique* partial view, i.e. it is enforced to be equal to the full view.

Reasoning modularly about sibling wrappers

In the case where multiple wrappers $x_1, \ldots, x_n$ have a common layer-below wrapper $\mathrm{pr}(x_1)$, reasoning about each wrapper's view can happen modularly if the wrappers have *orthogonal* views on the trace, where orthogonality for two filters F and F' is denoted $\mathrm{orthogonal}(F, F')$ and defined in Figure 3.15. For example, F_{21} and F_{22} are orthogonal in our first example, so we can update the view of the trace $F_{21}(t)$ that wrapper$_{21}$ has, without requiring any knowledge of wrapper$_{22}$'s view $F_{22}(t)$, and vice versa. Similarly, we might have multiple wrappers that only write certain ranges of output values, that only ever read, respectively write values, that always write specific pairs of MMIO values, etc. Note that orthogonality is more flexible than disjointness since it does not presuppose a notion of intersection, but that it degenerates to the latter in case we consider filters that filter on individual events (as is the case in our examples).

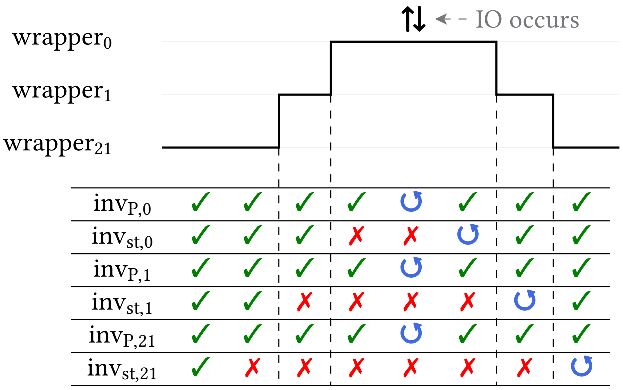
Given this notion of orthogonality, we defined a novel *filter* resource algebra (an Iris construct used to define custom separation logic resources) to reason about independent, orthogonal updates to a trace. The previous resources filter_full γ t

and filter_val γ F t are in fact defined in terms of this resource algebra. Figure 3.15 defines the three most important properties that these two resources satisfy. First, we define a filter F to be *valid* if it filters the trace by a decidable event predicate P_e. Property 1 states that t' is indeed a view of t through the filter F. Property 2 states that any two filter_val resources are guaranteed to be orthogonal. Property 3 then leverages orthogonality to express how we may update traces modularly: if all orthogonal, valid filters are unaffected by a trace update, then we can update a filter_full and filter_val *without requiring ownership of any other filters.* Figure 3.16 illustrates how the filters $x_1, \ldots, x_n$ are connected to their common layer-below filter, assuming orthogonal filtering predicates.

The reader might wonder about the case where the views of $x_1, \ldots, x_n$ are *not* orthogonal. This case does not occur in our current examples, but Iris offers multiple resource algebras that can be used to reason about different forms of shared views on the trace. For example, imagine a scenario where a clock closure (similar to the timer closure in our second example) is shared between wrapper$_{21}$ and wrapper$_{22}$_bis, such that both wrappers can read timestamps from it and enforce their own predicates (e.g. “at least X seconds have to pass between any two writes to a_2”). The wrappers 21 and 22_bis would still require an orthogonal view to know about *all* events at respectively a_1 and a_2, but now also a *partial view* for their reads from a_{clock}. The partial view ensures that wrapper$_{22}$_bis does not need to update its view whenever wrapper$_{21}$ reads a timestamp from the clock, and vice versa. Iris already contains a *monotone* resource algebra, that could allow implementing such partial views.

Incorporating local state

While the security objective $P_x(t_x)$ has to be satisfied at each step of execution (i.e. atomically), the local state can temporarily be out of sync with the trace, e.g. if a wrapper updates its local state before invoking the layer-below wrapper to perform IO, or if updating the local state takes multiple instructions. For this reason, $\mathit{invs}(x)$ consists of two parts: an atomic invariant $\text{inv}_{\text{P},x}$, which ensures that the security objective holds continuously, and a so-called *non-atomic* invariant [69] $\text{inv}_{\text{st},x}$, which ensures (among other things) that the local state is satisfied. The resources cur_tr $\gamma_{\text{ls}x}$ t_x and cur_tr′ $\gamma_{\text{ls}x}$ t_x in Figure 3.16 enforce that the view of the trace in both invariants is the same. Figure 3.17 illustrates how both types of invariants are upheld differently when invoking one of wrapper$_{21}$'s closures: the local state invariants can temporarily be broken while a lower-level wrapper is executing and reestablished afterwards, whereas all atomic invariants have to be updated at the same time, during the instruction that performs the physical MMIO effect.



$\mathrm{inv}_{\mathrm{P},0}$	✓	✓	✓	✓	↺	✓	✓	✓
$\mathrm{inv}_{\mathrm{st},0}$	✓	✓	✓	✗	✗	↺	✓	✓
$\mathrm{inv}_{\mathrm{P},1}$	✓	✓	✓	✓	↺	✓	✓	✓
$\mathrm{inv}_{\mathrm{st},1}$	✓	✓	✗	✗	✗	✗	↺	✓
$\mathrm{inv}_{\mathrm{P},21}$	✓	✓	✓	✓	↺	✓	✓	✓
$\mathrm{inv}_{\mathrm{st},21}$	✓	✗	✗	✗	✗	✗	✗	↺

Figure 3.17: Overview of when different invariants need to hold when invoking wrapper$_{21}$ in our first example, where ↺ denotes an invariant being reestablished for a new view of the trace.

Putting it all together

Given the representation of $\mathrm{inv}_{\mathrm{P},x}$ in Figure 3.16, we now denote this atomic invariant with all of its parameters as $\mathrm{inv}_{\mathrm{P}}(F_x, P_x, tp_x, \gamma_{\mathrm{pr}(x)}, \gamma_{\mathrm{ls}x}, \gamma_x)$. The only new parameter is the boolean tp_x, which denotes whether x is currently considered a top-level wrapper. This is important to know since, as mentioned, the resource filter_full γ_x t_x is not present if x is a top-level wrapper. The arguments $\gamma_{\mathrm{pr}(x)}$,$\gamma_{\mathrm{ls}x}$,γ_x are omitted if they are clear from context.

To exemplify the previous discussion, we investigate the atomic invariants involved in proving Obj-21 (the non-atomic invariants are more tedious and less interesting). The invariants are as follows:

$$\begin{gathered}\mathrm{inv}_{\mathrm{P}}(\mathrm{id}, \_, \mathrm{False}, \gamma_0, \gamma_{\mathrm{ls0}}, \gamma_1) * \mathrm{inv}_{\mathrm{P}}(\mathrm{id}, \_, \mathrm{False}, \gamma_1, \gamma_{\mathrm{ls1}}, \gamma_{21}) \\ * \ \mathrm{inv}_{\mathrm{P}}(\mathrm{F}_{21}, \mathrm{P}_{21}, \mathrm{True}, \gamma_{21}, \gamma_{\mathrm{ls21}}, \_) \\ * \ \boxed{\exists t.\ \mathrm{filter\_val}\ \gamma_2\ \overline{\mathrm{F}}_2\ [\,]}\end{gathered}$$

The first three invariants represent wrappers 0, 1 and 21. Since wrappers 0 and 1 are the only wrappers in their layer, they apply the identity filter id to the previous layer's trace. The resource filter_val γ_0 id t in wrapper$_0$'s invariant is linked to the actual physical trace. The third invariant, wrapper$_{21}$'s proper invariant, enforces the first conclusion of Obj-21, through the presence of F$_{21}$ and P$_{21}$. Additionally, the fourth invariant ensures that no other addresses than a_1 and a_2 receive `MMIO` events. From the point of view of the top-level wrapper$_{21}$ in Figure 3.9c, the third layer is irrelevant, so the third invariant has tp set to True, whereas non-top-level wrappers 0 and 1 have it set to False. Also note the _ in place of P_0 and P_1 in the first two invariants, indicating that $\mathrm{wrapper}_{21}$ does not care what predicate the lower wrappers enforce when proving its own security objective.

3.4.2 Functional correctness of wrappers

The invariants discussed in the previous section are used to prove two contracts for every wrapper: a form of functional correctness and a form of security. We will discuss the proofs of security in more detail in the next section, and the proofs of correctness now. The correctness contract expresses that when a wrapper is invoked, it either generates the desired external effect or throws an error in case the effect would violate their policy.

These contracts are expressed in terms of a program logic for our capability machine, which we inherit from Georges et al. [62] and extend with rules for the MMIO cases of the `load` and `store` instructions. The program logic contains the following weakest precondition assertion:

$$\mathsf{wp}\ \mathsf{Repeat}\ \mathsf{SingleStep}\ \{s.\, Q(s)\}$$

which is read as "repeating the fetch decode execute loop of the capability machine until it either halts or fails, will produce a final state s (Done Failed or Done Halted) for which Q holds". In terms of wp {}, we can define a form of contract triple $\{P\}$ Repeat SingleStep $\{Q\}$ that we define (roughly) as follows[2]:

$$\begin{aligned}&\forall \varphi.\, P \twoheadrightarrow (Q \twoheadrightarrow \mathsf{wp}\ \mathsf{Repeat}\ \mathsf{SingleStep}\ \{s.\, \varphi(s)\})\\&\quad \twoheadrightarrow \mathsf{wp}\ \mathsf{Repeat}\ \mathsf{SingleStep}\ \{s.\, \varphi(s)\}\end{aligned}$$

In terms of this abstraction, the functional correctness contract of the write closure of a wrapper x could look roughly as follows (omitting error cases and technical

[2] Two caveats: (1) we do not use this abstraction in our Coq development but use the unfolded definition directly and (2) most of our contracts use a variant of this definition that allows the program to fail at an arbitrary point of execution.

details):

$$\left\{\begin{array}{l}\text{filter\_val}\ \gamma_x\ (\lambda t.\, \text{filter}(P_e, t))\ t * P_e(\text{IOWrite}, a, v) \\ * \ \mathsf{pc} \mapsto (p_w, b_w, e_w, a_w) \\ * \ \mathsf{r}_0 \mapsto w_{\text{ret}} * \mathsf{r}_1 \mapsto v * \mathsf{r}_2 \mapsto a\end{array}\right\}$$

$$\text{Repeat SingleStep}$$

$$\left\{\begin{array}{l}\text{filter\_val}\ \gamma_x\ (\lambda t.\, \text{filter}(P_e, t))\ (t +\!\!+ [(\text{IOWrite}, a, v)]) \\ * \ \mathsf{pc} \mapsto \mathsf{updatePcPerm}(w_{\text{ret}}) \\ * \ \mathsf{r}_0 \mapsto w_{\text{ret}} * \mathsf{r}_1 \mapsto v * \mathsf{r}_2 \mapsto a\end{array}\right\}$$

where the updatePcPerm function maps E capabilities to their RX counterparts and leaves other capabilities untouched, and we used the fact that filters in our current examples are event-based (cfr. the definition of validity in Figure 3.15) to rewrite F_x as $(\lambda t.\, \text{filter}(P_e, t))$. This contract expresses that when the write closure is invoked and the pc contains a capability (p_w, b_w, e_w, a_w), with arguments v and a and a return capability w_{ret} in appropriate registers, then execution will jump back to w_{ret} with unmodified register contents. Additionally, a filter_val γ_x $(\lambda t.\, \text{filter}(P_e, t))$ t resource is required to update the filter_full γ_x t resource in $\text{inv}_{\text{P},x}$ using property 3 in Figure 3.15. To prove the orthogonality precondition to this property, P_e needs to accept the requested external effect $(\text{IOWrite}, a, v)$. In the postcondition, an updated filter_val resource is returned to express that the effect has been performed.

Unfortunately, the above contract does not quite work. The problem is that higher-level wrappers who wish to invoke a lower-level write closure do not directly own the necessary resource filter_val γ_x $(\lambda t.\, \text{filter}(P_e, t))$ t. As we discussed, this resource is embedded in an invariant inv_P. Because this invariant is atomic, it must be restored after every individual instruction, as was already illustrated in Figure 3.17. It is therefore not possible to extract the resource from the invariant for the duration of the wrapper invocation.

To solve this problem, we employ the technique of *Higher-Order Concurrent Abstract Predicates* (*HOCAP*) [70, 151]. The client wrapper will not extract the filter_val $\gamma_{\text{pr}(x)}$ $(\lambda t.\, \text{filter}(P_e, t))$ t resource from its invariant and restore it after the invocation. Instead, it delegates the work of updating its invariant to the invoked wrapper, and the invoked wrapper will do this at exactly the right execution step, namely the step that executes the MMIO write. This means the contract will look as follows (again omitting error cases and technical details):

$$\left\{\begin{array}{l}\left(\begin{array}{l}\forall t.\, \text{filter\_full}\ \gamma_x\ t * P \Rrightarrow \\ \quad \text{filter\_full}\ \gamma_x\ (t \mathbin{+\!\!+} [(IOWrite, a, v)]) * Q\end{array}\right) \\ * P * \mathsf{pc} \mapsto (p_w, b_w, e_w, a_w) \\ * \mathsf{r}_0 \mapsto w_{ret} * \mathsf{r}_1 \mapsto v * \mathsf{r}_2 \mapsto a\end{array}\right\}$$

$$\textsf{Repeat SingleStep}$$

$$\left\{\begin{array}{l}Q * \mathsf{pc} \mapsto \mathsf{updatePcPerm}(w_{ret}) \\ * \mathsf{r}_0 \mapsto w_{ret} * \mathsf{r}_1 \mapsto v * \mathsf{r}_2 \mapsto a\end{array}\right\}$$

In this contract, the caller provides a so-called *view shift* to the wrapper invocation: a type of logical callback that expresses how the lower-level wrapper $\text{pr}(x)$ can update the client wrapper x's invariant for them. View shifts are the reason why, in Figure 3.17, wrapper$_0$ was capable of immediately reestablishing the atomic invariants of wrapper$_1$ and wrapper$_{21}$ when it performed MMIO. Note that the client does not need to provide their filter_val γ_x $(\lambda t.\, \text{filter}(P_e, t))$ t resource beforehand, but instead is allowed to rely on the invoked wrapper's filter_full γ_x t resource in the proof of the view shift. In this proof, the view shift is allowed to consume additional client resources P that the caller has to provide at the start of the invocation, and produces resources Q. Note that, since top level wrappers contain no filter_full γ_x t resource that needs to be updated externally, their contracts need not be parameterized by a view shift. In other words, the same boolean tp we discussed in the previous section will determine whether a wrapper's contract is parameterized by a view shift.

If x is neither a top-layer nor a bottom-layer wrapper, then both x and $\text{pr}(x)$ are parameterized by a (different) view shift, and a conversion between both needs to happen to prove the contract of x. Using the invariant $\text{inv}_{\text{P},x}$, we can prove the following generally applicable *view shift lowering* lemma, to abstract away most reasoning related to view shifts when proving contracts:

Theorem 3.1 (View Shift Lowering)**.**

$$\begin{array}{l}\text{inv}_{\text{P}}((\lambda t.\, \text{filter}(P_e, t)), P_x, tp_x, \gamma_{\text{pr}(x)}, \gamma_{\text{ls}x}, \gamma_x) \mathrel{-\!\!*} \\ (\forall t.\, \text{filter\_full}\ \gamma_x\ t * P \Rrightarrow \text{filter\_full}\ \gamma_x\ (t \mathbin{+\!\!+} [e]) * Q) \mathrel{-\!\!*} \\ (\forall t.\, \text{filter\_full}\ \gamma_{\text{pr}(x)}\ t * P * P_e(e) * P_x(t \mathbin{+\!\!+} [e]) * \\ \quad \text{cur\_tr}'\ \gamma_{\text{ls},x}\ t \Rrightarrow \text{filter\_full}\ \gamma_{\text{pr}(x)}\ (t \mathbin{+\!\!+} [e]) * Q * \\ \quad \text{cur\_tr}'\ \gamma_{\text{ls},x}\ (t \mathbin{+\!\!+} [e]))\end{array}$$

This theorem states that the view shift that x is parameterized by can be lowered to one that satisfies the contract for $\text{pr}(x)$, if x can provide the additional guarantee that $P_e(x)$ and $P_x(t \mathbin{+\!\!+} [e])$ hold (which is precisely what x checks before admitting

$$
\begin{aligned}
\boxed{\mathcal{E}(v)} &\triangleq \forall reg.\, \big(\mathcal{R}(reg) * \mathsf{pc} \mapsto v * \mathop{\Large\ast}_{(r,w)\in reg, r\neq \mathsf{pc}} r \mapsto w\big) \\
&\qquad\qquad \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{Repeat\ SingleStep}\ \{\_.\, \top\} \\
\boxed{\mathcal{R}(reg)} &\triangleq \mathop{\Large\ast}_{(r,w)\in reg, r\neq \mathsf{pc}} \mathcal{V}(w) \\
\boxed{\mathcal{V}(w)} &\begin{cases}
\mathcal{V}(z), \mathcal{V}(\textsc{o}, -) & \triangleq \top \\
\mathcal{V}(\textsc{e}, b, e, a) & \triangleq \Box \triangleright \mathcal{E}(\textsc{rx}, b, e, a) \\
\mathcal{V}(p, b, e, a) & \triangleq \mathop{\Large\ast}_{a' \in [b,e)} \exists p'.\, p \preccurlyeq p' \wedge \\
& \qquad\qquad \boxed{\exists w.\, a' \mapsto_{p'} w * \mathcal{V}(w)}
\end{cases}
\end{aligned}
$$

Figure 3.18: Logical relations describing capability safety. Figure adapted from Georges et al. [62].

the event anyway), and if x makes $\mathrm{pr}(x)$ update the trace in the local state invariant $\mathrm{inv}_{\mathrm{st}}$. Recursively applying this theorem allows efficiently deriving higher-level driver specifications from lower-level ones. Note that view shift lowering accumulates resources in P and Q until the bottom-level driver is reached, at which point all higher-level invariants $\mathrm{inv}_{\mathrm{P}}$ are updated at once.

3.4.3 A semantic model for capability safety

In this section, we describe how we reason about the fact that unknown adversarial code satisfies the security objectives. We use a logical relations model to capture the notion of capability safety, i.e. the universal contract, that the hardware capabilities provide. Concretely, we will verify that our own wrapper closures are safe to execute, and that the adversary's code is safe to execute, starting from safe register states, in particular states containing the wrappers' closures.

Since our capability machine builds upon the bare-bones capability machine of Georges et al. [62], we reuse their *logical relation without revocation* and its formalization in Iris, and repeat a simplified version of the separation logic definition in Figure 3.18. Additionally, we reprove their fundamental theorem (Theorem 3.2 below) in the presence of MMIO.

We restrict our explanation of Figure 3.18 to the essentials required to understand the key concepts behind our proofs. The model consists of three different, mutually recursive relations:

- The value relation $\mathcal{V}$: Word $\rightarrow$ iProp (with iProp the type of propositions in Iris) defines when a word is safe.
- The expression relation $\mathcal{E}$: Word $\rightarrow$ iProp defines when a capability can safely be executed in the pc register.

- The register relation $\mathcal{R} : \text{Reg} \rightarrow \text{iProp}$ lifts the value relation to an entire register bank (bar the pc). A register bank is safe if each general purpose register is safe.

Technically, these relations are well-defined by so-called *guarded recursion*, see [62, 80].

We first discuss the definition of the value relation. $\mathcal{V}(w)$ specifies an upper bound on the authority over memory that the word w carries. Since integers and O-capabilities do not represent any authority over memory, they are always safe, as expressed by $\top$. Enter capabilities are safe if it is safe to execute them in the pc after jumping to them (hence the RX permission). The presence of the persistent modality $\square$ and the later modality $\triangleright$ can be ignored by readers unfamiliar with them. All remaining capability types have read permission and these capabilities are safe if, for each a' in their memory range $[b, e)$, an Iris invariant (denoted by a boxed assertion) exists that asserts that memory location a' will always contain a safe value. The assertion $a' \mapsto_{p'} w$ in the invariant expresses unique ownership of the memory location a' with permission p' and stored word w. Additionally, this assertion implies that a' cannot be an `MMIO` location. This ensures that adversaries cannot gain direct access to `MMIO` memory and bypass our wrappers. Note that the invariant permits using a p' that is at least as strong as p, i.e. $p \preccurlyeq p'$. This allows for downgrading of capability permissions and aliasing of different permissions. Notice that no additional assertions have to be added to the definition of $\mathcal{V}(p, b, e, a)$ for capabilities that carry additional write or execute authority. The intuitive reason is that the invariant already enforces that any written value needs to be valid, and that having read authority over a part of memory suffices for an adversary to copy code over to a region that it has write-execute permission over, and execute it there.

Next, we discuss the execution relation $\mathcal{E}$. It states that, given ownership of any initial register bank *reg* that satisfies the register relation $\mathcal{R}$, we can safely run the machine with v in the pc register. The weakest precondition assertion wp uses a trivial postcondition $\top$, which at first sight might seem odd. This suffices because Iris' weakest precondition implicitly enforces that all invariants hold at each step of execution. Thus, any invariants related to OBJ-**X** that a wrapper might define will also be enforced through the expression relation by the wp assertion. For example, if wrapper_1 were to set up an invariant that ensures that P_1 holds over the current trace t, i.e. $\text{P}_1(t)$, then this invariant is upheld at each step of execution when jumping to Adversary 1, under two conditions. First, Adversary 1 should be safe to execute. i.e. $\mathcal{E}(\text{RWX}, adv\_b_1, adv\_e_1, adv\_b_1)$ holds. Second, to meet the precondition of the expression relation, Set-up Code 1 has to pass the adversary a safe register block, which notably requires proving that the read and write closures for wrapper_1 are safe, since they are shown to be located in r_1 and r_2 when jumping to Adversary 1 in Figure 3.10. This latter condition can be proven relatively straightforwardly from the contracts we described in Section 3.4.2. We hence focus on proving the different adversaries in our examples safe, by leveraging the *FTLR* (*fundamental theorem of the logical relation*).

The fundamental theorem has the following simple statement:

Theorem 3.2 (FTLR). $\forall w. \mathcal{V}(w) \mathrel{-\!\!*} \mathcal{E}(w)$.

In other words: if a word is safe, it can safely be executed as well. The statement is identical to Georges et al.'s FTLR, but the proof is slightly different due to the presence of `MMIO`.

Through this theorem, it is easy to prove that for each region Adversary **X** in Figure 3.9, $\mathcal{E}(\textsc{rwx}, adv\_b_{\mathbf{X}}, adv\_e_{\mathbf{X}}, adv\_b_{\mathbf{X}})$ holds. Since each init_config_**X** predicate requires initial memory to not contain any capabilities, since integers are always safe, and since the different set-up codes do not change the adversary's memory, $\mathcal{V}(\textsc{rwx}, adv\_b_{\mathbf{X}}, adv\_e_{\mathbf{X}}, adv\_b_{\mathbf{X}})$ holds. The fundamental theorem then proves the result.

3.4.4 Sharing verification effort for fixed-structure wrappers

For wrappers x that satisfy some conditions, we have developed a method in Coq that requires minimal manual verification effort in proving the functional contract and safety of the closures for these drivers. The conditions are as follows:

- Respect the calling convention described in Section 3.3.
- Enforce their security objective using an atomic invariant of the form $\mathrm{inv}_{\mathrm{P}}(F_x, P_x, tp_x)$.
- Consist of a single read and write closure, and only jump to the read and write closures of their layer below driver. The state invariant $\mathrm{inv}_{\mathrm{st}}$ expresses both ownership of the code for x, and describes the closures for $\mathrm{pr}(x)$ in the local state, and will hence be parameterized by the relevant addresses of both x and $\mathrm{pr}(x)$. It is then denoted as follows: $\mathrm{inv}_{\mathrm{st}}(d\_r_{\mathrm{pr}(x)}, d\_w_{\mathrm{pr}(x)}, d\_e_{\mathrm{pr}(x)}, d\_r_x, d\_w_x, d\_e_x)$.
- Have an implementation that satisfies a specific template, which is parameterized by instructions *check_read* and *check_write* to check $P_x \circ F_x$ in respectively the read and write closures. Wrapper 1 in Figure 3.13 is structured like this, for example.

If these conditions are satisfied, the only manual effort involved is a proof that *check_read* and *check_write* indeed correctly enforce $P_x \circ F_x$ on x's view of the trace, denoted by $\mathrm{check\_spec\_read}(F_x, P_x, check\_read)$ and $\mathrm{check\_spec\_write}(F_x, P_x, check\_write)$, respectively. These conditions are enforced in the local state invariant $\mathrm{inv}_{\mathrm{st}}$, adding the parameters P_x and F_x to it.

Under the above conditions, we can define simple contracts in the style of Section 3.4.2 for the driver's read and write closures, that only depend on d_r_x, d_e_x, d_r_x/d_w_x, and tp_x (to determine whether or not a view shift is required in the contract), and are denoted read_spec(d_r_x, d_e_x, d_r_x, tp_x) and write_spec(d_r_x, d_e_x, d_w_x, tp_x).

We can then state the following *lifting theorem*:

Theorem 3.3 (Lifting-theorem-read)**.**

$$\frac{\begin{array}{c}\text{inv}_{\text{P}}(F_x, P_x, tp_x) \qquad \text{read\_spec}(d\_r_{\text{pr}(x)}, d\_e_{\text{pr}(x)}, d\_r_{\text{pr}(x)}, \text{False}) \\ \text{inv}_{\text{st}}(d\_r_{\text{pr}(x)}, d\_w_{\text{pr}(x)}, d\_e_{\text{pr}(x)}, d\_r_x, d\_w_x, d\_e_x, F_x, P_x)\end{array}}{\text{read\_spec}(d\_r_x, d\_e_x, d\_r_x, tp_x)}$$

A similar result holds for the write spec. Note that this theorem is generic in tp_x. It states that if all necessary conditions on the state of a wrapper are met (inv_{st}), and if an invariant enforces F and P on the trace (inv_{P}), then we can lift a spec for the pointed-to wrapper in the previous layer, to the current layer. The bottom-most wrapper, i.e. wrapper_0 in the examples, constitutes the base case in this theorem, and still has to be manually verified to satisfy read_spec and write_spec, since its code and local state do not satisfy the template (cfr. Figure 3.11 and LS_0 in Figure 3.7).

Finally, we can prove top-level wrapper closures secure using the following theorem:

Theorem 3.4 (Wrapper-safety-read)**.**

$$\begin{array}{c}\text{read\_spec}(d\_r_x, d\_e_x, d\_r_x, \text{True}) \mathrel{-\!\!*} \\ \mathcal{V}(\mathcal{E}, \text{Global}, d\_r_x, d\_e_x, d\_r_x)\end{array}$$

Again, a similar result holds for the write spec.

Since we have abstracted most reasoning related to wrappers into Theorems 3.3 and 3.4, the `check_read` and `check_write` instructions are indeed the only code we need to hand-verify to ensure wrapper safety. We used this approach to verify the first example. The shape of the local state and the code of the second example is slightly different, due to the presence of the timer closure, so it did not fit this approach. However, Theorem 3.4 still applied once the read_spec and write_spec were proven.

3.5 Related Work

Hardware-supported security mechanisms as well as software verification have been important ingredients of secure system development for a long time [97]. We discuss the most important lines of related work and how they differ from our results.

One distinguishing feature of our work is that we support *robust* modular verification. Robust verification requires underlying programming language or hardware support to protect verified code from untrusted code. Earlier work has demonstrated how to robustly verify safety properties in settings where that protection is not nested. For instance, Sammler et al. [136] and Jia et al. [73] have proposed approaches to robustly verify safety properties in the presence of untrusted code that is confined using some sandboxing mechanism. Alternatively, Agten et al. [7] have used trusted execution environments (TEEs) like Intel SGX [37] or Sancus [114] to protect a verified module from an unverified context, but verification is at the level of C code and their focus is on proving assertions about the protected module rather than full system properties. In our approach, protection of verified code is provided by the capability-based instruction set architecture, and this enables handling of nested protection.

Protecting verified code from unverified code is of course closely related to protecting trusted system software from untrusted user code. Operating systems, microkernels, and hypervisors use hardware privilege levels to protect themselves, and hence the rich line of work on verifying properties of such system-level software can be seen as an instance of robust modular verification and hence related to our work. Some important milestones include the verification of the seL4 microkernel [86, 85], and the verification of Microsoft's Hyper-V hypervisor [36]. The focus is however on proving properties (such as functional correctness, or selected safety properties) of the system software under a single attacker model where all non-privileged code is untrusted. Like our work, the verification of CertiKOS [29] supports modular (compositional) and layered verification of device drivers. But an important difference is that verification in CertiKOS is *compositional* but not *robustly* modular: only user-level code is isolated at run time from kernel-level code, and any unverified code at kernel-level becomes part of the trusted computing base. The journal version of the CertiKOS driver verification paper [30] also has an extensive overview of other related work on operating system verification.

The usefulness of being able to nest protection systems has been recognized in the system security research community, and several systems have been proposed that support, for instance, nested virtualization [56, 81]. However for none of these systems, any code-level formal guarantees are provided.

It is the reliance on capabilities as the underlying protection mechanism that enables arbitrary nesting for our approach. Capability-based architectures have a rich history [94], and have been proposed as a security foundation for both high-level languages [99, 143, 152] and assembly languages [28]. The fact that capabilities support nesting has been observed before, e.g. in Mark Miller's PhD thesis [105]. It has also been known for a long time that they provide a great foundation for nestable security architectures in high-level languages [170]. Over the past decade, capabilities at the instruction set architecture level have seen renewed interest, largely thanks to the CHERI project [176]. CHERI is a hybrid architecture that supports capability based protection for user-level code and classical memory protection for isolating kernel and processes. Hence, CHERI does not use the nested encapsulation for wrappers that we study in this paper. However, CheriRTOS [183], a CHERI-aware real-time operating system, supports capability-based fine-grained isolation for device drivers and would be a candidate implementation platform for our verified wrapper stacks.

Capability-based systems support the enforcement of security properties in the presence of arbitrary untrusted code in the system through what are typically called *object capability patterns*, like the membrane or caretaker patterns [105]. It is only relatively recently that sufficiently powerful formal reasoning approaches have been developed that can prove such properties. Devriese et al. [44] proposed a reasoning approach based on logical relations, (what we now call) universal contracts and the concept of *effect parametricity*. Swasey et al. [152] developed the first program logic, OCPL, that can compositionally specify and verify the properties enforced by object capability patterns. Building on these ideas, program logics have been developed to reason about software in low-level capability-based instruction set architectures [142, 62]. These logics, as ours, are built in Iris [80], a separation logic framework for building program logics. Iris integrates, unifies, and simplifies a wide variety of mechanisms for reasoning about programs that can be higher-order, concurrent, or use mutable state. The results in this paper can be seen as an application and extension of these logics to prove security properties for multiple stakeholders in a system with nested encapsulation.

One of the motivations for our approach is the minimization of the Trusted Computing Base (TCB). The various stakeholders in the system want to ensure their security objectives while trusting as little other software as possible. The use of small software modules isolated by some hardware protection mechanism to enforce full-system security properties has been proposed in multiple guises in the system security field. DriverGuard [31] uses virtualization techniques to implement fine-grained protection on I/O through specific devices with a small TCB. Para-passthrough virtualization [140] specifically aims to provide full-system guarantees while relying only on a small piece of software, albeit for only a single attacker model. One could argue that micro-kernels or hypervisor-based systems are similar. We show that it is possible to apply this principle at multiple levels in the same system and verify security.

Related to minimization of the TCB is the idea of *compartmentalization*, breaking a large program in smaller mutually distrusting parts and relying on some underlying security mechanism to protect the parts from one another. Juglaret et al. [76] have studied the formal guarantees provided by a compartmentalizing compiler. They consider multiple *compromise* scenarios, where an attacker can compromise different subsets of program parts, somewhat similar to our consideration of multiple attacker models. However, they only consider the benefits provided by a compartimentalizing compiler, and do not consider verification.

If the full system security objectives to be verified include statements about I/O through a given device, then necessarily the device driver(s) for that device will need to be verified. Hence, the approach proposed in this paper verifies a subset of the driver stack for a device depending on the attacker model. Device drivers have been a target of verification in a wide body of related work, using techniques ranging from model checking (e.g. [17]) to deductive verification (e.g. [127]). The objective of these verification efforts is to improve kernel reliability by showing that drivers correctly use specific kernel APIs, or do not have memory safety or concurrency bugs. That is very different from our objective of verifying that a thin wrapper around a device enforces a specified security property.

3.6 Conclusion and Future Work

The fine-grainedness of hardware capabilities, in combination with object capabilities as a primitive enabling encapsulation, allows for efficient and safe nesting of wrappers enforcing security properties without costly context switches. This would be hard to achieve through conventional, more coarse-grained security primitives. By capitalizing on capabilities and extending an existing formal model for a capability machine, we managed to generalize classical robust modular verification to a nested setting. With *nested* robust modular verification, a system that consists of multiple layers is verified robustly several times, each time verifying an increasing number of layers and enforcing more security objectives, and considering the rest of the code base as well as the environment untrusted. Crucially, our approach retains the compositionality of the programming logic and does not require verifying the same code multiple times, even when proving security objectives for different layers. Our Iris development provides ease of use when modularly verifying nested wrappers, only requiring specific checking instructions to be verified and lower layer wrappers proven safe, in order to obtain proofs of safety for an entire stack of wrappers. These proofs of wrapper safety are essential in proving the different full-system security properties.

In future work, we wish to extend our basic model of IO, to achieve nested full-system guarantees in more involved settings. Concretely, we believe it to be possible to achieve similar guarantees when adding interrupts to the basic capability machine. This would require formalizing interrupts dynamics, redefining the notion of weakest precondition correspondingly, as well as proving specifications for registered interrupt handlers themselves. Another interesting extension is allowing peripherals to perform some form of capability-restricted DMA access, as hinted at by Markettos et al. [100]. Additionally, with the advent of modern capability hardware in the form of Arm's Morello prototype [11], it will soon become possible to perform practical experiments on capability-enabled hardware, thereby obtaining a fair comparison between our work and similar approaches using different security primitives.

Acknowledgments

We thank the anonymous reviewers for valuable comments and suggestions. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation; by the Research Foundation - Flanders (FWO); and by DFF project 6108-00363 from The Danish Council for Independent Research for the Natural Sciences (FNU). This research was partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity. Thomas Van Strydonck holds a Research Fellowship of the Research Foundation - Flanders (FWO). Amin Timany was a postdoctoral fellow of the Flemish research fund (FWO) during parts of this project.

Chapter 4

CHERI-TrEE: Flexible enclaves on capability machines

Publication Data

This chapter contains currently unpublished work that we are looking to submit. Its publication data are as follows:

> Thomas Van Strydonck, Job Noorman, Leonardo Alves Dias, Jennifer Jackson, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. CHERI-TrEE: flexible enclaves on capability machines. *In Submission*

The paper discusses a novel design for enclaved execution atop capability machines. Additionally, three implementations of the design are demonstrated, and were mainly contributed to by the following people:

- The formal specification in Sail-CHERI-RISC-V was contributed by my thesis student, Robin Vanderstraeten, and cleaned up and expanded by me.
- The RTL-implementation in the extensible RISC-V core was implemented by Job Noorman, where Leonardo Alves Dias and Jennifer Jackson took care of the FPGA implementation.
- The implementation in ARM Morello was done by Jennifer Jackson.

The writing was a joint effort by all collaborators.

Compared to the version we submitted, the version included in this thesis has the following additions:

- Section 4.3.2: more nuanced discussion of trade-offs between memory sweep and linear capabilities.
- Section 4.3.3: additional explanation of how our design does not allow reuse of otypes between enclaves.
- Section 4.3.4: clarification that capabilities also help enclaves defend against API-level attacks.
- Section 4.3.4: further details of how nested enclaves are initialized in our design.
- Section 4.6: further explanation of why we think enclaves in general and CHERI-TrEE in particular are an interesting target for verification.

Abstract

This paper studies the integration of two successful hardware-supported security mechanisms: capabilities and enclaved execution. *Capabilities* are a powerful and flexible security mechanism for implementing fine-grained memory access control and for compartmentalizing untrusted or buggy software components. Capabilities have a long history, but have gained significant momentum recently, as evidenced by ARM's experimental Morello processor that supports the Capability Hardware Enhanced RISC Instructions (CHERI). *Enclaved execution* is a popular mechanism for dynamically creating Trusted Execution Environments (TEEs), called *enclaves*. Enclaves are isolated execution contexts that protect integrity and confidentiality of software in the enclave (even against compromised system software) and that support attestation.

Integrating capabilities and enclaved execution in a single processor is challenging because they overlap partially in their security objectives and a clean integration should unify the way in which these overlapping objectives are achieved. In addition, it is not obvious how attestation should interact with capabilities. In this paper, we propose CHERI-TrEE: a novel design for a processor that cleanly integrates support for both capabilities and enclaved execution. CHERI-TrEE targets low-end embedded systems without virtual memory. We show that CHERI-TrEE is greater than the sum of its parts, by showing how it naturally supports useful features that have traditionally been hard to support in enclaved execution, like dynamically growing and shrinking enclaves, non-contiguous and nested enclaves, sharing of memory between enclaves etc. We implement our proposal both in hardware on a RISC-V processor, as well as in a small software hypervisor on top of ARM Morello, and evaluate impact on performance and hardware resources.

4.1 Introduction

There is a wide variety of hardware supported mechanisms to securely isolate software, each with its own strengths and limitations. This paper is about the integration of two related but fundamentally different such mechanisms: capabilities and enclaved execution.

Capabilities are unforgeable tokens of authority granting rights to system objects. They are a powerful security mechanism for implementing fine-grained access control and for compartmentalizing untrusted or buggy software components. *Capability machines* implement the concept of capabilities at the machine code level: they provide hardware support for capabilities by defining an instruction set architecture (ISA) that provides access to system memory only through *memory capabilities*, a kind of hardware-supported fat pointers. The ISA is designed to make sure that software can only create capabilities that represent a subset of the authority that the software already holds, and hence that capabilities are a secure basis for implementing memory access control and isolation. Next to memory capabilities, capability machines can support a wide variety of other kinds of capabilities, including for example *object capabilities* that can control access to software defined objects, or *sealing capabilities* that can symbolically encrypt or decrypt other capabilities. Capability machines have a long history [94], but have gained significant momentum over the last decade with, for instance, the development of the CHERI system [176], and with the ARM Morello project [11] that integrates the concepts of CHERI in the widely used ARM architecture.

Enclaved execution is a security mechanism that supports the run time creation of *enclaves*, execution environments for software components that are strongly isolated and that can attest their identity to other code running either locally on the same platform or remotely on other platforms. The idea of Enclaved Execution Systems (EES) is more than a decade old [102], and currently many implementations are available [113, 88, 25, 38, 114, 90], both in research prototypes, as well as in commercial systems like Intel's Software Guard Extensions (Intel SGX).

Capabilities and enclaved execution have been studied for both low-end embedded systems, and higher-end systems that support virtual memory or virtualization instructions. In this paper, we focus on low-end embedded systems, similar to CHERI-RTOS [183] (for capabilities) or Sancus [114] (for enclaved execution). Additionally, other desirable enclave properties that are largely orthogonal to the integration of capabilities and enclaves are left out of scope: we do not consider availability guarantees, hardware-based attacks (cold-boot attacks, malicious DRAM, …) or side-channel attacks (timing or cache-based side-channels, memory bus snooping, …).

Problem statement The problem addressed in this paper is how to integrate capabilities and enclaved execution in a single design. This is challenging because (1) these mechanisms overlap partially in their security objectives (for instance both mechanisms provide their own form of controlled invocation) and a clean integration should unify how these overlapping objectives are achieved, and because (2) it is not obvious how (remote or local) attestation should interact with capabilities.

To the best of our knowledge, this paper is the first to propose a design that cleanly integrates both mechanisms in a single processor ISA. Roughly speaking, our design roughly proceeds as follows: enclaved execution is known to be a *complex* security mechanism [54], hence we first decompose it into a set of maximally independent primitives. Next, we implement these primitives on top of the state-of-the-art CHERI capability machine [176, 175], maximally reusing existing CHERI mechanisms. For instance, enclave isolation reuses standard CHERI object capabilities. The new mechanisms we introduce are lightweight, orthogonal and contribute features that do not yet exist in CHERI. They allow (1) obtaining guarantees about exclusive ownership of a memory region (i.e., no other code on the system has a reference to said memory), and (2) obtaining sealing capabilities (a private key used to symbolically encrypt or sign other capabilities) derived from an enclave's identity.

We demonstrate that the resulting system, CHERI-TrEE, can be used as an EES, allowing other code on the system to establish trust in an enclave. CHERI-TrEE is greater than the sum of its parts. Our reuse of existing CHERI features achieves economy of mechanism and reduces complexity and cost. At the same time, it yields an EES with novel characteristics like dynamically growing and shrinking enclaves, non-contiguous enclaves, nested enclaves, sharing of memory between enclaves, etc. Concretely, in this paper we contribute:

- A decomposition of enclaved execution into more basic primitives.
- The design of CHERI-TrEE, a capability-based ISA extension that supports these primitives and hence integrates capabilities and enclaved execution.
- A specification and security argument for classic enclaved execution operations (initialization, unloading, local attestation, secure communication) on top of CHERI-TrEE.
- An open-source implementation of CHERI-TrEE on top of RISC-V, including both a Sail specification of the ISA and a hardware implementation on FPGA, as well as a proof-of-concept implementation on ARM Morello [11]. Our evaluations and benchmarks are open source.
- An evaluation of the impact on performance and hardware resources of the extension on the considered platforms.

For reasons of anonymity, we cannot provide the implementation sources during the review process. Files can be made available to reviewers on request.

4.2 Background

We implement all required primitives by extending a CHERI-based capability machine [176]. We thus briefly recap the background on enclaved execution and capabilities.

4.2.1 Enclaved execution

Enclaved execution is a security mechanism that enables *secure remote computation* [37] with a small Trusted Computing Base (TCB). It supports the runtime creation of *enclaves*, software components that are strongly isolated and that can attest their identity to other code running either locally on the same platform or remotely on other platforms. A number of variations on this idea have been designed and implemented by researchers [102, 88, 114, 38, 90], and Intel have commercially implemented the Software Guard Extensions (Intel SGX). We provide an overview of existing designs in Section 4.7.

The typical life cycle of an enclave goes as follows: First, untrusted code creates the enclave and initializes it from a static binary code image (e.g., , a .dll file). After initialization, the enclave is supposed to be isolated from all other (non-TCB) software, and has an enclave identity based on the code image initially loaded. At this point, interaction with the enclave is possible: the enclave context (untrusted code and/or other enclaves) can call into the enclave, and the enclave can call out to its context. Such interactions can be authenticated: the context can get proof of the identity of an enclave (*attestation*), and similarly the enclave can get proof about the identity of other enclaves calling in. Most EESs extend attestations to remote platforms: a remote party can get proof about the identity of an enclave it is interacting with. Finally, an enclave can be terminated. Care must be taken to ensure that enclave secrets do not leak on termination.

Existing EESs vary widely in implementation details of this life cycle. There are differences in the way isolation is implemented, in the interaction with other isolation mechanisms (like virtual memory), and in how termination is handled. However, all existing systems face challenges in securely handling natural features, including: nesting of enclaves, non-contiguous enclaves, dynamic enclave memory (de-)allocation, efficient memory sharing between enclaves, the number of supported enclaves, and so forth.

4.2.2 Capability machines and CHERI

Capability machines define an instruction set architecture (ISA) that provides access to system memory only through *memory capabilities*, a kind of hardware-supported fat pointers, i.e. pointers that carry metadata about the bounds within which they are valid. The ISA is designed to enforce *monotonicity* of authority: software can only create capabilities that provide access to a subset of memory that the software initially had access to. Hence, capabilities are a secure basis for implementing memory access control and isolation. The base platform that we build our design on is a variant of CHERI-RISC-V. Capabilities in CHERI-RISC-V can be thought of as pointers (memory addresses) extended with additional metadata. The metadata relevant for this paper includes:

- *Base* and *length* fields that define a contiguous range of memory addresses.
- An *otype* field used for sealed capabilities and explained in more detail below.
- A *permissions* field that determines what can be done with the capability (e.g., read/write/execute).

For memory capabilities, the *address* of a capability represents a memory address. Such a capability can be used in store and load instructions to access that memory address, provided the address is within the bounds specified by the base and length fields, and compliant with the permissions specified in the permissions field. The ISA provides instructions to inspect and modify metadata fields, but only in ways that do not increase the authority that a capability carries. For instance, bounds can be reduced but not expanded.

Besides memory capabilities, CHERI supports other kinds of capabilities, for which the interpretation of the metadata, and the possible use of the capability varies. Notable for our purposes are so-called *sealed capabilities* [176], a type of CHERI capability that cannot be used in operations like stores and loads, or have its fields altered; only reading fields is allowed. Sealed capabilities are sealed with a specific seal (i.e., a key), represented by a value in the otype field. The highest otype value represents an unsealed capability.

Capabilities can be sealed and unsealed through the `CSeal` and `CUnseal` instructions. The `CSeal cd, cs1, cs2`[1] instruction takes unsealed capabilities in `cs1` and `cs2`, and seals `cs1` with the otype in the address field of `cs2`, placing the result in `cd`. To avoid arbitrary capabilities being used to seal other capabilities, `cs2` must have the bespoke `Permit_Seal` permission bit set, which allows it to seal other capabilities with otypes within its bounds. The `CUnSeal cd, cs1, cs2` instruction is the dual of `CSeal`; `cs1` must now be sealed, and `cs2` is required to have the `Permit_UnSeal`

[1] As in the RISC-V and CHERI specifications, we use `rs1` and `rs2` for source and `rd` for destination integer registers, while using `cs1`, `cs2`, and `cd` for capability registers throughout this paper.

permission bit set. The otype in the address field of cs2 must match the otype of cs1. The unsealed result is placed in cd.

The purpose of these capabilities in CHERI is dual. First, otypes can be used to implement efficient symbolic encryption and signing, using the aforementioned permission bits. A capability that has both Permit_Seal and Permit_UnSeal bits set can be used for symmetric encryption, because it can both encrypt (seal) and decrypt (unseal) messages (capabilities). If a capability carrying the Permit_Seal permission is made public and a Permit_UnSeal counterpart kept private, we get public key encryption. The converse set-up results in a digital signature scheme.

Secondly, sealed capabilities can be used to implement secure domain transitions through the CInvoke cs1, cs2 instruction. If both cs1 and cs2 contain capabilities with matching otypes and solely cs1 has execute permission, then the two capabilities are atomically unsealed, and cs1 is installed in the program counter register, continuing execution from the invoked domain. Together, cs1 and cs2 are said to constitute a *sealed pair*, with cs1 the code and cs2 the data capability. Such pairs can safely be passed to adversarial code, as both capabilities are sealed and can only be invoked together. To distinguish sealed pairs from other sealed capabilities, a permission Permit_CInvoke determines whether sealed capabilities can be invoked together.

4.3 The design of CHERI-TrEE

In this section, we propose a decomposition of the requirements for enclaved execution into orthogonal properties (Section 4.3.1). These properties serve as a guideline for the design of CHERI-TrEE, our capability-based EES. Section 4.3.2 considers the CHERI capability machine as a starting point, and lists for each property whether it can be enforced through existing capability primitives or whether extensions have to be defined. With this analysis in hand, Section 4.3.3 discusses the concrete instructions we implement to achieve secure enclaves, as well as pitfalls developers should be mindful of. Finally, Section 4.3.4 highlights how our bottom-up approach, maximally reusing the flexibility of architectural capabilities, allows for greater flexibility than the state of the art.

Capabilities and enclaved execution have been studied both for systems that support virtual memory, and for systems with a single physical memory address space. This paper focuses on the latter kind of system. The combination with virtual memory and address translation brings additional challenges that are left for future work discussed in Section 4.6.

4.3.1 Decomposing enclaved execution into core properties

In our view, the core security properties required to build an EES can be decomposed into the four categories listed below. Some other properties (e.g., confidential deployment [114], secure storage) are also relevant, but not *essential* to the construction of a functional EES. The core properties are:

① **Exclusive access**: A mechanism to guarantee exclusive access to specific memory areas is required for the secure initialization of an enclave. In principle, the enclave can be allowed to share its uniquely owned memory once it has been securely initialized, but few enclaved execution systems support this. For example, Intel SGX relies on *Processor Reserved Memory* and the initialization process to guarantee exclusive access, while Sancus relies on program-counter based access control.

② **Controlled invocation**: To avoid reasoning about the correctness and security of many possible control-flow paths, enclaves can only be invoked at predefined *entry points*. Examples are *ecalls* in Intel SGX and *entry points* in Sancus. In most systems, entry points are declared at enclave creation time, but at least conceptually, new entry points could also be created dynamically.

③ **Enclave identities and attestation**: To enable multi-enclave/distributed applications, enclaves and third parties require a mechanism to establish trust in the correct initialization and execution of another enclave. This authentication process, called attestation, occurs either locally or remotely. It requires a notion of *enclave identity*, usually based on a cryptographic hash of the enclave code and metadata. Locally, the architecture provides a mechanism to communicate identities and authenticate enclaves to other code running on the same platform. Remote attestation requires cryptographic support, either in the form of a public key infrastructure (as in Intel SGX) or a symmetric key derivation scheme (as in Sancus).

④ **Secure communication**: A mechanism to securely communicate between two enclaves, both locally and remotely, is essential to achieve integrity and confidentiality. Locally, either CPU registers or shared memory can be used. Because enclaves might be deinitialized and replaced at any point, encrypting and signing messages cryptographically is required to ensure integrity and confidentiality. The efficiency of the involved cryptography is performance-critical. To avoid having to sign and encrypt messages, some EESs have built-in mechanisms to check liveness of the sender and receiver enclave. Here, time-of-check to time-of-use (TOCTOU) attacks can be an issue (e.g., for Sancus). For the remote case, the key distribution infrastructure of attestation can often be reused for secure communication with standard protocols.

⑤ **Secure interruptability**: When an enclave's execution is interrupted, its register state should not be accessible to untrusted code as this potentially breaks confidentiality guarantees offered by the enclave. Additionally, the enclave's register state must be correctly and securely restored after servicing the interrupt.

Because capabilities provide no protection during network transition, the design of remote attestation and remote secure communication would mostly reuse existing solutions. The remote aspects of enclaved execution are therefore left for future work and further discussed in Section 4.6.

4.3.2 Satisfying the security properties

We now discuss how we enforce these properties in our design that builds on CHERI-RISC-V, reusing CHERI primitives where possible (✓) and extending the TCB otherwise (✗). In the remainder of Section 4.3, the TCB is taken to be hardware-only (except for a small software interrupt handler). However, section 4.4.4 illustrates that this is not required.

✗ **Exclusive access** at runtime is not supported out-of-the-box in CHERI, so we add *memory sweep* functionality to the TCB. During a memory sweep, the TCB checks all of memory and all architectural registers (including special registers) for the presence of capabilities that overlap with a given capability. To successfully complete enclave initialization, the unique ownership of an enclave's code and data sections need to be verified through such a sweep. We could have alternatively used *linear capabilities* [144, 162, 175] for providing exclusive access without a sweep. Admittedly, their design is complicated by technical concerns related to concurrency and implementation in hardware (see [144] for a discussion). However, if linear capabilities are only used to support enclaves, the runtime overhead they cause might be limited. Additionally, they naturally seem to avoid the denial-of-service attacks that naively implemented memory sweeps could be abused for, at least as long as the linear capabilities do not need to be revoked themselves: this would again require a memory sweep. We leave further investigation of the hurdles involved in implementing linear capabilities and the trade-offs versus memory sweeps for future work.

✓ **Controlled invocation** can be implemented using the aforementioned sealed capability pairs [176]. Concretely, if the enclave only shares sealed capabilities to specific entry points with adversary code, then capability safety ensures that the enclave cannot be otherwise entered. Alternatively, controlled invocation can be implemented through so-called *enter capabilities* [28], which intuitively combine both the code and data capabilities of a sealed pair into a single capability. This alternative could have simplified a few aspects of our design, but we did not implement it, as enter capabilities have only recently been added to CHERI.

✗ **Enclave identities and attestation** are not built into CHERI. The architecture hence has to use a cryptographic hash function to calculate an enclave's identity by hashing the code section. Additionally, to enable local attestation, an instruction needs to be added to securely look up this generated identity.

✓ **Secure communication** can be efficiently implemented locally through the symbolic encryption provided by sealed capabilities. Every enclave has a signing capability cap_sign with otype o_sign in its address and both `Permit_Seal` and `Permit_UnSeal` set. The enclave solely shares the `Permit_UnSeal` part of this capability with other code, so that it can exclusively authenticate its messages and other enclaves can verify them. Although it might seem unintuitive that a *capability* (rather than just the integer otype o_sign) is required to *verify* a signature, without this capability, recipients of signed values would have no way to remove the signature and access the payload underneath. Dually, the enclave has an encryption capability cap_enc with otype o_enc and shares only its `Permit_Seal` part so that only the enclave can decrypt messages encrypted with o_enc. We refer to the shared versions of both capabilities as an enclave's *public keys*, and to the full-authority versions as an enclave's *private keys*.

✗ **Secure interruptability** cannot be guaranteed in the presence of an untrusted interrupt handler: the confidentiality and integrity of an enclave depend on an adversarial interrupt handler not being able to read security-critical capabilities (e.g., the capability for the data section or cap_sign) from its register state. We resolve this problem by installing a minimal, trusted interrupt handler that cannot be altered or bypassed by untrusted code. The handler performs the necessary register sanitization before passing control to the adversary, and restores the register state on return. This solution is similar to the *security monitor* employed in Keystone [90], but our handler is more limited in scope (e.g., it does not need to manipulate memory protection state). Although we currently implement interrupt handling in software as a proof-of-concept, a Sancus-style hardware implementation is entirely feasible.

4.3.3 Fleshing out the design

Having studied how a capability architecture can accommodate secure enclaves at a conceptual level, we now consider different parts of an enclave's lifetime and define the instructions and software measures required to make our design secure in practice. We study enclave (de)initialization, local attestation, communication between enclaves, stand-alone memory sweeps and interrupts in more detail. Figure 4.1 serves as a guide for this section: it provides a schematic to illustrate a single enclave's initial layout in memory and the contents of the TCB that keeps track of all registered enclaves.

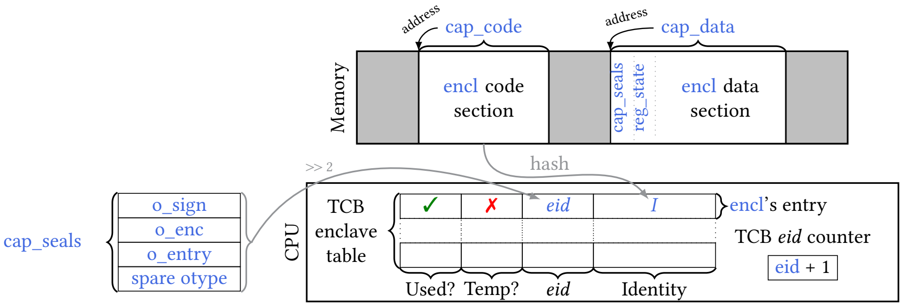


Figure 4.1: Memory and TCB state after initialization of an enclave encl. All values relating to encl typeset in blue; notably its identity (*I*) and *eid*, code and data sections cap_code and cap_data, its four assigned otypes and their capability cap_seals. The reg_state region stores the enclave's register state during an interrupt. Gray arrows illustrate how *I* is obtained by hashing the code section and *eid* by right-shifting the otypes. The TCB's four enclave table fields indicate whether an entry is in use (Used?), whether it is temporary (Temp?), and record the enclave's *eid* (*eid*) and identity (Identity). The TCB does not track the enclave's memory layout, but solely guarantees connection between the enclave's *eid* and identity. Figure inspired by Noorman *et al.* [114].

Initialization

An instruction `EInit` requests initialization of a new enclave, given unsealed capabilities cap_code and cap_data for the code and data section. It will:

- Perform a memory sweep to check unique ownership of cap_code and cap_data.
- Allocate a fresh otype o_entry to seal cap_code and cap_data, transforming them into a sealed pair and ensuring controlled invocation. This otype has to be spatially and temporally unique to avoid collisions with previously allocated enclaves and other usages of sealed capabilities.
- Generate fresh signing and sealing otypes o_enc and o_sign and write the corresponding capabilities cap_enc and cap_sign to the start of cap_data so that the enclave can set-up its symbolic encryption upon invocation. Again, both o_enc and o_sign have to be unique in space and time.
- Generate the enclave's identity by hashing its code section.
- Store this identity in the TCB along with o_enc, o_sign and o_entry so other parties can verify the identity associated with received sealed capabilities during local attestation. If the TCB is full, the instruction fails.

One might ask whether o_enc or o_sign could be reused to seal the entry point instead of o_entry. The answer is negative; reusing o_sign would allow adversaries to simply unseal the enclave's sealed code and data sections without calling `CInvoke`, whereas reusing o_enc would allow the adversary to create their own code section, and use that to unseal the enclave's data section by executing `CInvoke`.

Unfortunately, `EInit` is difficult to implement on a RISC ISA like RISC-V where instructions generally have only one output operand. `EInit` requires two outputs, because it has to overwrite both cap_code and cap_data with their now-sealed variants. To solve this, we split `EInit` into two separate instructions; `EInitCode` and `EInitData`, which initialize the enclave's code and data sections in two consecutive phases.

The `EInitCode cd, cs1` instruction is called first, with cap_code in `cs1`. To ensure that the unsealed cap_code is overwritten by its sealed counterpart, and to avoid writes to two different registers, `cd` and `cs1` are required to be equal. For efficiency reasons, `EInitCode` does not yet check uniqueness of cap_code but `EInitData` will perform a single sweep for cap_code and cap_data simultaneously. Computing the enclave's identity is also deferred until `EInitData`, as adversarial code might still have access to cap_code. `EInitCode` will hence generate a *temporary* TCB entry (to be later finalized by `EInitData`), which does contain the enclave's otypes but not its identity (we explain *eid* below):

✓	✓	*eid*	Don't care

`EInitCode` *does* generate all three aforementioned otypes. They are represented by adjacent otypes, to avoid storing all three separately and we reserve an additional fourth otype, which the enclave can use to e.g., create additional entry points. Because the four otypes are adjacent and 4-aligned, all but the last 2 bits are shared and these shared bits constitute *enclave identifier* (abbreviated *eid*, cfr. Figure 4.1) in the TCB. Note that any enclave otype is efficiently convertible into the *eid* by a 2-bit right shift and the specific role of each otype (except for o_entry) is up to software convention. Figure 4.1 defines cap_seals and illustrates where it is stored in cap_data. Once the *eid* has been generated, `EInitCode` seals cap_code with o_entry, writes it to `cd` and sets its address to its base (to avoid arbitrary entry offsets).

To ensure uniqueness of enclave otypes, a hardware counter (TCB *eid* counter in Figure 4.1) keeps track of the next *eid* and will fail to generate more *eid*'s when the otype space has been depleted. In case otypes are required for other purposes on the capability machine, the otype space should be split up.

For proper operation, we assume the otype space available to enclaves to be sufficiently large. This requirement is currently not met for CHERI Concentrate 64 (CC64), the compressed 64-bit representation of capabilities that CHERI defines for RV32 (32-bit RISC-V); only 4 bits are available for the otype field [179]. To resolve this issue, the CHERI specification contains a proposal to store otypes of sealed capabilities as metadata in memory instead of having the otype be in the capability itself [175]. For RV64 (64-bit RISC-V), 18 bits are available in CHERI Concentrate 128 (CC128), which poses fewer problems.

After `EInitCode`, the `EInitData cd, cs1, cs2` instruction finalizes the enclave's initialization. It requires a sealed code section cap_code (the result of calling `EInitCode`) in `cs1`, and an unsealed data section cap_data in `cs2`. As before, `cd` and `cs2` are required to be equal. The `EInitData` instruction now:

1. Looks up a temporary entry with *eid* cap_code's otype.
2. If found, checks unique ownership of cap_code and cap_data in a single memory sweep.
3. Verifies that cap_code does not contain any capabilities. Note that cap_data *is* allowed to contain capabilities (including internal references to cap_data and cap_code themselves), and is therefore not zero-initialized, as in e.g., Sancus. cap_data is the data part of a sealed pair, and those are not allowed to have execute permission in CHERI, in order to avoid data and code sections being used interchangeably. Therefore, execute permission for the cap_data memory region is lost upon enclave initialization, unless cap_data is allowed to store a self-referencing executable capability. The capabilities stored in the data section are software responsibility and of no importance to the architecture.
4. Calculates the enclave's identity *I* as the hash of the contents of cap_code, stores *I* in the temporary entry for *eid*, and marks the entry as permanent.
5. Stores the aforementioned cap_seals capability in the first address of cap_data.
6. Writes cap_data, sealed with o_entry, to `cd`.

After successful completion, the enclave has been initialized. Untrusted code can now invoke the enclave by executing `CInvoke` on cap_code and cap_data.

Upon first invocation, the enclave's code will initialize any necessary state and enable symbolic encryption by returning the public parts of its sealing and encryption keys. The enclave can create a so-called *fast entry point* at a different offset to skip initialization on subsequent invocations (taking care that the initial entry point does not reinitialize the enclave if it is invoked again).

Deinitialization

To deinitialize an enclave, the TCB can simply remove the entry for this enclave. To this end, an instruction `EDeInit rd, cs1` is provided. The register `cs1` takes a capability with the otype of the target enclave in its address field. Success or failure is indicated by writing 1 or 0 to `rd`. To prevent arbitrary code from executing `EDeInit` using e.g., entry points, `cs1` must have both `Permit_Seal` and `Permit_UnSeal` permissions set. Thus, only the enclave itself or parties that were granted access to (subranges of) cap_seals can deinitialize it. After `EDeInit`, an enclave can clear sensitive data (including its seals) and return the capabilities for its code and data section to untrusted code. As in Sancus, a processor reset is required to deinitialize rogue enclaves. A processor reset is also required in order to reuse the otypes of decommissioned enclaves in future initializations; our design currently makes no effort to reclaim used otypes, as this would require additional memory sweeps to reclaim otypes, and more sophisticated data structures to keep track of the otypes that are currently in use.

Local attestation

An enclave gets access to another enclave's entry points and public encryption/signing keys by e.g., retrieving them from an (untrusted) enclave registry, or by executing `CInvoke` on the entry point and receiving the keys or additional entry points as part of the return value. Regardless of the origin, local attestation requires a way to verify that an otype corresponds to an enclave of interest. The instruction `EStoreId rd, rs1, cs2` serves this purpose; `rs1` takes an otype (any one of the four assigned to an enclave), and the enclave identity of the corresponding enclave in the TCB (if any) is written to memory through the capability in `cs2`. As for `EDeInit`, a boolean is written to `rd` denoting success or failure. Naturally, `EStoreId` ignores temporary TCB entries, as these correspond to partly initialized enclaves. Note that when an enclave calls into another enclave, the callee is not required to attest the caller ahead of time. Indeed, the caller can pass its public keys along with the arguments, so the callee can perform local attestation of the caller while processing the call.

Secure communication

Different primitives combine to ensure secure communication between enclaves: First, any confidential arguments or return values should be *encrypted* with the recipient's public encryption key. Additionally, as capabilities carry authority, even non-confidential capability arguments and return values must be encrypted if their authority should not be made available to untrusted code. It is important that `Permit_CInvoke` is unset on capabilities sealed with these encryption keys, for a similar reason than why we cannot combine o_sign or o_enc with o_entry (as explained earlier).

Second, when returning from a call, the callee should *sign* (part of) the return value to allow the caller to confirm that its call was indeed processed by the callee. Conversely, the callee might also require the caller's signature to authenticate the caller; the callee

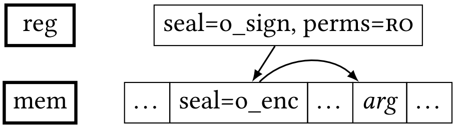


Figure 4.2: Combination of different secure communication primitives: argument *arg* is encrypted with otype o_enc, then signed with otype o_sign.

could e.g., have a whitelist of allowed enclaves. Lastly, to avoid replay attacks, the caller can make use of a *nonce* as part of its requests.

One might wonder how a capability with a single otype field can be used for both encryption and signing at the same time: The solution is indirection as demonstrated in Figure 4.2: a memory argument *arg* is first encrypted by an in-memory capability with otype o_enc, which is in turn signed by a capability with otype o_sign, present in one of the registers. This immediately illustrates one of the pitfalls of secure communication in our setting: messages are represented by capabilities, rather than bit-strings. Copying a bit-string corresponds to a deep copy, whereas copying a capability creates an alias. This is the reason the top-most capability in Figure 4.2 has read-only permission—if it allowed writes, an adversary could take a copy of the capability, remove the signature on the copy using the public signing key, and replace the underlying sealed capability with one of their own. This would allow the adversary to create arbitrary capabilities, signed with a third party's seal. Similar concerns are at play when encrypting capabilities.

As the literature shows, it is possible (but difficult) to construct a wide variety of secure communication protocols using the primitives (asymmetric encryption, signing, and nonces) [103]. Because our aim is to introduce a capability-based design for enclaved execution, not design secure communication protocols on top, we defer the development of these protocols to future work.

Separate memory sweep

As shown in the following, there is benefit in offering the memory sweep functionality of `CInitData` as a stand-alone instruction `IsUnique rd, cs1` as well. This instruction thus performs a memory sweep for the capability in `cs1` and uses `rd` to indicate success or failure.

Interrupt handling

For secure interrupts and system calls, we install a fixed, non-bypassable, trusted interrupt handler in the RISC-V `mtvec` register. When an enclave is interrupted, control passes to the trusted handler. The handler uses the enclave's data capability cap_data to save its register state into the reg_state region (cfr. Figure 4.1). Then, it seals cap_data using a unique, private otype o_handler, places the result in a predefined register, and jumps to the untrusted interrupt handler. For this scheme to work, cap_data should be present in a fixed register at all times. We use the `idc` (invoked data capability) register because it is atomically set to the unsealed data section when calling into an enclave using `CInvoke`. Once the untrusted code finishes servicing the interrupt, it returns to the trusted handler, which unseals the provided data section, restores the enclave's registers (apart from `pc`) and finally returns using the `mret` instruction. Calling `mret` simultaneously reenables interrupts and restores the enclave's code capability.

The tag of the `pc` capability stored in the reg_state region effectively functions as an "is interrupted"-flag: the trusted handler sets it by storing an enclave's `pc` at interrupt time, and unsets it before returning to said enclave. This flag is used by enclaves to avoid reentrancy issues: if the tag is enabled when an enclave is invoked, the enclave simply returns. Additionally, the trusted handler uses the flag to avoid storing the registers of a previously interrupted enclave and restoring the registers of a non-interrupted enclave. Much like Keystone, we leave the extension of our scheme to a multi-threaded setting (and nested interrupts, attestation of the interrupt handler, and the delegation of synchronous interrupts and errors to an enclave-private interrupt handler) to future work.

4.3.4 Flexibility of our bottom-up design

There are two main reasons for the maximal reuse of existing capability primitives: First, the additional hardware TCB remains limited. Second, the resulting enclaves do not rely on the hardware to manage their authority, but are self-governing, resulting in more flexible software-based design patterns. The cost of this flexibility is establishing unique ownership at runtime (i.e., the memory sweep) and an increased burden on the software developer to write correct and secure code. Although capabilities open up new avenues for application developers to shoot themselves in the foot (e.g. by leaking authority to an adversary), we hasten to point out that the spatial memory safety they offer also mitigates API-level exploits such as the ones identified by Van Bulck et al. [158]. For example, if an enclave does not share any capabilities pointing into its own memory with the adversary, then it does not require checks to ensure that pointers passed in from untrusted memory fall outside of its bounds. Hence, many attacks that depend on the improper implementation of these checks in enclave runtime implementations are avoided by construction.

Ways in which capability-based enclaves are more flexible than traditional enclaves include:

Growing, shrinking, nested and non-contiguous enclaves

In state-of-the-art EES, growing and shrinking enclaves is usually either impossible or prohibitively expensive. Nesting enclaves (i.e., creating an enclave inside an existing enclave) is impossible, because current hardware protection is defined on regions of memory, disallowing overlap. One motivation for having nested enclaves is that it would allow for easy virtualization (inside an enclave) of code that itself uses enclaves. All of these are possible in our design because an enclave's footprint is not managed by the TCB, but rather determined by the capabilities it owns. This also implies that enclaves can have non-contiguous footprints, e.g., to take exclusive ownership of an MMIO region corresponding to an external device, reported to be impossible in Sancus [157].

To grow an enclave, we employ the previously introduced `IsUnique` instruction. An enclave can request a sweep for a capability provided by untrusted code to verify unique ownership and add it to the enclave's memory footprint. In order to shrink, an enclave simply shares capabilities for part of its footprint with untrusted code. Nested enclaves can be initialized anywhere inside an enclave's memory footprint. The only restriction is that if an external party still holds a sealed pair for an entry point that overlaps with the nested enclave, the memory sweep will fail. Two solutions are currently available to enclaves wanting to initialize nested enclaves. First, they can extend their footprint with memory passed in from the adversary, and instantiate the new enclave there. Second, enclaves can create a more narrow entry point that does not overlap with the nested enclave, share it with the adversary, and rely on the adversary willingly giving up ownership of the initial entry point. In future work, we intend to experiment with a version of enclave initialization where the enclave's initial entry point only covers *part* of the enclave's initial memory footprint, such that a nested enclave can be instantiated in the remainder.

Sharing memory

Sharing memory between enclaves amounts to simply sharing a capability for a uniquely owned memory region between said enclaves. No encryption is required. Once enclaves are done sharing memory, there needs to be a way to revoke access to this memory. For short-term sharing of e.g., arguments in memory, so-called *local* capabilities [175] (which reside in registers only) could be used. Alternatively, an enclave can check whether its shared memory has been released using the `IsUnique` instruction. Lastly, to share non-confidential memory that does not contain any security-critical capabilities, enclaves can simply share a read-only view of said memory.

Early `EDeInit`

Once all parties that wish to communicate with an enclave performed local attestation, there is no need to keep the enclave's entry in the TCB. This is possible as the TCB entry does not provide any hardware protection, but serves the purely informative purpose of linking otypes and enclave identities. Hence, enclaves can prematurely execute `EDeInit` to free space and still operate correctly.

Secure communication without liveness checks

One of the most interesting characteristics of inter-enclave communication in our design is that we do not need to rely on enclave liveness checks (such as the ones present in Sancus) during secure communication. The caller enclave can invoke the callee without verifying whether the callee enclave still exists and the callee returns to the caller without checking its liveness. The reason we can afford to omit these checks is that the alternative, namely both signing and encrypting messages, is very cheap (taking a single instruction, contrary to non-symbolic solutions), and can hence be applied by default. The advantages of omitting liveness checks are that the hardware is simplified and any TOCTOU issues related to checking and then invoking an enclave are avoided.

Two-way sandbox

Capability enclaves naturally provide a *two-way sandbox* [95], meaning that other code running in the same address space is protected from the enclave (preventing attacks like [96]): enclaves can only manipulate memory they have appropriate capabilities for.

Dynamic entry points

Traditionally, enclaves either list their entry points at creation time or have standard, predefined entry points. As we reuse sealed pairs to enforce entry points, an enclave can dynamically create more entry points at runtime by creating a sealed pair.

Such dynamically created entry points could be selectively shared with attested counterparties to avoid the need to re-attest them on subsequent calls. To avoid mixing parts of different pairs, the enclave either needs to use different otypes for different entry points or have an entry point identifier in each data section.

Relocatable enclaves

`EInitData` does not include the enclave's base into the hash contrary to hashing in e.g., Sancus [114]. Consequently, a deployer is not required to know the memory address at which an enclave is located beforehand, providing greater ease in deployment. This does place the requirement of writing Position Independent Code (PIC) on the developer, but RISC-V has efficient support for this [171]. Our design decision implies that multiple instances of the same enclave have the same identity. Fortunately this poses no security risk, as both enclaves have different otypes and hence are distinguishable locally. In the remote case, key derivation can include each enclave's otype to create different keys.

4.4 Implementation

We now discuss three different implementations of our design from Section 4.3: one that serves as a specification of the ISA extensions and is written at the instruction level, abstracting away from the hardware, one RISC-V hardware implementation for studying performance, and a prototypical implementation on the ARM Morello Fixed Virtual Platform (FVP) simulator to show commercial feasibility.

4.4.1 Sail specification of the extended ISA

To obtain a more formal account of the architectural extensions proposed in Section 4.3.3, we created a software implementation of our design in Sail [13].

Sail

Sail is an ISA specification language, used to formalize ISA semantics. On top of that, Sail models serve various purposes: documentation can easily be generated from them, emulators (in C and OCaml) can be derived from the source code, and definitions can be exported to various proof assistants to enable reasoning about properties of the ISA. Sail is heavily used by CHERI researchers to formalize different architectural implementation, and hence contains an existing formalization of both

32-bit and 64-bit CHERI-RISC-V [132], on top of which we implemented extensions for CHERI-TrEE.

Implementation of CHERI-TrEE in Sail

The Sail implementation of CHERI-TrEE is complete; all previously described architectural implementation details are included. The implementation inherits the characteristics of Sail-CHERI-RISC-V: both RV32 and RV64 implementations are supported in modular fashion, the model does not support split register banks (instead, a single register bank is shared for both capabilities and integers), and the use of compressed CHERI Concentrate capabilities is mandatory. For our purposes, we were most interested in the emulation functionality. Because generating an assembler from a Sail specification is not yet supported, we also extended CHERI's fork of the LLVM compiler to support the newly added instructions. This allowed using the LLVM toolchain to compile assembly files into well-formed ELF-binaries that can be executed by Sail's C-emulator. We developed basic unit tests to check functional correctness of each new instruction, as well as a larger scenario test, and ran these on top of the C-emulator.

4.4.2 Proteus RISC-V CPU framework

To verify and evaluate our design, we implemented the primitives from Section 4.3.3 in hardware. We based our implementation on *Proteus*, our open-source RISC-V processor designed with configurability and extensibility as its main goals.

Proteus

Heavily inspired by VexRiscv [119], Proteus is designed in SpinalHDL [118], a Scala-based Hardware Description Language (HDL). It is useful to think of SpinalHDL as an HDL code generator: Scala is used to generate an HDL description at runtime (using primitives provided by the SpinalHDL library) and this is then converted to either Verilog or VHDL. Designs can be simulated using any HDL simulator or synthesized for FPGAs.

Proteus uses a plugin architecture to configure and extend processors. At its core, Proteus provides pipeline stages and the ability to pass values from one stage to the next. The logic contained in stages is not fixed but can be configured through *plugins*. This allows for a lot of flexibility in, for example, the number of pipeline stages and the supported features (e.g., the RISC-V "M" extension is an optional plugin). Concretely, our implementation uses a classic 5-stage RISC pipeline consisting of an

IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory Access), and WB (register Write Back) stage. The plugin system also enables the development of custom extensions without having to alter the core implementation files. Currently, Proteus provides plugins to implement in-order pipelines with support for RV32IM and machine mode. A powerful feature of Proteus is the concept of *services*, which allows plugins to provide customization points to other plugins. For example, the plugin that implements load and store instructions offers an interface to intercept the generated addresses. This is used by one of our CHERI plugins to perform the necessary permission checks without having to modify the existing plugin.

CHERI Proteus

We extended Proteus with plugins implementing most of version 8 of the CHERI-RISC-V 32-bit specification [175]. It implements a split register file, capability manipulation, implicit memory access through *DDC* (the *Default Data Capability* register [175]) and *PCC* (the *Program Counter Capability* register [175]), explicit memory access through capabilities, exception handling, and storing capabilities in memory. To track capabilities in memory, an on-chip tag table is maintained that stores one tag-bit per capability-aligned word.

CHERI Proteus is not compliant with the CHERI specification in terms of the memory representation of capabilities: Instead of compressed capabilities (which are mandatory for RISC-V), Proteus uses a full-precision representation, which is easier to implement but causes in-memory capabilities to use 128 bits instead of 64 bits for CHERI-RISC-V 32 bit. This has the advantage that we have ample otype-space for enclave seals (cfr. Section 4.3.3). This also means we currently cannot take full advantage of the CHERI compiler toolchain, but are limited to the use of the assembler. However, this is sufficient to run complex software examples (see Section 4.5.1).

4.4.3 Implementing CHERI-TrEE on Proteus

To implement our detailed design (Section 4.3.3), a number of components have to be added to the CPU core. To store the TCB state of enclaves, we add a table (`EidTable`), in which each entry stores the *eid* and identity (*I*) of an enclave. Entries also keep track of the current state of the enclave, which can be *allocated* (`EInitCode` has been called but `EInitData` not yet), *ready* (enclave has been fully initialized), or *empty* (the entry does not contain any enclave information). The table provides an interface to allocate a new entry (which increases the *eid* counter, see Section 4.3.3), retrieve an existing entry (used by `EInitData`), and to finalize an entry by storing *I*. When searching, `EidTable` iterates through the entries one by one. Note that the size of the table is parameterized and implemented as a plugin, which allows us to easily create an alternative implementation (e.g., one that stores entries in memory) without changing the rest of the design. To calculate enclave identities, we use a SHA256 implementation. from the SpinalCrypto library [146].

As `EInitData` needs access to the memory bus, we decided to implement all new instructions in the MEM stage of the 5-stage pipeline. All instructions are based on state machines of varying complexity. `EInitCode` simply asks `EidTable` to allocate a new entry and, if successful, seals its input capability with o_entry (Section 4.3.3). `EInitData` is by far the most complex instruction to implement, as it needs to scan all registers and memory for overlapping capabilities. After performing sanity checks on its inputs (e.g., whether the code capability corresponds to an entry in `EidTable`), it starts by requesting exclusive access to the pipeline. This operation makes sure that the requesting instruction is the only one executing in the pipeline by flushing or invalidating stages containing other instructions. This is necessary to correctly perform the register scanning as otherwise, copies of registers might be available in pipeline stages. While having exclusive access, the state machine iterates over all general purpose and special capability registers to verify that there are no overlaps with the code or data capability of the enclave.

To scan memory, we iterate over all valid capabilities by using one of the services of the CHERI plugins. This service accesses the tag storage to produce addresses of valid capabilities without the need to perform memory loads. Once we encounter a capability, we load it and perform the overlap check. Having verified that there are no overlapping capabilities, the hash of the code section is calculated by loading its contents word-by-word and providing it to the SHA256 block. The result is then stored in `EidTable` and the corresponding entry is marked as *ready*. Finally, the sealing capability is created and stored at the first address of the data section.

`EStoreId` verifies that its input capability allows storing a full hash and iterates over the entries of `EidTable` to find the entry corresponding to the given seal. If it finds one and it is marked as *ready*, the hash is written to memory.

4.4.4 Implementation on ARM Morello

We implemented CHERI-TrEE on the first commercial CHERI-enabled processor, ARM Morello [11], using the FVP simulator. While for Proteus we realized functionality as an ISA extension, here we show that it is also feasible to implement CHERI-TrEE in (low-level) software: On Morello, one can use the Exception Level 3 (EL3) monitor or the EL2 hypervisor to pause (at least in a single-core scenario) an OS running at EL1 to perform the CHERI-TrEE operations, including the memory sweep. We built a corresponding small trusted hypervisor at EL2 to implement CHERI-TrEE operations, defining hypervisor calls to trigger an exception to EL2. The inputs are held in the first two registers on entry to EL2, and then passed on as parameters in the handler code.

For the register sweep, all EL1 register values are saved on the EL2 stack following a hypervisor call. A capability pointing to the stack where register values reside is passed through to the exception handler function so that the sweep is performed on the state of EL1 registers. The registers are restored on return, except for the overwritten return capability/value. Other EL1 registers are either maintained or transferred to other system registers by the CPU, e.g., the EL1 program counter, which is also checked as part of the sweep. Our implementation is arranged to closely align with the Proteus implementation for ease of testing and comparison. Differences in hardware however inevitably lead to deviations, e.g., the `BRS` instruction on Morello (`CInvoke` on Proteus) has a lower bound on the otype, forcing the eid counter to start at 1 rather than 0.

An additional challenge is presented by the support of virtual memory in ARM Morello: a malicious OS at EL1 could give up ownership of an enclave capability, but then re-map another virtual address (and hence different capability) to again point to that enclave's physical memory. This issue can be overcome by either fully blocking changes to the page tables by EL1 (while the enclave is executing) and/or through appropriate MMU memory permissions and register access restrictions set at EL2. This is included in the prototype by setting bits in the hypervisor control register to block manipulation of EL1 system registers that could cause MMU changes. As the hypervisor is in control of the set up of the page tables for EL1, it is also possible to make the page table area in memory read only. However, both solutions preclude integrating the design with a rich OS using virtual memory. Yet, we note that the CHERI-TrEE design would be a promising candidate to use as the basis for a trusted OS inside ARM Trustzone, similar to the approach taken by Komodo [54], but with the shown benefits of a capability-based system.

4.5 Evaluation

Next, we evaluate the performance and (hardware) implementation costs of CHERI-TrEE on different platforms, our Proteus (Sections 4.5.1 and 4.5.2) and ARM Morello (Section 4.5.3).

4.5.1 Performance on Proteus

To asses the performance impact of our proposed extension on software, we conducted a number of micro and macro benchmarks. The micro benchmarks quantify the cost of individual instructions, while the macro benchmark measures the overhead on a full application consisting of multiple enclaves. Note that the new instructions `EInitCode` and `EInitData` are only used once at enclave initialization time.

Micro benchmarks

As EInitCode simply allocates a new entry in EidTable, its runtime only depends on the current occupation of that table. We measured that executing it with an empty table takes 4 cycles, and one extra cycle is needed for every non-empty entry at the beginning of the table (i.e., the second execution takes 5 cycles). For EInitData, there is a fixed and a variable cost. The fixed part consists of getting exclusive access over the pipeline and scanning registers for overlapping capabilities. Exclusive access requires at most one cycle (because EInitData is implemented in the MEM stage, only the instruction in the WB stage needs to be completed), while one cycle per register is needed for the scan. The variable cost consists of multiple parts: First, the entry corresponding to the code capability needs to be looked-up in EidTable. Second, memory needs to be scanned for overlapping capabilities, which obviously depends on the memory size. As mentioned in Section 4.4.3, tag-bits are scanned at a rate of one per cycle, so to get all addresses containing capabilities, the amount of cycles needed is the memory size divided by the size of a capability (128 bits). Then, for each capability, a capability load needs to be performed, which takes 4 cycles (four 32-bit loads, each with a single cycle latency). The last variable part is creating the enclave's identity by hashing its code section. Table 4.1 shows the measured performance of EInitData. The execution time of EStoreId is mostly fixed and depends on the size of the hash. The only variable part is the index of the hash in EidTable, as it has to be searched for the correct hash. For the initial enclave, we measured the execution time to be 19 cycles.

	Code size (B)		
RAM state	256	512	1024
128+0	8811	9271	10 191
128+100	9211	9671	10 591
256+0	17 003	17 463	18 383
256+100	17 403	17 863	18 783

Table 4.1: Cycle count of EInitData. RAM state is size (in KiB) plus number of valid capabilities in memory.

Macro benchmark

To measure the performance of a more realistic application, we built a scenario where two enclaves communicate with each other. A "sensor enclave" provides an entry point to read the (encrypted and signed) value of a sensor. The "client enclave" attests the sensor, gets a sensor reading, and performs some operation on the value.

To correctly initialize enclaves, untrusted code starts by setting up a memory allocator for uniquely-owned memory. Memory is split in two parts: the lower part as "normal" memory and the upper part as "uniquely-owned" memory. All special capability registers (e.g., DDC and PCC) are configured to only overlap with normal memory and the only capability for the uniquely-owned part is given to the allocator. When uniquely-owned memory is allocated, this capability is split to ensure that only a single capability to the allocation exists.

Untrusted code continues by relocating the code of both enclaves in a uniquely-owned region and allocating data sections for them. Then, it registers these (using `EInitCode` and `EInitData`) and invokes their initialization routines as described in Section 4.3.3. After sanity checks, the stack pointer is initialized and stored in the data section. Enclaves also store their own entry point capabilities to pass them as return pointers when calling another enclave. As a last step, the public parts of the sign and encryption seals are derived from the seal stored by `EInitData` and returned.

To support multiple entry points, enclaves dispatch based on the value in an agreed-upon register. One specific value is used as a "return" entry point, which is invoked when returning from a call with the return address popped from the stack. The next step is for the client enclave to attest the sensor: it provides an entry point to receive the entry capabilities and public seals of the sensor from untrusted code. `EStoreId` is used to retrieve the actual identity of the sensor and compare it with the expected identity stored in the code section of the client. If the identity is correct, the client stores the entry capabilities and seals of the sensor.

The actual application starts by calling the "use sensor" entry point of the client. The client creates a buffer in its data section, storing the nonce and the public part of its encryption seal in this buffer and leaving space for the return value. It then encrypts the capability to this buffer using the sensor's encryption seal. It then stores the return address on its stack and calls the sensor, providing its own entry capabilities as return pointer. The sensor decrypts the input capability and stores a sensor reading in the buffer. Note that because our current implementation does not support the `IsUnique` instruction yet, we cannot verify exclusive access to an MMIO device. Therefore, we simulate such a device by storing an arbitrary value as sensor reading. The sensor then loads the encryption seal of the caller to encrypt the capability to the buffer. This encrypted capability is stored and another capability pointing to this location is sealed using the sensor's signing seal and returned to the caller. On return, the client verifies the signature and decrypts the returned capability. Then, it checks if the nonce matches the one passed to the sensor and loads and processes (currently simply doubling) the returned reading. The processed value is then stored in the input buffer to return to the caller.

We ran this scenario on a cycle-accurate Verilator-based [166] simulation of CHERI-TrEE with 128KiB of RAM. We used a bare-metal system without OS or scheduler

Step	Runtime (cycles)
Init client	9773
Init sensor	9163
Attestation	384
Sensor use	471

Table 4.2: Execution times in our macro benchmark: The "init" steps include `EInitCode`, `EInitData`, invoking enclave initialization code and preparation (e.g., clearing registers). "Sensor use" includes the round-trip from client to sensor enclave.

so that our measurements are completely deterministic. The code size of the client enclave is 656 bytes, while that of the sensor enclave is 336 bytes. The execution times of the different steps of our scenario are shown in Table 4.2. For comparison, to execute an "unprotected" version of our scenario (where no initialization or attestation is performed and arguments are passed without sealing), we obtained a cycle count of 79. The initialization of enclaves makes up the bulk of the execution time. The attestation is cheap, but the overall overhead of the sensor use is large compared to the unprotected case. However, almost no useful computation is performed in this scenario, so the relative overhead reduces in more realistic use cases.

4.5.2 Hardware implementation of Proteus

Processor (128 KiB memory)	Area occupation					Operating frequency (MHz)	Dynamic power (mW)
	LUTs	Flip-flops	BRAMs	DSPs	CLBs		
Proteus	3054 (1.1%)	1663 (0.3%)	32 (3.5%)	-	694 (2.03%)	180	40
CHERI Proteus	8059 (2.94%)	3915 (0.7%)	32 (3.5%)	-	1298 (3.79%)	70	23
CHERI-TrEE	12 806 (4.7%)	7514 (1.4%)	32 (3.5%)	-	2385 (6.96%)	70	69

Table 4.3: Implementation results for the Proteus processor and its variants on the Zynq UltraScale+ XCZU9EG-2FFVB1156 FPGA board. All tests with 128 KiB memory. Percentages indicate area usage relative to total FPGA size.

To quantify the performance and resource requirements of the Proteus processor, we implemented and ran it using the Xilinx Vivado tools [184] on a Zynq UltraScale+ XCZU9EG FPGA [185]. We extended the Verilog file generated by SpinalHDL with support circuitry (e.g., for clocking) and set up constraints and pin connections for our FPGA board. We considered all variants: Proteus (without capabilities); the capability-enabled variant CHERI Proteus; and the EES-enabled variant that also includes capabilities, CHERI-TrEE, cfr. Section 4.3.2. We analyzed the results regarding commonly used key metrics: area occupation, operating frequency, and power consumption. Table 4.3 presents the results. In Section 4.A, we also compare

Proteus and variants to the CHERI Piccolo processor developed by the Cambridge team [173].

First, we evaluated the impact of adding CHERI and CHERI-TrEE functionality, using 128 kB memory in all tests. As can be seen in Table 4.3, the area occupation increases with the complexity of the processor, i.e., by adding capabilities and CHERI-TrEE functionality. Compared to Proteus, CHERI Proteus uses more hardware primitives, such as Look-Up Tables (LUTs) and flip-flops, occupying ≈ 1.87× more FPGA area in terms of Configurable Logic Blocks (CLBs). This increase in area occupation can be mainly attributed to CHERI capabilities and tag-bits, which are implemented with distributed RAM and, thus, realized as LUTs. Similarly, CHERI-TrEE again uses more hardware primitives, occupying ≈ 3.44× and ≈ 1.84× more CLBs than Proteus and its CHERI variant, respectively. In addition to the ISA extensions, this increase can be mainly attributed to the SHA256 block. All variants use 32 Block Random Access Memories (BRAMs) due to the identical data memory size of 128 KiB.

Regarding performance, the CHERI Proteus processor and its CHERI-TrEE variant decrease the maximum clock frequency by ≈ 2.57× compared to Proteus. As expected, adding capabilities substantially increased the processor's complexity. This is due to the CHERI trap and exception handling circuits deployed with logical primitives on the critical path. We note that we did not specifically optimize the hardware design for maximum clock frequency; thus, substantial improvements are likely possible, e.g., by adding register stages in the critical path. Finally, CHERI Proteus has a 74% lower dynamic power consumption than Proteus as it runs at a lower clock frequency. Conversely, due to its higher resource usage, CHERI-TrEE consumes 69 mW, i.e., 72.5% more.

In summary, our Proteus core and its capability/EES variants scale well and largely independently of the memory size, requiring ≤ 7% of the total area on our FPGA. We also deployed the CHERI Proteus and CHERI-TrEE variants on the low-end Arty A7-35T XC7A35TICSG324-1L FPGA. CHERI Proteus used 28.23% of the total available area, while CHERI-TrEE occupied 51.57%, which shows that our design can also be implemented on small FPGAs.

4.5.3 Performance on ARM Morello

The predominant cost of `EInitData` is the memory sweep. On the Morello FVP, an EL1 memory sweep by the EL2 hypervisor took about 2.2 billion cycles to sweep a 1 GB block of DRAM, corresponding to 1.1 s on a single core running at 2 GHz. This includes overlap checks of 233 capabilities that were detected. A "clean" memory sweep (no capabilities) is only marginally faster than this, because if the tag-bit is not set, there is no need for an overlap check. Given that the sweep is performed on 16 bytes at a time (capabilities are aligned to 16-byte boundaries), this corresponds to approx.

33 cycles to check each capability memory location. The `EInitCode` and `EStoreID` operations take significantly fewer clock cycles, and their runtime is proportional to the number of table entries searched. For a look-up that hits the first entry, the `EInitCode` operation takes 471 cycles and `EStoreID` takes 1797 cycles.

4.6 Further extensions and future work

Being a research prototype, CHERI-TrEE allows for various extensions and follow-up work.

Going from local to remote

CHERI-TrEE currently only supports local attestation. To support remote attestation and secure remote communication, appropriate cryptographic functionality is required. For example, we could allow each enclave to govern its own keys for remote attestation, possibly in the form of a special token capability representing the key, and add a primitive to perform (e.g., symmetric) encryption and decryption given an appropriate capability and a key. We expect to be able to reuse a lot of the key distribution infrastructure of e.g., Sancus.

Added flexibility motivates verification

A disadvantage of the flexibility highlighted in Section 4.3.4 is that it becomes more difficult to gain assurance over the security of the system. To clarify, the size of the TCB in our proposal is not fundamentally larger than in other systems like SGX or Sancus. However, the flexible nature of our system simply creates more potential pitfalls for enclave developers (although, as we discussed, capabilities help avoid some API-based vulnerabilities). We believe that verifying the security of such implementations will be an interesting challenge to address using formal methods, either using protocol verification or in a proof assistant. Enclaves in general are already rewarding targets for verification, since they rely on such a small TCB for their proper functioning (no language runtime or operating system is trusted). For relatively small enclaved applications (no library operating systems inside the enclave) on RISC processors (allowing simple enclave runtimes because of the limited amount of hidden processor state), obtaining full-system security properties for enclaved code should already be an achievable goal, and scaling these results is interesting future work.

Supporting virtual memory

Related to the issues discussed in Section 4.4.4, integration of CHERI-TrEE with virtual memory in a more principled way is interesting, as it would allow CHERI-TrEE to function with a rich OS using virtual memory. One idea is to have physically-addressed capabilities, which bypass the MMU and carry permission over physical memory directly. If the operating system cannot map virtual memory to ranges covered by physically-addressed capabilities, enclaves can be secured in the presence of virtual memory. Another idea is to place restrictions on page table manipulation (by e.g., using capabilities rather than integer addresses as page table entries).

4.7 Related Work

Both trusted computing and capability systems have a very rich history going back many decades. For trusted computing, an excellent survey is given by Parno *et al.* [120], while Maene *et al.* [98] provide a more up-to-date survey of hardware-supported systems. For capability systems, Levy [94] covers early systems, and Watson *et al.*'s paper on CHERI's support for compartmentalization [176] discusses more recent systems.

Our work is most related to the class of systems we called *enclaved execution systems*. Flicker [102] was the first system to propose the idea of fine-grained attestation and secure execution of small pieces of code, isolated even from malicious system software. Flicker and other early systems [101, 147] were implemented as a small hypervisor that used the late launch feature of Intel/AMD processors to start a isolated VM containing the enclave. Later systems relied on hardware extensions to avoid the use of a hypervisor, further reduce the TCB, and increase security [38, 113]. Intel SGX [37] is a commercial implementation of full support for enclaves. ARM Trustzone [8] initially only supports a single secure world, which then runs a separate operating system to support multiple trusted applications in parallel. The third major commercial system, AMD SEV, isolates complete virtual machines (rather than small enclaves) from the untrusted OS [9].

The mechanisms of enclaved execution are complex, and both research prototypes for enclaved execution as well as commercial systems have undergone revisions: For instance, Intel SGX2 adds support for larger enclave sizes and enclave dynamic memory management. Sancus 2.0 [114] adds support for confidential loading. TrustLite [88] introduces an execution-aware memory protection unit to support more flexible allocation of memory to embedded enclaves. Tytan [25] adds support for real-time guarantees. TIMBER-V [178] enables memory sharing for fine-grained enclaves by using a tagged memory architecture. By decomposing enclaved execution into simpler primitives that can then be combined in software, our work enable more flexible implementations of extensions and new features. We are not the first to observe the complexity of enclaved execution, or the usefulness of making it more extensible and configurable: Keystone [90] avoids the fixed set of trade-offs in existing systems and improves customization by offering a framework for building enclaved execution systems.

Very recently, Elasticlave [186] proposed a significantly more flexible memory model for enclaves, and implements it on top of Keystone. Elasticlave enforces a novel discretionary access control model for memory regions, designed to make common data sharing patterns efficient. An important difference with our approach is that, rather than designing a new access control model, we can *reuse* the capability memory access control model as used in non-enclave settings. Sanctum [38] shares our goal of identifying minimal hardware extensions or modifications and then combining these in software but it builds on page table-based isolation rather than capabilities as the base platform. Komodo [54] also identifies minimal hardware requirements and then implements enclave management instructions in a small trusted software monitor that is formally verified. The prototype is implemented on top of ARM Trustzone. SANCTUARY [26] uses ARM Trustzone together with the address-space Ccontroller present in some modern ARM processors to dynamically construct user-space enclaves. Like Sanctum, Komodo and SANCTUARY do not consider capability-based hardware primitives to build on.

Most closely related to our work is the concurrent work on the CheriOS capability operating system described in Esswood's PhD thesis [53]. The goal of CheriOS is to achieve high performance in the presence of a strong adversary where not even the OS is trusted. The CheriOS microkernel is built on a trusted firmware nanokernel, which provides the security properties required to implement an EES. As in our work, remote attestation is out of scope. The nanokernel supports interrupt handling and a limited form of single-address-space virtual memory, where the OS cannot alter page tables.

To obtain unique ownership of memory, the nanokernel offers *reservations*; sealed capabilities that represent a right to uniquely allocate a region of memory, and are governed by a state machine. Once memory is allocated through one reservation, all other reservations lose the right to allocate it, until the memory is revoked. In this way, the owner of the memory can be assured of their unique ownership. Non-allocated Reservations can be used to create a type of enclaves, called *foundations*, similar to our design. Foundations are represented using a *foundation ID*, which is essentially a read-only hash of the enclave's code section. Rather than reusing sealed pairs for the entry point, the nanokernel creates an *entry token*. Later invocations can be set up to use CHERI's `CInvoke` mechanism to jump to the enclave directly. The foundation owns an *authority token*, an object capability analog to our sealing capabilities, which is tied to the Foundation ID and can be used to perform both symmetric and asymmetric signing and encryption through the nanokernel, as opposed to our sealed capabilities. In conclusion, both approaches are sufficiently flexible to allow for growing/nested enclaves and memory sharing between enclaves. The work of Esswood *et al.* involves a larger TCB and more software-level abstractions, leading to overhead, but allows for greater flexibility with respect to revocation and a larger sealing space. The exact implications of these trade-offs need to be investigated further.

4.8 Conclusion

In this paper, we investigated how enclaves can be built on a capability machine like CHERI. Implementing enclaves without duplicating existing functionality was only possible by decomposing the concept of an EES into a set of orthogonal features. This decomposition made it clear that we can reuse CHERI's existing features and only need to add a limited amount of new features to obtain an expressive EES that performs well in many respects (see Section 4.5). Even better, the resulting design is more flexible in important ways (growing/nested/non-contiguous enclaves, sharing memory, two-way sandboxing, dynamic entry points, etc.). In addition to the design itself, we believe our decomposition of EESs is useful to analyse the design space and inform future designs.

4.A Appendix: Comparison to Piccolo

To compare our design to other RISC-V cores, the only other 32-bit processor implementation providing CHERI capabilities is—to our knowledge—the Bluespec Piccolo core. Table 4.4 details the main modules of each core.

The non-capability Piccolo is developed by Bluespec [23], while the CHERI Piccolo variant is developed by Cambridge University [173], both designed in a high-level hardware description language, Bluespec HDL [154]. Note that neither Piccolo variant implements extensions for trusted execution. Therefore, we can mainly compare the different CHERI implementations. Secondly, Piccolo is a commercial-grade processor with additional functionality, while Proteus is currently a research design. Nonetheless, we implemented both Piccolo and CHERI Piccolo on the same FPGA as Proteus and used 128 kB memory in all tests. As Proteus is an RV32IM, we configured the Piccolo variants to the same architecture (by default Piccolo is an RV32ACIMU).

Comparing CHERI Proteus and CHERI Piccolo, our processor uses ≈ 3.31×, ≈ 2.95×, and ≈ 2.31× fewer LUTs, flip-flops, and BRAMs, respectively, while not requiring dedicated DSP blocks. Different from CHERI Piccolo, CHERI Proteus does not implement CHERI compressed capabilities and uses BRAMs for the instruction and

Processor (128 kB memory)		Area Occupation					Operating frequency (MHz)	Dynamic power (mW)
		LUTs	Flip-flops	BRAMs	DSPs	CLBs		
Proteus	Machine Timers module	32	128	-	-	48	180	40
	Pipeline module	2867	1410	-		641		
	Data and Instruction memories	-	-	32		-		
	Total	3054 (1.1%)	1663 (0.3%)	32 (3.5%)		694 (2.03%)		
CHERI Proteus	Machine Timers module	80	128	-	-	37	70	23
	Pipeline module	7797	3662	-		1260		
	Capability Register File	161	-	-		21		
	Data and Instruction memories	-	-	32		-		
	Total	8059 (2.94%)	3915 (0.7%)	32 (3.5%)		1298 (3.79%)		
CHERI-TrEE	Machine Timers module	80	128	-	-	40	70	69
	Pipeline module	12 510	7261	-		2353		
	Capability Register File	160	-	-		20		
	SHA module	3422	3193	-		839		
	Data and Instruction memories	-	-	32		-		
	Total	12 806 (4.7%)	7514 (1.4%)	32 (3.5%)		2385 (6.96%)		
Bluespec Piccolo [23]	CPU module	4807	2583	3	15	835	40	68
	Debug module	292	405	-	-	183		
	PLIC+CLINT modules	1810	1194	-	-	504		
	Data and Instruction memories	-	-	33	-	-		
	Total	10 152 (3.7%)	7986 (1.4%)	36 (3.9%)	15 (0.6%)	1942 (5.67%)		
CHERI Piccolo [173]	CPU module	18 347	6464	5	15	3471	120	194
	Debug module	471	402	-	-	176		
	PLIC+CLINT modules	1840	1146	-	-	467		
	Tag Controller module	4349	2124	36	-	966		
	Data and Instruction memories	-	-	33	-	-		
	Total	26 685 (9.7%)	11 533 (2.1%)	74 (8.1%)	15 (0.6%)	4865 (14.2%)		

Table 4.4: Implementation results for the Proteus and Piccolo processors and their variants on the Zynq UltraScale+ XCZU9EG-2FFVB1156 FPGA, depicting the main modules within each Core.

data memory, using considerably fewer resources. The dynamic power consumption is ≈ 8.43× less than CHERI Piccolo. Similarly, for the non-capability versions, Proteus occupies ≈ 3.32×, ≈ 4.80×, and ≈ 1.12× fewer LUTs, flip-flops, and BRAMs than Bluespec Piccolo. The dynamic power consumption is 1.7× less.

The higher resource usage of both Piccolo variants compared to Proteus can be partially attributed to higher circuit complexity due to additional components, e.g., debug circuitry and interrupt controllers. Besides, CHERI Piccolo uses BRAMs to implement its (relatively large) Tag Controller module. Finally, as mentioned, CHERI Proteus does not currently implement compressed capabilities.

Regarding clock frequency, Proteus reaches 180 MHz, ≈ 1.29× faster than Bluespec Piccolo. However, CHERI Proteus is ≈ 1.71× slower than CHERI Piccolo, which could be due to the fact that Piccolo is already optimized for real-world deployment.

Besides, considering only the actual CPU module of Piccolo cores detailed in Table 4.4, our Proteus processors use significantly fewer resources than Piccolo: the non-capability Proteus uses ≈ 1.2× fewer CLBs than Bluespec Piccolo, while CHERI Proteus uses ≈ 2.67× fewer CLBs than its CHERI variant. We note that CHERI-TrEE also uses fewer resources than CHERI Piccolo despite the additional EES features. In summary, this comparison shows that even though Proteus is currently a research prototype and has not gone through extensive optimization cycles, it offers a smaller size and adequate performance when compared to the commercially maintained Piccolo. In addition, our modular design lends itself to the easy addition of functionality at an acceptable cost, as evidenced by the CHERI-TrEE implementation on Proteus.

Chapter 5

Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code

Publication Data

This chapter contains the following paper about secure compilation from verified C-like code to C-like code with support for linear capabilities:

> Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Journal Of Functional Programming*, 31(PII S0956796821000022):1–55, Mar. 2021

This paper is an extended version of the following conference paper:

> Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, 3(ICFP), 2019

Full details can be found in the following technical report:

> Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code - technical appendix including proofs and details. 2020. URL: https://soft.vub.ac.be/~dodevrie/seplogic-lincaps-tr20201130.pdf

Most of the proof labor and writing work were done by me, with indispensable guidance from Dominique and Frank, especially at the start of the project.

Compared to the published paper, the version included in this thesis has the following additions:

- Introduction: small clarification concerning the efficiency of linear capabilities
- Section 5.5.2: new section that illustrates, through examples, the guarantees we get from proving our compiler fully abstract.
- Section 5.8.2: additional discussion of how we could potentially support a richer separation logic within a single trust boundary.
- Section 5.9: comment about the use of coarse-grained encapsulation, as opposed to our fine-grained use of capabilities.

Abstract

Separation logic is a powerful program logic for the static modular verification of imperative programs. However, *dynamic* checking of separation logic contracts on the boundaries between verified and untrusted modules is hard, because it requires one to enforce (among other things) that outcalls from a verified to an untrusted module do not access memory resources currently owned by the verified module.

This paper proposes an approach to dynamic contract checking by relying on support for capabilities, a well-studied form of unforgeable memory pointers that enables fine-grained, efficient memory access control. More specifically, we rely on a form of capabilities called *linear* capabilities for which the hardware enforces that they cannot be copied.

We formalize our approach as a fully abstract compiler from a statically verified source language to an unverified target language with support for linear capabilities. The key insight behind our compiler is that memory resources described by spatial separation logic predicates can be represented at run time by linear capabilities. The compiler is *separation-logic-proof-directed*: it uses the separation logic proof of the source program to determine how memory accesses in the source program should be compiled to linear capability accesses in the target program.

The full abstraction property of the compiler essentially guarantees that compiled verified modules can interact with untrusted target language modules as if they were compiled from verified code as well.

This article is an extended version of one that was presented at ICFP 2019 [162].

5.1 Introduction

Separation logic is the basis for tools that support sound, modular verification of C programs, such as VeriFast [72]. However, for such verification to be sound for a whole program, *all* modules of the program have to be verified [7].

In this paper, we are concerned with scenarios where verified code interacts with untrusted code (for example, when installing plugins from the internet). Our goal is to compile the verified code securely, i.e. in such a way that we can preserve the guarantees obtained from verification, even under this interaction with untrusted code. To achieve this, the compiler has to dynamically enforce separation logic contracts on the boundary between verified and untrusted code.

As a concrete example of our approach, consider a separation-logic-verified video player that runs locally on a user's computer and allows for the installation of untrusted and unverified plug-ins to extend its functionality. An example plug-in would be a codec (*coder-decoder*), that includes support for the decompression of specific video encodings before display. The separation logic contract for the decompression function of this plug-in could, for instance, provide it access to the buffer where the compressed video is stored, but forbid it from retaining references to this buffer afterwards. The contract for the decompression function might then informally (we elided functional assertions) look as follows:

```
void decompress(char* b, format f)
//@pre    b ↦ contents_pre * ...
//@post   b ↦ contents_post * ...
```

The decompress function receives a pointer to the buffer b, and is supposed to decompress b using the proper codec for format f. In the precondition //@pre, decompress receives ownership of the buffer's contents (represented by the points-to chunk $\mapsto$), so that it can perform decompression in-place (assuming sufficient buffer size). In the postcondition //@post, decompress returns ownership over the contents of this buffer, and should consequently lose its access to it. However, this revocation of access is hard to enforce dynamically: an unverified plug-in could e.g. copy and store the reference to b and use it to freely read and write to it later, even after returning control. Plug-ins can violate their contracts in many other ways: they can deviate from their specified behavior while they legitimately hold references to the internal state (e.g. decompress could use the wrong format, or do nothing); they might read or write outside the intended ranges of the references they are provided with (e.g. perform a buffer overread or -write outside of b's bounds), or might return incorrect values. In the current state of the art, separation-logic verification guarantees are not enforced at run time for partially verified programs.

To perform dynamic checking of separation logic contracts *efficiently*, some form of hardware support for memory protection is required. Agten et al. [7] proposed an approach for dynamic checking of contracts based on a hardware protection primitive known as *protected module architectures* [113, 148, 37]. However, they only provide run time preservation of integrity guarantees, not of confidentiality, meaning that non-verified adversaries could still read data they should not contractually be allowed to read. Hence, they do not ensure *full abstraction*: a formal property that is often used to define secure compilation [1]. This property requires that attacker code interacting with the compiled code in the target language should not have more power than arbitrary verified code interacting with the verified source code.

The main contribution of this paper is the development of a *fully-abstract* compiler that dynamically enforces separation logic contracts by relying on another kind of hardware support. Our approach relies on support for *capabilities*: a well-studied form of unforgeable memory pointers that are in essence regular pointers, enhanced with a field containing privileges (read, write, execute, ...) and fields describing a memory range these privileges can be exerted on. Capabilities allow for fine-grained, efficient memory access control, and are implemented in special processors called *capability machines* [see 94, for an overview]. The CHERI processor [176] is an example of a recent design for a capability machine.

More specifically, we rely on a form of capabilities called *linear capabilities*. Linear capabilities are specially treated by the hardware to ensure that they can never be copied. They are related to, but different from, CHERI's local capabilities (which can, essentially, only be stored in registers or on the stack, not in memory). Although we believe linear capabilities are an elegant and efficient target feature for implementing a fully abstract compiler from verified C code, an implementation of linear capabilities needs to overcome certain hurdles related to concurrency and implementation in hardware, as we already mentioned in e.g. Section 4.3.2 [144]. For this reason, an efficient implementation of linear capabilities has not yet been demonstrated. However, we believe that given sufficiently appealing use cases, hardware designers would be incentivized to study and hopefully resolve these implementation obstacles. Skorstengaard et al. [144] have previously used linear capabilities in the secure calling convention `StkTokens`, and an early design for their implementation in CHERI is given in the latest CHERI ISA Spec [175]. Our compiler only requires basic linear capabilities with read/write permissions.

The key insight of our approach is that memory resources (described by spatial separation logic predicates) can be represented at run time as linear capabilities. Hence, transferring ownership of memory resources to another module on function call or return can be compiled to passing the corresponding linear capabilities as additional parameters and/or return values. Compiler-generated stubs, on the boundaries between verified and unverified code, can then dynamically check separation logic contracts. However, those linear capabilities cannot simply be substituted for regular pointers in existing programs, as they behave fundamentally differently (because of

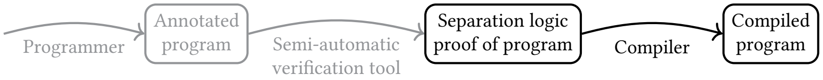

Programmer
Annotated program
Semi-automatic verification tool
Separation logic proof of program
Compiler
Compiled program

Figure 5.1: A usage model of our compiler.

their hardware-enforced non-duplicability). We can pass them in addition to regular pointers, but then the difficulty is that every memory access in the original program (through a regular pointer) needs to happen through an appropriate linear capability in the compiled program. Since those linear capabilities are not necessarily in one-to-one correspondence with the regular pointers that the program works with, it is not clear how the compiler can decide which one to use.

However, for verified programs, this information is apparent from the separation logic proof of verified code. In that proof, every memory access is justified using a single memory resource, and by carefully tracking the capability corresponding to every such resource, we can decide which capability to use. Hence, our compiler is *separation-logic-proof-directed*: it requires not simply the raw source code, but a separation logic proof of this code as input, and uses the information in the proof to generate correct and secure target code. Although it does not matter to our compiler where this proof comes from, we envision scenarios as depicted in Figure 5.1, where the programmer writes a program with contracts and minimal annotations, a semi-automatic verification tool like VeriFast [71] elaborates these into a full proof, and this proof is then passed to our compiler. To be clear, only the last (compilation) step is the topic of this paper, although we sometimes use VeriFast notations in our examples, for readability.

We clarify upfront that our goal is not gradual verification [7, 16], i.e. we do not support taking a large codebase, verifying parts of it and securely combining the verified and unverified parts. Instead, our compiled code can securely interact with arbitrary untrusted code (e.g., the downloaded plug-ins above). However, our compiled code will only interact correctly with other code that respects our calling convention: we only allow linking to untrusted code if it respects our compiled interfaces, i.e. sends and receives linear capabilities encoding memory resources as extra arguments and return values when handling incalls from or outcalls to our code.

Moreover, this paper contains but the first steps towards a practically applicable secure compilation scheme, since the power of the separation-logic-verified source language is limited. Concretely, the source language only consists of simple resources in the separation logic, has a simple type system and features restrictions on the separation logic contracts for calls to and from untrusted code. Perhaps most notably, our source language does not yet support a notion of C-like structures; it does not support any type of recursive data structures. Additionally, the separation logic does not support the abstract predicates that would be required to reason about such data structures. These restrictions and how to alleviate them will be further detailed in the future work section.

In summary, the contributions of this paper are:

- a novel approach to compile separation-logic-verified C code to linear-capability-enhanced unverified C code that dynamically checks the contracts at the boundaries of the verified code; we demonstrate our approach for an essential separation logic with array resources, and explain how the approach might be extended to more advanced logics;
- a formalization on a model of C, and a proof that our compiler is fully abstract;
- a new use-case for linear capabilities in capability machines like CHERI.

This paper is structured as follows. The compiler is illustrated in Section 5.2. Section 5.3 discusses the compiler's source and target language. Section 5.4 formally defines the separation logic rules and the full compiler. Section 5.5 formally defines full abstraction and discusses our proof approach, motivating the need for a new target-to-source transformation called the back-translation. This back-translation is illustrated by means of an example in Section 5.6. Having discussed both an example of compilation and back-translation, Section 5.7 zooms in on one part of the full abstraction proof, namely how we simulate source versus target code. Sections 5.8 and 5.9 respectively discuss future and related work. Section 5.10 concludes.

This paper is an extended version of one that was previously published and presented at ICFP 2019 [162]. The most important additions are:

- A better explanation of how linear capabilities work in the target language operational semantics (Section 5.3.3).
- The new Section 5.5.4 which explains the role played by two relations R and S that relate source and target code in the proof of full abstraction.
- A rewritten Section 5.6 explaining the back-translation in a more complete and more pedagogical way.
- Finally, a new Section 5.7 which further decomposes the aforementioned relations R and S, showcasing how their components respectively prove the correctness and security directions of full abstraction through simulation.

Nevertheless, in this paper, we have still omitted details and simplified inference rules to maintain readability and ease of understanding on multiple occasions. Interested readers can find full details and the entire full abstraction proof in a 120-page technical report submitted as supplementary material [164].

5.2 Compiler Illustration

Figure 5.2 illustrates the operation of our compiler with a trivial example in the C-like syntax (which includes separation logic annotations) we employ. As suggested in the introduction, the target language of the compiled code is actually again C-like and not assembly. The reason is that the compilation from C to assembly would require the (fully-abstract) compilation of many concepts (e.g. function calls, stack accesses) which are orthogonal to the topic of this paper and in at least one case already covered elsewhere [144].

The example contains the verified source function f, which performs an outcall to the context identity function g and afterwards sets the contents of pointer a from 0 to 1. Function f only knows g's separation logic contract and not its implementation.

To see how function declarations (including separation logic contracts) are compiled, we look at the function f (g is similar). The void function f takes a single argument: an integer pointer variable a. The precondition states that f receives a memory resource n to read and write pointer a in the heap, where a points to the single element array $[0]$. The resource can be seen as a (heap) permission. When f returns, its postcondition states that f hands back this permission, but a will contain the value 1. The verification of f proves that f upholds this contract.

Having introduced the example, the remainder of this section discusses how we

	Verified Component	Outcall Stub	Context Declaration

Source	void f(int* a)
	//@pre n: a ↦ [0]
	//@post n: a ↦ [1] {
	g(a);
	a[0] = 1
	}
			void g(int* a)
			//@pre n: a ↦ [0]
			//@post n: a ↦ [0]

Target	int* f(int*$_0$ a,int* n) {
	n = g$_{\text{stub}}$(a,n);
	n[0] = 1;
	return n
	}
		int* g$_{\text{stub}}$(int*$_0$ a,int* n) {
		n = g(a,n);
		guard(n != null);
		guard(length(n) == 1);
		guard(a == addr(n));
		guard(n[0] == 0);
		return n
		}
			int* g(int*$_0$ a,int* n)

Figure 5.2: Motivating example: a verified function f interacts with an untrusted context function g.

compile it in a *separation-logic-proof-directed* way. By *separation-logic-proof-directed compilation*, we mean that our compiler uses the separation logic proof of the source program to guide compilation, in order to ensure that the compiled code enforces the separation logic contracts. Concretely, our compiler combines three separate techniques to dynamically enforce all information contained in separation logic contracts:

1. The *shape* of separation logic resources (i.e. their linearity and bounds, but not their functional aspects such as contents) is enforced by reifying separation logic resources to linear capabilities.
2. The *contents* of these linear capabilities are updated (but *not* checked) in parallel with the resources they were reified from by having compiled source statements manipulate capabilities.
3. The actual *checking* of all aspects of contracts (i.e. the shape and contents of separation logic resources originating from adversaries, but also other non-spatial constraints) is performed by so-called checking functions or stubs at trust boundaries.

We now further detail these three aspects of proof-directedness in order.

First, we want to enforce the shape of separation logic resources at the target level, i.e. make sure that resources to access the heap are handled linearly and bounds-checked at the target level. The reason we care about the semantics of these resources is that e.g. resource n for address a determines whether f can access a's contents during the proof, not a itself. This fact is reflected in the compilation, where the heap resource n is reified into a target-level linear capability int$*$ n. Although separation logic resources exist only conceptually and are not represented in the source language, they are transformed into *real* target-level program variables, as concrete instantiations of the source-level permissions. The semantics of the linear capability n corresponds very well to the meaning of the heap resource n in the following ways:

- The fact that the reified resource int$*$ n is a *capability* ensures that anyone owning it cannot read outside of its intended bounds; the single location a in this case.
- The *linearity* of int$*$ n constitutes a guarantee to its owner, that they are the sole owner of the permission n to access a.

These reified resources are manipulated in parallel with the source-level resources and represent the otherwise erased separation logic proof guarantees. The precondition of f mentions that it receives the resource n at the start of execution. This resource is reified as an extra argument int$*$ n. The postcondition of f requires the resource n to be returned, so analogously, the reified version of n is returned in the compilation.

The source-language argument a, on the other hand, is compiled to a length-0 regular capability $\text{int}{*_0}\ a$. The length-0 capability type $\text{int}{*_0}$ denotes a non-linear capability of type int that cannot be dereferenced (this is in effect just an address). The reason $\text{int}{*_0}\ a$ is kept is for performing address operations and checks, as these non-spatial manipulations require no separation-logic resources.

Representing both addresses and resources at run time introduces a certain duplication of information in cases where addresses coincide with the bounds of the resources. This is the case, for example, in Figure 5.2, where the address a always coincides with the start address of the resource $a \mapsto [\_]$. However, in general, there is not necessarily a one-to-one connection between addresses and resources for them. It is hence impossible to just discard the pointer $\text{int}{*_0}\ a$ and perform address operations and checks on the reified resource n directly. For example, the function f could create multiple aliases for a through an assignment $b = a$, and it would not be clear which address to track in the reified resource. For the same reason, we cannot simply erase heap resources and compile the source pointer a to a linear target pointer a. In this case too, it would not be clear what to do when aliases are created for a. The problem in both cases is precisely that a is not a resource that should be handled linearly, but a regular non-linear program variable. Compiling both a and its aliases to non-linear length-0 capabilities and separately reifying the resources to linear capabilities instead, avoids this mismatch. By separately representing addresses and resources, we can keep our compiler general and uniform. In a subsequent phase, a compiler with a sufficiently clever static analysis engine could detect duplication of information and, for example, remove addresses of type $\text{int}{*_0}$ when their value can always be recomputed from a corresponding linear capability.

Secondly, the introduction of linear capabilities to represent separation logic resources, combined with the fact that the capabilities need to have the correct contents when passed to the context as resources, again requires our compiler to be separation-logic-proof-directed. The compiler inspects the verification proof to see how a source statement affects the state of the separation logic resources, and mirrors the change to these resources in the compiled version of this statement. For example, setting a to 1 in f is compiled to setting the reified resource for a, n, to 1. The call to g now also receives and returns the resource n along with the address a. In summary, separation-logic-proof-directedness entails that operations performed on pointers in the source language are performed on the reified resources corresponding to these pointer's resources in the target language, as these resources are what justify these operations in the separation logic verification in the first place.

Thirdly, the interaction between f and g at the trust boundary of the compiled component requires a checking function or stub (in this case an outcall stub for the outcall) that wraps g and verifies that it does indeed uphold its postcondition. This is necessary since f can outcall arbitrary compiled code g that might or might not uphold the contract that f expects of g. The postcondition of g says that it returns a resource n for address a with single-element contents 0. These conditions correspond exactly to the four guard statements in the outcall stub of Figure 5.2. If the verified component were to export f, allowing it to be called by untrusted code, then the compiled component would additionally contain an incall stub, to verify that f's precondition is met at call-time. To simplify the generation of stub functions, and thereby limit the size of the proofs, we have placed extra restrictions on contracts of functions at trust boundaries. These restrictions are described in more detail in Section 5.4.4, and ways to alleviate them are discussed in the future work section.

5.3 Source and Target Languages

The separation logic and the source and target languages are first discussed in Section 5.3.1. Section 5.3.2 then introduces notation to extend source language programs to source language proofs (as our compiler is separation-logic-proof-directed). Section 5.3.3 finally briefly discusses the operational semantics (including memory model) of the source and target languages.

In this paper k denotes an integer, id_ℓ a logical variable, id_p a program variable, f a function and n a heap resource. Logical and program variables are considered to have separate namespaces.

5.3.1 Source and Target Language Definition

The formalization of our separation logic assertions and the source and target languages is given in order of structural complexity by the BNF grammar in Figure 5.3, where the notation *symbol*⟨*parameter*⟩ is used for parameterized symbols. The concrete notation for separation logic annotations and assertions is inspired by the VeriFast tool [72]. For the features and syntax of the source language, we drew some inspiration from Clight, one of the intermediate languages used in CompCert [92, 93].

Both the source and target language (i.e. the program domain) build statements *sstm*/*tstm* out of expressions *sexp*/*texp*, components *scomp*/*tcomp* out of functions containing statements and programs *sprog*/*tprog* out of components. The separation logic (i.e. the logical domain), used in source function contracts and separation logic proofs, builds its assertions *assert* out of symbolic expressions *exp*. The remainder of this section discusses the BNF grammar in order.

Types To simplify types, the type bool is embedded in the type int, where 0 is true and $k \neq 0$ is false. The target type $\tau_\mathrm{t}*$ is assumed linear, requiring value erasure (i.e. replacing the value with null) whenever such a value is copied (e.g. assigned to another variable, passed to a function, stored in an array, ...), whereas the source type $\tau_\mathrm{s}*$ is a regular non-linear heap pointer. A type $(\tau, \ldots, \tau)$, representing length-n

$\tau ::= \text{int} \mid \tau* \mid (\tau, \ldots, \tau) \mid \text{list}_\tau$ (LOGICAL TYPE)

$\tau_\text{s} ::= \text{int} \mid \tau_\text{s}* \mid (\tau_\text{s}, \ldots, \tau_\text{s})$ (SOURCE TYPE)

$\tau_\text{t} ::= \text{int} \mid \tau_\text{t}* \mid (\tau_\text{t}, \ldots, \tau_\text{t}) \mid \tau_\text{t}*_0$ (TARGET TYPE)

$\mathit{cexp}\langle \mathit{exp} \rangle ::= k \mid \text{op1}\ \mathit{exp} \mid \mathit{exp}\ \text{op2}\ \mathit{exp} \mid \text{null} \mid (\mathit{exp}, \ldots, \mathit{exp}) \mid \mathit{exp}.k$

(COMMON EXPRESSIONS)

$\mathit{exp} ::= \mathit{id}_\ell \mid \mathit{cexp}\langle \mathit{exp} \rangle \mid \text{length}(\mathit{exp}) \mid \mathit{exp}[\mathit{exp}] \mid \forall \mathit{id}_\ell : \tau.\ \mathit{exp} \mid \exists \mathit{id}_\ell : \tau.\ \mathit{exp}$

$\quad \mid \text{repeat}(\mathit{exp}, \mathit{exp}) \mid \text{append}(\mathit{exp}, \mathit{exp}) \mid \text{take}(\mathit{exp}, \mathit{exp}, \mathit{exp}) \mid \text{update}(\mathit{exp}, \mathit{exp}, \mathit{exp})$

(LOGICAL EXPRESSIONS)

$\mathit{sexp} ::= \mathit{id}_\text{p} \mid \mathit{cexp}\langle \mathit{sexp} \rangle$ (SOURCE EXPRESSIONS)

$\mathit{texp} ::= \mathit{id}_\text{p} \mid \mathit{cexp}\langle \mathit{texp} \rangle \mid \text{addr}(\mathit{texp}) \mid \text{length}(\mathit{texp})$ (TARGET EXPRESSIONS)

$\mathit{cassert}\langle \mathit{assert} \rangle ::= \mathit{exp} \mid \mathit{assert} * \mathit{assert} \mid \mathit{exp}\ ?\ \mathit{assert}$ (COMMON ASSERTIONS)

$\mathit{assert} ::= n : \mathit{exp} \mapsto_{\tau_\text{s}} \mathit{exp}$ *array resource*

$\quad \mid n : [\mathit{assert}_\text{in} \mid \mathit{exp} \leq \mathit{id}_\ell < \mathit{exp}]$ *range resource*

$\quad \mid \mathit{cassert}\langle \mathit{assert} \rangle$ (OUTER SEPARATION LOGIC ASSERTION)

$\mathit{assert}_\text{in} ::= \mathit{exp} \mapsto_{\tau_\text{s}} \mathit{exp}$ *array resource*

$\quad \mid [\mathit{assert}_\text{in} \mid \mathit{exp} \leq \mathit{id}_\ell < \mathit{exp}]$ *range resource*

$\quad \mid \mathit{cassert}\langle \mathit{assert}_\text{in} \rangle$ (INNER SEPARATION LOGIC ASSERTION)

$\mathit{sstm} ::= \text{skip} \mid \mathit{id}_\text{p} = \text{malloc}(\mathit{sexp} * \text{sizeof}(\tau_\text{s})) \mid \text{foreach}(\mathit{sexp} \leq i < \mathit{sexp})\{\mathit{sstm}\}$

$\quad \mid \mathit{sstm}; \mathit{sstm} \mid \text{if}\ \mathit{sexp}\ \text{then}\ \mathit{sstm}\ \text{else}\ \mathit{sstm} \mid \tau\ \mathit{id}_\text{p} \mid \mathit{id}_\text{p} = \mathit{sexp} \mid (\mathit{id}_\text{p}^*) = f(\mathit{sexp}^*)$

$\quad \mid \text{guard}(\mathit{sexp}) \mid \mathit{id}_\text{p}[\mathit{sexp}] = \mathit{sexp} \mid \mathit{id}_\text{p} = \mathit{sexp}[\mathit{sexp}]$ *regular statements*

$\quad \mid //@\text{split}\ n[\mathit{sexp}] \mid //@\text{join}\ n\ n \mid //@\text{flatten}\ n$

$\quad \mid //@\text{collect}\ n^* \cdot \ldots \cdot n^*$ *ghost statements*

(SOURCE STATEMENTS)

$\mathit{tstm} ::= \text{skip} \mid \mathit{id}_\text{p} = \text{malloc}(\mathit{texp} * \text{sizeof}(\tau_\text{t})) \mid \text{foreach}(\mathit{texp} \leq i < \mathit{texp})\{\mathit{tstm}\}$

$\quad \mid \mathit{tstm}; \mathit{tstm} \mid \text{if}\ \mathit{texp}\ \text{then}\ \mathit{tstm}\ \text{else}\ \mathit{tstm} \mid \tau\ \mathit{id}_\text{p} \mid \mathit{id}_\text{p} = \mathit{texp} \mid (\mathit{id}_\text{p}^*) = f(\mathit{texp}^*)$

$\quad \mid \text{guard}(\mathit{texp}) \mid \mathit{id}_\text{p}[\mathit{texp}] = \mathit{texp} \mid \mathit{id}_\text{p} = \mathit{texp}[\mathit{texp}]$ *regular statements*

$\quad \mid (\mathit{id}_\text{p}, \mathit{id}_\text{p}) = \text{split}(\mathit{id}_\text{p}, \mathit{texp}) \mid \mathit{id}_\text{p} = \text{join}(\mathit{id}_\text{p}, \mathit{id}_\text{p})$ *built-in functions*

(TARGET STATEMENTS)

$\mathit{isfunc} ::= \tau_\text{s}^*\ f((\tau_\text{s}\ \mathit{id})^*)\ //@\text{pre}\ \mathit{assert}\ //@\text{post}\ \mathit{assert}\ \{\mathit{sstm}; \text{return}\ \mathit{sexp}^*\}$

(IMPLEMENTED SOURCE FUNCTION)

$\mathit{csfunc} ::= \tau_\text{s}^*\ f((\tau_\text{s}\ \mathit{id})^*)\ //@\text{pre}\ \mathit{assert}\ //@\text{post}\ \mathit{assert}$

(CONTEXT SOURCE FUNCTION)

$\mathit{itfunc} ::= \tau_\text{t}^*\ f((\tau_\text{t}\ \mathit{id})^*)\ \{\mathit{tstm}; \text{return}\ \mathit{texp}^*\}$ (IMPLEMENTED TARGET FUNCTION)

$\mathit{ctfunc} ::= \tau_\text{t}^*\ f((\tau_\text{t}\ \mathit{id})^*)$ (CONTEXT TARGET FUNCTION)

$\mathit{scomp} ::= \mathit{isfunc}^+\ //@\text{import}\ \mathit{csfunc}^*\ //@\text{export}\ \mathit{csfunc}^*$ (SOURCE COMPONENT)

$\mathit{tcomp} ::= \mathit{itfunc}^+\ //@\text{import}\ \mathit{ctfunc}^*\ //@\text{export}\ \mathit{ctfunc}^*$ (TARGET COMPONENT)

$\mathit{sprog} ::= \mathit{scomp}^+\ //@\text{main} = \mathit{id}$ (SOURCE PROGRAM)

$\mathit{tprog} ::= \mathit{tcomp}^+\ //@\text{main} = \mathit{id}$ (TARGET PROGRAM)

Figure 5.3: Grammar describing our separation logic and the source and target languages.

($n \geq 0$) tuples is present for all three cases. Pointers and arrays are seen as identical types $\tau*$ in our formalization, for simplicity's sake.

The target language has an extra type $\tau_{\text{t}}*_0$ of length-0 non-linear capabilities, used by the compiler to store the compiled version of the source-level permissionless program variables. The separation logic has a type list_τ, which is a type of logical list variables. These variables are used to represent the contents of source-level arrays in resources.

Expressions Separation logic makes a distinction between program variables id_{p}, which appear in source programs, and expressions *sexp* over them on the one hand and logical (or symbolic) proof-only variables id_ℓ, which only appear in separation logic contracts and proofs, and logical expressions *exp*, which are a third type of expression, on the other hand. Source expressions are the least expressive, allowing program variables and the common expressions only.

Target expressions *texp* additionally contain a function addr, which returns the $\tau_{\text{t}}*_0$-type address of a $\tau_{\text{t}}*$-type value, and a function “length", which returns the length of the region addressed by a linear capability $\tau_{\text{t}}*$. These functions are realistic (since linear capabilities encode their own length and address information) and needed (for the contract checks performed in incall and outcall stubs).

Logical expressions contain extra functions to manipulate list_τ-typed values: a length function and an indexing construct $exp[exp]$. Universal and existential quantification allow constraining elements of non-statically sized logical lists. For readability, we also provide repeat, append, take and update list constructs, but these can be desugared to the other logical expression constructs. The $\text{repeat}(exp_1, exp_2)$ construct returns a logical exp_1-length list where each element equals exp_2, $\text{append}(exp_1, exp_2)$ appends lists exp_1 and exp_2, $\text{take}(exp_1, exp_2, exp_3)$ constructs a new list from elements exp_2 up to (but not including) exp_3 of list exp_1 and $\text{update}(exp_1, exp_2, exp_3)$ updates index exp_2 of list exp_1 with exp_3.

Assertions Assertions are the building blocks of function contracts and separation logic proofs. Logical expressions *exp* are assertions, and so is the separating conjunction $*$ of two assertions. Conditional assertions of the form $exp \ ? \ assert$ express that *assert* only needs to hold if $exp == \text{true}$ can be derived. Lastly, two types of assertions represent spatial resources: array and range resources. Array resources $exp_1 \mapsto_{\tau_s} exp_2$ represent an array at address exp_1, containing the elements of list exp_2. In order to talk about fixed-size array resources, we use the syntax $exp_1 \mapsto_{\tau_s} [exp_2^1, \ldots, exp_2^k]$, which desugars to $exp \mapsto_{\tau_s} l * \overline{l[i] = exp_2^i} * \text{length}(l) = k$ and was already demonstrated in Figure 5.2.

Range resources $[assert \mid exp_1 \leq id_\ell < exp_2]$ represent the separating conjunction of *assert* for each value from exp_1 to (but not including) exp_2, where *assert* usually depends on id_ℓ. Range resources can be nested and will be useful for the back-translation in Section 5.6.

An alternative design could have made use of a single, primitive points-to resource $a \mapsto_{\mathrm{p}} v$, representing permissions to access the single value v at address a, instead of our array resources. All types of array resources could then be desugared in terms of this primitive primitive resource as follows: $a \mapsto l \triangleq [a + i \mapsto_{\mathrm{p}} l[i] \mid 0 \leq i < \mathrm{length}(l)]$. This change should not impact the rest of the formalization in any major way.

Since both types of resources will be reified during compilation, as demonstrated for array resources in Section 5.2, we have to associate names n with both types. However, within a named range resource, no more names should occur, as the outer resource will be reified as a whole. This is the reason for the distinction between outer and inner resources in the grammar.

Statements Statements in the target correspond one to one to statements in the source, except for the //@flatten and //@collect statements, which do not appear in the target language. The guard statement gets stuck during execution if its condition evaluates to false. The foreach statement executes its statement for every value of i in the given range. The foreach statement could technically be left out, since we have recursive calls, but it is kept for conciseness. Finally, both source and target language have array assignment and array lookup statements.

The malloc statement used in the target language does not correspond to the vanilla malloc function in C. Firstly, it returns a linear capability, not a regular pointer (since it essentially creates a target-level reified resource). Secondly, it guarantees a fresh heap location for the allocated variable. This avoids reuse of locations after free; if any newly allocated location could have been previously used by the context, it could have kept a reference to it and hereby broken the linearity guarantees. The target malloc statement hence respects *temporal safety*, an important desired property in any capability machine, even for non-linear capabilities [176]. In a practical implementation, freshness of the malloced heap locations could realistically be achieved by a form of garbage collection, much like in libgc [68]. Interestingly, the run time bounds that capabilities inherently provide will allow for more precise garbage collection, instead of the conservative variant that libgc necessarily employs.

For simplicity reasons, we do not consider free (see Section 5.8 for further discussion).

In addition to these regular statements, there are also ghost statements. These operate on logical state instead of program state in the source language, and are only relevant for the construction of the separation logic proof. VeriFast-style syntactic ghost statements are usually used as hints for a semi-automated proof tool during construction of the proof. We hence do not technically need them, since, as mentioned in the introduction, our compiler assumes a full separation logic proof as input (cfr. Figure 5.1). Instead, we could simply have separation logic rules manipulating ghost state without requiring corresponding syntactic constructs. For readability reasons in symbolic executions, and to make the source language correspond better to the target syntactically, we still use VeriFast-style syntactic ghost statements in this paper. Since separation logic resources are reified during compilation, ghost statements will have to be reified as well. Ghost statements and their target-level counterparts will hence be syntactically different between the source and target languages. We now discuss each ghost statement.

The source split statement splits a separation logic resource n in two at a given index *sexp*. For example, $//@\text{split } n[1]$ applied to the array resource $n : a \mapsto [1,2,3]$ splits this resource into $n' : a \mapsto [1]$ and $n'' : a+1 \mapsto [2,3]$. When applied to the range resource $n : [\mathit{assert} \mid 0 \leq i < 3]$, the same $//@\text{split } n[1]$ creates $n' : [\mathit{assert} \mid 0 \leq i < 1]$ and $n'' : [\mathit{assert} \mid 1 \leq i < 3]$. The source join statement is the inverse of split and merges two adjacent resources into one, e.g. both previous sets of resources n' and n'' are merged into n by $//@\text{join } n'\ n''$. As the resources that split and join operate on are reified, so are the operations themselves: built-in target functions to analogously split and join linear capabilities are provided in the target language.

The source flatten and collect statements are each other's inverse and respectively strip a top-level range resource or create it. This only works for statically-sized range resources. For example, the resources $n' : a+1 \mapsto [1]$ and $n'' : a+2 \mapsto [2]$ can be combined into $n : [a+i \mapsto [i] \mid 1 \leq i < 3]$ by the statement $//@\text{collect } n' \cdot n''$, where the $\cdot$ is used to delimit the sequence of resource names for each individual index of the constructed range resource. Notice again how only the top-level resource is named. The postcondition in the proof specifies which exact range resource is created. Conversely, the statement $//@\text{flatten } n$ creates resources n' and n'' from n.

Interestingly, the target language does *not* contain reified, built-in functions to flatten or collect linear capabilities. The reason is that ghost statements are the only way to manipulate resources in the source language, and a flatten and collect statement to switch representations are hence required. In the target language however, all resources are reified to linear capabilities, which *can* be manipulated by target code. The flatten and collect statements can hence be compiled to regular target-level statements, obviating the need for built-in functions. For example, the effect of $//@\text{collect } n' \cdot n''$ could be realized by the compiled code: $\text{int}{*}{*}\ n;\ n = \text{malloc}(2 * \text{sizeof}(\text{int}*));\ n[0] = n';\ n[1] = n''.$

Functions Two classes of functions exist; implemented and context functions. Implemented functions consist of both a function declaration and a body. Context functions solely consist of the declaration that a component expects of this function. For simplicity, tuple return types (τ^*) exist in both source and target language and every function has to end in a single return statement return exp^*. Source language functions are annotated with separation logic contracts that use the separation logic assertions mentioned before for pre- and postcondition. As mentioned, contracts are situated in the separation logic domain and hence range over logical variables id_ℓ, not program variables id_p.

Components and programs A sequence of implemented functions that uses context functions in its function import and export lists is called a component. A sequence of components with a main function *id* forms an entire program.

5.3.2 Source Language Proofs

As explained, our compiler is separation-logic-proof-directed, i.e., not a regular source program, but its separation logic proof is the input to the compiler. In addition to the grammar defining the syntax of source language programs, we need a notation for separation logic proofs (in this subsection) and a set of inference rules that describe how to construct such proofs starting from the source code (Section 5.4). *Hoare triples* are the building blocks of separation logic rules [129].

Classical separation logic uses Hoare triples of the form $\{P\}\ c\ \{Q\}$. In this paper, they have a partial correctness semantics: $\{P\}\ c\ \{Q\}$ states that given precondition P, either postcondition Q holds after execution of the piece of source code c, or c diverges [129]. A triple $\{P\}\ c\ \{Q\}$ is only provable if there exists a proof tree, constructed from the individual separation logic rules, that has this triple as the root. In our formalization of separation logic, however, we split the condition P (and Q) into two separate parts, partly inspired by the approach of VeriFast [168]. These parts are called the *symbolic heap* P and the *environment* γ and give rise to the extended Hoare triple notation $\{P\}_\gamma\ c\ \{Q\}_{\gamma'}$, stating that if (P, γ) holds, then either c diverges or (Q, γ') holds after execution of c. If $\gamma == \gamma'$, we shorten the Hoare triple notation to $\{P\}\ c\ \{Q\}$. The two aspects of extended Hoare triples and the triples themselves are defined by the following BNF grammar:

$P ::= \ \mathit{assert}$	(Symbolic Heap)	$c ::= \ \mathit{sstm}$	
$\gamma ::= \ \cdot[\mathit{id}_\mathrm{p} : \mathit{exp}]^*$	(Environment)	$\mid \mathit{sstm};\ \mathrm{return}\ \mathit{sexp}$	(Source Code)
		$\mathit{triple} ::= \ \{P\}_\gamma\ c\ \{P\}_\gamma$	(Hoare Triple)

We will often use this notation c independently, to denote a piece of source (or target) code.

The two parts, γ and P, of separation logic states have the following meaning:

- The *environment* γ maps program variables id_p to expressions *exp* over logical, proof-level variables id_ℓ. The environment γ hence relates the program domain to the logical domain.
- The *symbolic heap* P is a $*$-separated list of assertions representing the symbolic program state. It is of the same form *assert* as the contracts described in

Figure 5.3.

Hoare triple syntax is only useful for verifying (parts of) function bodies. In Section 5.4, we verify entire functions, components and programs. Given *any* piece of source code $\mathfrak{s}$, be it (part of) a function (body), a component or a program, the notation $\vdash \mathfrak{s}$ represents a specific, valid separation logic proof tree for $\mathfrak{s}$. This proof $\vdash \mathfrak{s}$ is what our proof-directed compiler uses as input.

5.3.3 Operational Semantics

We define C-style small-step operational semantics for both the source and target languages. The operational semantics rules in both languages are of the following form:

$$\frac{\mathit{Premise}}{\langle \overline{s}, h\rangle \mid \overline{c} \hookrightarrow \langle \overline{s'}, h'\rangle \mid \overline{c'}}\ (\textsc{RuleName})$$

The small-step operational semantics $\hookrightarrow$ transform a program state $\langle \overline{s}, h\rangle \mid \overline{c}$ into a state $\langle \overline{s'}, h'\rangle \mid \overline{c'}$, where $\overline{s}$ is the list of stack frames containing local variables, h the heap, and $\overline{c}$ a list of partly-executed function bodies, separated by return statements, where the sequence $\overline{c}$ corresponds to the sequence of stack frames $\overline{s}$. A function call creates a new stack frame and accompanying executing function body and adds them to $\overline{s}$ and $\overline{c}$ respectively. A return statement erases one of each. Erroneous programs get stuck, because no operational semantics rules apply to them. The same happens for false guard statements.

One special case should be considered; at the very start of execution, $\overline{c}$ is a single, monolithic source or target program *prog*. Execution of a full program *prog* starts off by executing the function whose *id* is given in //@main = *id* , thereby creating the first stack frame. Since we are interested in the termination behavior of our programs, rather than their exact output, we require the main function *id* to have return type void. Main functions are not allowed to have arguments either. The initial state before calling the main function is $\langle \bullet, \bullet\rangle \mid \mathit{prog}$: both the stack and heap are empty, denoted by $\bullet$. We say that *prog* terminates, denoted $\mathit{prog} \Downarrow$, if a sequence of small-step transitions exists that reduces the program to a single return statement with no arguments (i.e. the original return statement of the main function). We can then define termination formally as follows:

$$\mathit{prog} \Downarrow\ \triangleq \exists\, s, h, \overline{exp}.\ \langle \bullet, \bullet\rangle \mid \mathit{prog} \hookrightarrow^{*} \langle s, h\rangle \mid \text{return}$$

Note that termination does not include getting stuck.

The memory model for both source and target languages is location-based, i.e. addresses are pairs (l, i) of an opaque location and an index i and hence $h \in$

$(\mathrm{Loc}, \mathrm{Index}) \overset{\text{fin}}{\rightharpoonup} \mathrm{Val}$. A malloc statement that allocates k units of type τ creates a new location l in the heap, populated with default values (out of simplicity considerations) for type τ at indexes 0 through $k-1$. The default values for pointers is the null pointer, which implies that it is currently impossible to have a non-nullable pointer type in the source or target language. A possible solution to avoid this technical limitation would be to stick closer to the C semantics by not specifying a default value and making a dereference of an uninitialized pointer undefined behavior instead. Separately malloced variables are hence logically separate.

Source pointer values of type $\tau_{\mathrm{s}}*$ follow the heap memory model and are hence denoted as either null, in case of the null pointer, or pairs (l, i). Target-level linear capabilities $\tau_{\mathrm{t}}*$, on the other hand, are denoted as either null or $l^{[a,b]}$, where $[a, b]$ is the closed interval of indexes at location l that they carry authority over. They do not need an index i, as source pointer arithmetic is compiled to target pointer arithmetic on length-0 capabilities and not on linear capabilities (see Section 5.2). A capability value $l^{[a,b]}$ hence always points to index a. Target-level length-0 capabilities are represented by l_0^i-values, which do not carry any authority, but keep an index i for pointer arithmetic.

The operational semantics for the source language are standard, but those for the target are not. The two main differences between these semantics are discussed in the following paragraphs, where the target-level semantics are illustrated by means of some representative rules in Figure 5.4. The rules in this figure make the simplifying assumption that $\bar{s} = s$, i.e. only a single stack frame is considered, and that $\bar{c}$ consists of a single source statement. In the following discussion, $h[(l,i) \longrightarrow v]$ denotes an update of the existing value of the heap h at location l and index i with value v. For the stack, $s[id_{\mathrm{p}} \longrightarrow v]$ similarly denotes an assignment of the value v to the previously declared variable id_{p}. The evaluation of exp in stack-frame s is denoted by $[\![exp]\!]_s$.

The first difference between the source and target semantics is caused by the linearity of capabilities, as they cannot be duplicated. When a linear capability $l^{[a,b]}$ is copied, the original value is set to null. We call this process linear capability *erasure*. The target level judgment $v \rightsquigarrow_{\mathrm{ValErase}} v'$ describes how a target value v is erased into a result value v' by replacing linear capabilities with null pointers. For example:

$$l^{[a,b]} \rightsquigarrow_{\mathrm{ValErase}} \mathrm{null} \qquad\qquad (l_1^{[a_1,b_1]}, 5, l_2^{[a_2,b_2]}) \rightsquigarrow_{\mathrm{ValErase}} (\mathrm{null}, 5, \mathrm{null})$$

The rule ArrayLkup in Figure 5.4 describes the operational semantics of target-level array lookup. It is equivalent to its source-level counterpart, except for the addition of a $\rightsquigarrow_{\mathrm{ValErase}}$ judgment. It illustrates how the read value v should be erased inside the array, by setting $h'(l, a+n) = v'$.

The rule ArrayMut in Figure 5.4 describes the operational semantics of target-level array mutation. The sole difference with the source-level version is caused by the

$$\dfrac{\begin{array}{ccc} \llbracket texp_1 \rrbracket_s = l^{[a,b]} & \llbracket texp_2 \rrbracket_s = n & h(l, a+n) = v \\ 0 \leq n \leq b - a & & v \rightsquigarrow_{\text{ValErase}} v' \\ s' = s[id_{\text{p}} \to v] & & h' = h[(l, a+n) \to v'] \end{array}}{\langle s, h \rangle \mid id_{\text{p}} = texp_1[texp_2] \hookrightarrow \langle s', h' \rangle \mid \text{skip}} \ (\textsc{ArrayLkup})$$

$$\dfrac{\begin{array}{ccc} s(id_{\text{p}}) = l^{[a,b]} & \llbracket texp_1 \rrbracket_s = n & \llbracket texp_2 \rrbracket_s = v \\ 0 \leq n \leq b - a & & texp_2 \rightsquigarrow^{s}_{\text{StoreLinCap}} [env] \\ s' = s[env] & & h' = h[(l, a+n) \to v] \end{array}}{\langle s, h \rangle \mid id_{\text{p}}[texp_1] = texp_2 \hookrightarrow \langle s', h' \rangle \mid \text{skip}} \ (\textsc{ArrayMut})$$

$$\dfrac{\begin{array}{c} s(id_{\text{p3}}) = l^{[a,b]} \qquad \llbracket texp \rrbracket_s = n \qquad 1 \leq n \leq b - a \\ s' = s[id_{\text{p1}} \to l^{[a,a+n]}][id_{\text{p2}} \to l^{[a+n,b]}][id_{\text{p3}} \to \text{null}] \end{array}}{\langle s, h \rangle \mid (id_{\text{p1}},\ id_{\text{p2}}) = \text{split}(id_{\text{p3}}, texp) \hookrightarrow \langle s', h \rangle \mid \text{skip}} \ (\textsc{Split})$$

$$\dfrac{\begin{array}{c} s(id_{\text{p2}}) = l^{[a,n]} \qquad s(id_{\text{p3}}) = l^{[n,b]} \\ s' = s[id_{\text{p1}} \to l^{[a,b]}][id_{\text{p2}} \to \text{null}][id_{\text{p3}} \to \text{null}] \end{array}}{\langle s, h \rangle \mid id_{\text{p1}} = \text{join}(id_{\text{p2}}, id_{\text{p3}}) \hookrightarrow \langle s', h \rangle \mid \text{skip}} \ (\textsc{Join})$$

Figure 5.4: Rules illustrating the target language operational semantics and its linear aspects.

judgment

$$texp_2 \rightsquigarrow^{s}_{\text{StoreLinCap}} [env]$$

which we illustrate below. This judgment is used to erase any linear capabilities present in $texp_2$ that were written into the array $l^{[a,b]}$ by the ArrayMut operation, to avoid them being duplicated. The resulting evironment $[env]$ nulls these capabilities in the current stack frame, as shown by the assignment $s' = s[env]$ in the ArrayMut rule. Additionally, this judgment makes the semantics get stuck if the same linear capability is used twice in $texp_2$.

Generally, $texp \rightsquigarrow^{s}_{\text{StoreLinCap}} [env]$ can be seen as the lifting of $v \rightsquigarrow_{\text{ValErase}} v'$ from values v to expressions $texp$. Rather than erasing capabilities inside values v, now stack variables id_{p} that appear inside $texp$ have to be reassigned in order to erase their linear capabilities. This is the reason the judgment's output is not an expression $texp'$, but rather, a reassignment of these local variables, i.e. an environment $[env]$. The judgment $texp \rightsquigarrow^{s}_{\text{StoreLinCap}} [env]$ is used whenever a target operational semantics rule evaluates and uses $texp$, and the linear capabilities that are moved in the process have to be erased. The judgment is therefore also used when e.g. calling functions using linear arguments or when assigning variables.

More concretely, $\mathit{texp} \rightsquigarrow^{s}_{\text{StoreLinCap}} [\mathit{env}]$ erases the linear capabilities that appear inside *texp* by reassigning (in the current stack frame s) local variables id_{p} appearing inside *texp*. Notice that solely the linear capabilities that are actually used linearly should be erased; e.g. id_{p} appearing under an equality or as an argument to the addr function do not require erasure. The environment $[\mathit{env}]$ computes values for the erased id_{p} by using $\rightsquigarrow_{\text{ValErase}}$. For example, assuming $[\![\mathit{id}_{\text{p1}}]\!]_s = l_1^{[a_1,b_1]}$ and $[\![\mathit{id}_{\text{p2}}]\!]_s = (l_2^{[a_2,b_2]}, l_3^{[a_3,b_3]})$, we have:

$$\begin{aligned}
(\mathit{id}_{\text{p1}}, \mathit{id}_{\text{p2}}) &\rightsquigarrow^{s}_{\text{StoreLinCap}} [\mathit{id}_{\text{p1}} \rightarrow \text{null}][\mathit{id}_{\text{p2}} \rightarrow (\text{null}, \text{null})] \\
\text{addr}(\mathit{id}_{\text{p2}}) &\rightsquigarrow^{s}_{\text{StoreLinCap}} [\,] \\
\mathit{id}_{\text{p2}}.2 &\rightsquigarrow^{s}_{\text{StoreLinCap}} [\mathit{id}_{\text{p2}} \rightarrow (l_2^{[a_2,b_2]}, \text{null})] \\
(\mathit{id}_{\text{p1}}, \mathit{id}_{\text{p2}}).1 &\rightsquigarrow^{s}_{\text{StoreLinCap}} [\mathit{id}_{\text{p1}} \rightarrow \text{null}]
\end{aligned}$$

Additionally, $\rightsquigarrow^{s}_{\text{StoreLinCap}}$ should ensure that a single linear capability is not used multiple times in the same *texp*, since this would cause duplication. For example, assuming the same stack frame as before, we have $\neg\exists v'.\, (\mathit{id}_{\text{p1}}, \mathit{id}_{\text{p1}}) \rightsquigarrow^{s}_{\text{StoreLinCap}} v'$, causing the semantics to avoid duplication by *getting stuck*.

The second big difference between source and target occurs where the built-in target-level functions join and split are concerned. These ghost statements and their reification were already discussed in Section 5.3.1. As mentioned, source-level ghost statements solely have an effect on the separation logic proof and are hence equivalent to skip in the source semantics. In the target, on the other hand, the reified ghost statements manipulate physical linear capabilities instead. The rules for these reified ghost statements are given by Split and Join in Figure 5.4. As expected, they respectively split and join linear capabilities. Notice that both rules erase their source operands to ensure linearity, by explicitly setting them to null. A use of $\rightsquigarrow^{s}_{\text{StoreLinCap}}$ is not required, given that the source operands are constrained to be simple program variables.

5.4 Inference Rules and Compilation by Example

This section introduces the separation logic inference rules that constitute separation logic proof trees. Because our compiler is separation-logic-proof-directed, the compilation rules directly derive from these rules, so we present both simultaneously. The rules we present in this section are syntactic, i.e. presented axiomatically rather than derived from the operational semantics. While it allowed us to focus more on the essential points of this paper, this approach has disadvantages. For a discussion of syntactically versus semantically derived rules, see the future work section. A relevant selection of rules is spread over Figures 5.7, 5.10, 5.11 and 5.12. The separation logic rules and the compilation rules are obtained by respectively ignoring and not ignoring all green text. Compilation of a separation logic proof $\vdash \mathfrak{s}$ to target code $\mathfrak{t}$ is denoted $\vdash \mathfrak{s} \rightsquigarrow \mathfrak{t}$. Other judgments appearing in these figures are explained as needed below.

We illustrate the rules using the running example in Figure 5.5. Since our compiler is separation-logic-proof directed, it receives a proof of the verified component in Figure 5.5 as input. To keep the example simple and concise, we avoid foreach loops and range resources, stick to arrays of statically-known size, and we use the fixed-size array resource syntax discussed in Section 5.3.1. The inference and compilation rules will still be presented in their general form.

By f's contract in Figure 5.5, it receives a pointer a and a resource m to access a two-element integer array corresponding to this pointer. For simplicity, f leaks the memory resource m by not handing it back in its postcondition. It adds 1 to either the second element of a or its negation, depending on the first element c. It will use an untrusted library function *add1* to add the 1. The contract of *add1* specifies that it takes a pointer a and a resource m to access a one-element array corresponding to a. It returns the value *result*, equaling the contents of a increased by 1, and returns the same resource from the precondition, now named n, in its postcondition. The symbolic variable identifier *result* is a privileged name, used to denote a function's return value(s) in its postcondition ($\overline{\mathit{result}_i}$ is used in case of multiple return values). Based on the value of a's first element c, f either calls *add1* directly, or emulates adding 1 to the negation by inverting the last element of a, storing it in a new array b and only then calling *add1*.

There are four types of compilation rules: structural, basic, higher-level and stub compilation rules. We discuss these classes in the next subsections and illustrate them using the running example.

5.4.1 Structural Rules

Structural rules are rules that build more complex proofs from simpler proofs. They are more involved in the construction of the separation logic proof itself than in the proof-directed aspects of the compilation and therefore have very straightforward compilation rules. The structural rules are ConseqPost, Frame, Seq and If, presented in Figure 5.7.

The consequence and frame rules ConseqPost and Frame are pure proof glue rules that allow altering proofs and do not influence the compiled program. The

Source

Verified Component

```
int f(int* a)
//@pre m: a ↦int [c,a1]
//@post result == (c == 0 ? a1 + 1 : −a1 + 1) {
   int res; int c; c = a[0];
   //@split m[1];
   if c == 0
   then   {res = add1(a + 1)}
   else   {int* b; int a1; a1 = (a + 1)[0];
            b = malloc(1 * sizeof(int));
            b[0] = −a1;
            res = add1(b)};
   return res }
```

Context Declaration

```
int add1(int* a)
//@pre m: a ↦int [a1]
//@post n: a ↦int [a1] * result == a1 + 1
```

Target

Verified Component

```
int fcomp(int*0 a,int* m){
   int res; int c; c = m[0];
   int* m1; int* m2; m1,m2 = split(m,1);
   if c == 0
   then {int* r1; res,r1 = add1comp(a + 1,m2)}
   else  {int*0 b; int a1; a1 = m2[0];
            int* l; l = malloc(1 * sizeof(int));
            b = addr(l);
            l[0] = −a1;
            int* r2; res,r2 = add1comp(b,l)};
   return res }
```

Context Declaration

```
(int,int*) add1(int*0 a,int* m)
```

Outcall Stub

```
(int,int*) add1comp(int*0 a,int* m){
   int*0 a^pre; int a1^pre;
   a^pre = addr(m); a1^pre = m[0];
   int result; int* n;
   (result,n) = add1(a,m);
   guard(n!=null); guard(length(n) == 1);
   int*0 a^post; int a1^post;
   a^post = addr(n); a1^post = n[0];
   guard(result == a1^post + 1);
   guard(a^post == a); guard(a1^post == a1^pre);
   return (result,n) }
```

Figure 5.5: Illustrative example: conditionally add 1 to the second element of a length-2 array or its negation.

```
{m : a ↦int [c, a1]}.[a:a]
int res;
{m : a ↦int [c, a1]}.[a:a][res:0]
int c; c = a[0];
{m : a ↦int [c, a1]}.[a:a][res:0][c:c]
//@split m[1];
{m1 : a ↦int [c]
* m2 : a + 1 ↦int [a1]}.[a:a][res:0][c:c]
if c == 0
then
{m2 : a + 1 ↦int [a1] * c == 0}.[a:a][res:0]
{res = add1(a + 1)}
{c == 0 * x == a1 + 1}.[res:x]
else
{m2 : a + 1 ↦int [a1] * c != 0}.[a:a][res:0]
{int* b; int a1;
{m2 : a + 1 ↦int [a1]
* c != 0}.[a:a][res:0][b:null][a1:0]
a1 = (a + 1)[0];
{c != 0}.[res:0][b:null][a1:a1]
b = malloc(1 * sizeof(int));
{c != 0 * l : y ↦int [0]}.[res:0][b:y][a1:a1]
b[0] = −a1;
{c != 0 * l : y ↦int [−a1]}.[res:0][b:y]
res = add1(b)};
{c != 0 * x == −a1 + 1}.[res:x]
{x == (c == 0 ? a1 + 1 : −a1 + 1)}.[res:x]
return res
{result == (c == 0 ? a1 + 1 : −a1 + 1)}.
```

Figure 5.6: Separation logic proof of the function given in the illustrative example.

$$\dfrac{\begin{array}{c} \{P\}_{\gamma}\ c\ \{Q\}_{\gamma'} \rightsquigarrow p \\ \mathrm{dom}(\gamma_{\mathrm{post}}) \subseteq \mathrm{dom}(\gamma') \qquad \vdash Q \Rightarrow Q^{\mathrm{post}} \\ \forall x \in \mathrm{dom}(\gamma_{\mathrm{post}}).\, Q \vdash \gamma'(x) == \gamma_{\mathrm{post}}(x) \\ \text{(}\textsc{ConseqPost}\text{)} \end{array}}{\{P\}_{\gamma}\ c\ \{Q_{\mathrm{post}}\}_{\gamma_{\mathrm{post}}} \rightsquigarrow p} \qquad \dfrac{\begin{array}{c} \{P\}_{\gamma}\ c\ \{Q\}_{\gamma'} \rightsquigarrow p \\ \mathrm{CN}(R) = \overline{n} \qquad \overline{n}\ \text{fresh} \\ \gamma_{\mathrm{s}} = \gamma \uplus \gamma_{\mathrm{frame}} \qquad \gamma'_{\mathrm{s}} = \gamma' \uplus \gamma_{\mathrm{frame}} \\ \text{(}\textsc{Frame}\text{)} \end{array}}{\{P * R\}_{\gamma_{\mathrm{s}}}\ c\ \{Q * R\}_{\gamma'_{\mathrm{s}}} \rightsquigarrow p}$$

$$\dfrac{\begin{array}{c} \{P\}_{\gamma}\ \mathit{sstm}_1\ \{Q\}_{\gamma'} \rightsquigarrow p_1 \\ \{Q\}_{\gamma'}\ \mathit{sstm}_2\ \{R\}_{\gamma''} \rightsquigarrow p_2 \\ \text{(}\textsc{Seq}\text{)} \end{array}}{\{P\}_{\gamma}\ \mathit{sstm}_1; \mathit{sstm}_2\ \{R\}_{\gamma''} \rightsquigarrow p_1; p_2} \qquad \dfrac{\begin{array}{c} \{P * \mathit{sexp}_{\gamma}\}_{\gamma}\ \mathit{sstm}_1\ \{Q\}_{\gamma'} \rightsquigarrow p_1 \\ \{P * !\mathit{sexp}_{\gamma}\}_{\gamma}\ \mathit{sstm}_2\ \{Q\}_{\gamma'} \rightsquigarrow p_2 \\ \text{(}\textsc{If}\text{)} \end{array}}{\begin{array}{c} \{P\}_{\gamma}\ \text{if}\ \mathit{sexp}\ \text{then}\ \mathit{sstm}_1\ \text{else}\ \mathit{sstm}_2\ \{Q\}_{\gamma'} \\ \rightsquigarrow \text{if}\ \mathit{sexp}\ \text{then}\ p_1\ \text{else}\ p_2 \end{array}}$$

Figure 5.7: Structural separation logic rules that can be extended to compilation rules.

consequence rule ConseqPost allows weakening of both the symbolic heap and the environment in the postcondition of a separation logic triple, in order to link it to a subsequent triple. The judgment *assump* ⊢ *cond* denotes that the boolean condition *cond* holds under the assumptions in *assump*. In fact, we also need a dual rule ConseqPre that allows strengthening the precondition of a separation logic triple. The rules ConseqPre and ConseqPost combine to form the full consequence rule Conseq. Our full Conseq rule also allows renaming outer separation logic resources n and manipulating conditional assertions. Both these operations will influence the compiled code p, but are omitted for brevity.

The frame rule is a classical separation logic rule. It allows neglecting a redundant part of the symbolic heap and the environment in order to simplify the separation logic state. The function $\mathrm{CN}(R)$ returns all resource names that appear in the separation logic assertion R. We require these names to be fresh, to avoid name clashes. The sequence and conditional rules Seq and If describe proofs for the sequencing and conditional source statements, respectively. If all applications of these four structural proof rules are left implicit in a function body's separation logic proof, the proof tree can be represented as a *symbolic execution* [168]. Such a symbolic execution of the body of f is used in Figure 5.6 to illustrate the rules in this subsection and the next.

The Conseq rule is used to omit information that is no longer useful as quickly as possible in order to keep the proof concise. Examples are the resource m_1 that is dropped after line 7 and the environment entry $[a : a]$ that is omitted after line 16. Consequence is also used to reshape postconditions to match the conditions required by a different rule: Conseq unifies e.g. the symbolic heaps on lines 12 and 24 by weakening them to the symbolic heap on line 25. Because of this unification, the If rule becomes applicable.

The If rule obviously creates the separation logic triple for the if-statement on lines

8-24. Notice how the If rule introduces $c == 0$ and $c\ != 0$ in the symbolic heaps on lines 10 and 14, respectively.

5.4.2 Basic Rules

Basic rules construct a proof triple from the ground up for a single non-sequenced source statement. They are the elementary building blocks of the symbolic execution in Figure 5.6 and the workhorses of the separation-logic driven compiler. The rules are named after the source statements they create a proof for, i.e. Skip, Malloc, For, Flatten, Collect, Split (has 2 versions: one for range resources and one for array resources), Join (again has 2 versions), VarDecl, VarAsgn, ArrayMut, ArrayLkup, Guard, FApp and Return. A representative selection is presented in Figure 5.10.

In the following descriptions of the inference rules, $sexp_\gamma$ denotes $sexp$ where each program variable id_p is substituted by $\gamma(id_\mathrm{p})$ (implicitly requiring that $id_\mathrm{p} \in dom(\gamma)$). Also note that compilation is the identity for expressions $sexp$, since the variables that represent pointers contained in $sexp$ are automatically converted to address capabilities by the compilation.

The auxiliary judgments $\tau_\mathrm{s} \rightsquigarrow_{\mathrm{CompileType}} \tau_\mathrm{t}$, which compiles a source type τ_s to the corresponding target type τ_t, and $\tau_\mathrm{s} \rightsquigarrow_{\mathrm{def}} v$, which returns the default value v for the type τ_s, are first defined in Figures 5.8 and 5.9 respectively.

$$\frac{(\textsc{CompileInt})}{\mathrm{int} \rightsquigarrow_{\mathrm{CompileType}} \mathrm{int}} \qquad \frac{(\textsc{CompileSrcPtr})}{\tau* \rightsquigarrow_{\mathrm{CompileType}} \tau*_0}$$

$$\frac{\tau_1 \rightsquigarrow_{\mathrm{CompileType}} \tau_1' \quad \ldots \quad \tau_k \rightsquigarrow_{\mathrm{CompileType}} \tau_k' \quad (\textsc{CompileTuple})}{(\tau_1, \ldots, \tau_k) \rightsquigarrow_{\mathrm{CompileType}} (\tau_1', \ldots, \tau_k')}$$

Figure 5.8: Inference rules defining $\tau_\mathrm{s} \rightsquigarrow_{\mathrm{CompileType}} \tau_\mathrm{t}$.

$$\frac{(\textsc{DefInt})}{\mathrm{int} \rightsquigarrow_{\mathrm{def}} 0} \qquad \frac{(\textsc{DefPtr})}{\tau* \rightsquigarrow_{\mathrm{def}} \mathrm{null}} \qquad \frac{\tau_1 \rightsquigarrow_{\mathrm{def}} def_1 \quad \ldots \quad \tau_k \rightsquigarrow_{\mathrm{def}} def_k \quad (\textsc{DefTuple})}{(\tau_1, \ldots, \tau_k) \rightsquigarrow_{\mathrm{def}} (def_1, \ldots, def_k)}$$

Figure 5.9: Inference rules defining $\tau_\mathrm{s} \rightsquigarrow_{\mathrm{def}} v$.

$$\frac{\begin{array}{c}\tau \rightsquigarrow_{\mathrm{def}} v \qquad \tau \rightsquigarrow_{\mathrm{CompileType}} \tau' \qquad n, \mathit{id}_\ell \text{ fresh}\\ \mathit{id}_{\mathrm{p}} \in \mathrm{dom}(\gamma) \qquad \gamma' = \gamma[\mathit{id}_{\mathrm{p}} : \mathit{id}_\ell]\end{array}}{\begin{array}{c}\{\mathit{sexp}_\gamma > 0\}_\gamma\ \mathit{id}_{\mathrm{p}} = \mathrm{malloc}(\mathit{sexp} * \mathrm{sizeof}(\tau))\\ \{n : \mathit{id}_\ell \mapsto_\tau \mathrm{repeat}(\mathit{sexp}_\gamma, v)\}_{\gamma'}\\ \rightsquigarrow \begin{array}{c}\tau' * n;\ n = \mathrm{malloc}(\mathit{sexp} * \mathrm{sizeof}(\tau'));\\ \mathit{id}_{\mathrm{p}} = \mathrm{addr}(n)\end{array}\end{array}}\ (\textsc{Malloc})$$

$$\frac{}{\begin{array}{c}\{P\}\ \mathrm{guard}(\mathit{sexp})\ \{P * \mathit{sexp}_\gamma\}\\ \rightsquigarrow \mathrm{guard}(\mathit{sexp})\end{array}}\ (\textsc{Guard})$$

$$\frac{n', n'' \text{ fresh} \qquad \tau \rightsquigarrow_{\mathrm{CompileType}} \tau'}{\begin{array}{c}\{n : \mathit{exp}_{\mathrm{a}} \mapsto_\tau l\\ * \mathrm{length}(l) == \mathit{exp}_{\mathrm{l}} * 0 < \mathit{sexp}_\gamma < \mathit{exp}_{\mathrm{l}}\}_\gamma\\ //@\mathrm{split}\ n[\mathit{sexp}]\ \{n' : \mathit{exp}_{\mathrm{a}} \mapsto_\tau \mathrm{take}(l, 0, \mathit{sexp}_\gamma)\\ * n'' : (\mathit{exp}_{\mathrm{a}} + \mathit{sexp}_\gamma) \mapsto_\tau \mathrm{take}(l, \mathit{sexp}_\gamma, \mathit{exp}_{\mathrm{l}})\}_\gamma\\ \rightsquigarrow \tau' * n'; \tau' * n''; \{n', n''\} = \mathrm{split}(n, \mathit{sexp})\end{array}}\ (\textsc{Split})$$

$$\frac{\begin{array}{c}\mathit{id}_{\mathrm{p}} \in \mathrm{dom}(\gamma)\\ \gamma' = \gamma[\mathit{id}_{\mathrm{p}} : \mathit{sexp}_\gamma]\end{array}}{\begin{array}{c}\{\}_\gamma\ \mathit{id}_{\mathrm{p}} = \mathit{sexp}\ \{\}_{\gamma'}\\ \rightsquigarrow \mathit{id}_{\mathrm{p}} = \mathit{sexp}\end{array}}\ (\textsc{VarAsgn})$$

$$\frac{\begin{array}{c}\mathit{id}_{\mathrm{p}} \notin \mathrm{dom}(\gamma) \qquad \tau \rightsquigarrow_{\mathrm{def}} v\\ \tau \rightsquigarrow_{\mathrm{CompileType}} \tau' \qquad \gamma' = \gamma[\mathit{id}_{\mathrm{p}} : v]\end{array}}{\{\}_\gamma\ \tau\ \mathit{id}_{\mathrm{p}}\ \{\}_{\gamma'} \rightsquigarrow \tau'\ \mathit{id}_{\mathrm{p}}}\ (\textsc{VarDecl})$$

$$\frac{\begin{array}{c}\{P\}_\gamma\ \mathit{sstm}\ \{Q\}_{\gamma'} \rightsquigarrow p\\ \mathrm{CN}(Q) = \overline{n}\end{array}}{\begin{array}{c}\{P\}_\gamma\ \mathit{sstm};\ \mathrm{return}\ \{\overline{\mathit{sexp}}\}\\ \{Q * \overline{\mathit{result}} == \overline{\mathit{sexp}_{\gamma'}}\}_{\gamma'}\\ \rightsquigarrow p; \mathrm{return}\ (\overline{\mathit{sexp}}, \overline{n})\end{array}}\ (\textsc{Return})$$

$$\frac{}{\begin{array}{c}\{n : \mathit{id}_{\mathrm{p},\gamma} \mapsto \mathit{exp} * \mathrm{length}(\mathit{exp}) == \mathit{exp}_{\mathrm{l}}\\ * 0 \le \mathit{sexp}_{1,\gamma} < \mathit{exp}_{\mathrm{l}}\}_\gamma\\ \mathit{id}_{\mathrm{p}}[\mathit{sexp}_1] = \mathit{sexp}_2\\ \{n : \mathit{id}_{\mathrm{p},\gamma} \mapsto \mathrm{update}(\mathit{exp}, \mathit{sexp}_{1,\gamma}, \mathit{sexp}_{2,\gamma})\}_\gamma\\ \rightsquigarrow n[\mathit{sexp}_1] = \mathit{sexp}_2\end{array}}\ (\textsc{ArrayMut})$$

$$\frac{\begin{array}{c}\Sigma(f) = \{\mathit{PRE}_f, \mathit{POST}_f, \overline{\mathit{id}_{\mathrm{arg}}}\} \qquad \mathit{PRE}_f \approx_{\mathrm{Names}} \mathit{PRE}\\ \mathit{POST}_f \approx_{\mathrm{Names}} \mathit{POST} \qquad \overline{\mathit{id}} \in \mathrm{dom}(\gamma) \qquad \gamma' = \gamma[\overline{\mathit{id}} : \overline{\mathit{id}_{\mathrm{res}}}]\\ [\mathit{subst}_{\mathrm{pre}}] = [\overline{\mathit{id}_{\mathrm{arg}}} \mapsto \overline{\mathit{sexp}_\gamma}] \qquad [\mathit{subst}_{\mathrm{post}}] = [\mathit{subst}_{\mathrm{pre}}][\overline{\mathit{result}} \mapsto \overline{\mathit{id}_{\mathrm{res}}}]\\ \overline{\mathit{id}_{\mathrm{res}}}, \overline{n} \text{ fresh} \qquad \mathrm{CN}(\mathit{PRE}) = \overline{m} \qquad \mathit{POST} \rightsquigarrow_{\mathrm{resDecl}} \overline{\tau_n\ n}\end{array}}{\begin{array}{c}\{\mathit{PRE}[\mathit{subst}_{\mathrm{pre}}]\}_\gamma\ \overline{\mathit{id}} = f(\overline{\mathit{sexp}})\ \{\mathit{POST}[\mathit{subst}_{\mathrm{post}}]\}_{\gamma'}\\ \rightsquigarrow \overline{\tau_n\ n};\ \{\overline{\mathit{id}}, \overline{n}\} = f_{\mathrm{comp}}(\overline{\mathit{sexp}}, \overline{m})\end{array}}\ (\textsc{FApp})$$

Figure 5.10: Basic separation logic rules that can be extended to compilation rules.

The remainder of this section consists of a discussion of the depicted basic separation logic rules and compilation rules. Separation logic rules are illustrated by referencing lines from Figure 5.6. Compilation rules are illustrated using a combination of source code lines from Figure 5.6 and lines from the compiled verified function f_{comp} in Figure 5.5.

The Malloc rule assigns a fresh logical variable id_ℓ to id_p in γ and creates a new array resource n, consisting of the default value v repeated $sexp_\gamma$ times. In the corresponding Malloc compilation rule, the variable $\tau' *\ n$ is declared and assigned the malloced linear capability, i.e. the resource corresponding to id_p in the source language. This target-level assignment to the variable n clearly makes it the reified version of the source resource n (also freshly introduced by the rule). As id_p is itself merely a permissionless address on the target level, we assign it using the addr function, maintaining the correspondence between a and n from the separation logic proof. The Malloc rule is demonstrated on lines 18-20, where a new resource l is created. Lines 18-20 are compiled to lines 7-8 in f_{comp}.

The Split and Join rules for both array resources and range resources are dual, with the difference that Split has to check whether the given splitting index to split on is in bounds, whereas Join has to check memory adjacency of the two resources it is given. Given the similarities, only the array version of the split rule is detailed in Figure 5.10. Notice how this rule indeed performs the same operation on separation logic resources that the target operational semantics rule Split in Figure 5.4 performed on linear capabilities. Consequently, the Split compilation rule simply mirrors the source level split statement in the target language, using the built-in split operation on the corresponding reified linear capabilities. The Split rule is demonstrated on lines 5-7, where resource m is split. Lines 5-7 are compiled to line 3 in f_{comp}.

The rule VarDecl and VarAsgn prove variable declaration and assignment, respectively. Their compilation rules are straightforward, apart from the change in type in VarDecl. The VarDecl rule is demonstrated on e.g. lines 1-3, which compile to the first declaration on line 2 in f_{comp}.

The rules ArrayMut and ArrayLkup are very similar, so Figure 5.10 only shows the former. The mutation of source arrays is again compiled to the same action on the corresponding reified resource. Both rules are demonstrated on lines 20-22 and lines 16-18, respectively. The ArrayMut rule enforces the logical address of the resource n to be exactly equal to $id_{\text{prog},\gamma}$, rather than allowing for some additional offset, out of simplicity considerations (ArrayLkup enforces something similar). For this reason, line 17 contains $(a + 1)[0]$ and not $a[1]$; $(a + 1)_\gamma$ equals the logical address of the resource m_2, whereas a_γ does not. These lines respectively compile to lines 9 and the end of line 6 in f_{comp}.

The GUARD rule adds the asserted conditions to the symbolic heap and compilation is the identity.

Function application FAPP is the most intricate basic rule. The variable Σ denotes a component-wide function environment that contains the contract and argument names for each function f (including imported functions). The caller does not need to match the called function's contract exactly: we can allow outer resource names to differ. That is why the relation $\approx_{\text{Names}}$ is used, to enforce equality up to resource names of pre- and postcondition. In the caller's pre- and postcondition *PRE* and *POST*, the concrete logical expressions $\overline{sexp_\gamma}$ used in the function call are substituted for the arguments $\overline{id}_{\text{arg}}$, instantiating the callee's contract with the caller-provided arguments. Additionally, in the caller's postcondition, the privileged $\overline{result}$ variables are substituted with fresh logical variables $\overline{id_{\text{res}}}$, linked to $\overline{id}$ in γ'. Fresh resource names $\overline{n}$ are required to avoid name clashes.

Given this rule, the FAPP compilation can now be discussed. The resource names $\overline{m}$ in *PRE* will be reified and are extracted using the function CN. The resource names $\overline{n}$ in *POST* have to be fresh and will hence need to be declared in the compiled code first, before reification. We use an auxiliary judgment $assert \rightsquigarrow_{\text{ResDecl}} \overline{\tau_n\ n}$ that extracts all resource names $\overline{n}$ in *assert* and pairs them with their reified types $\overline{\tau_n}$ in target-level declarations $\overline{\tau_n\ n}$. This judgment extracts the correct names $\overline{n}$ from *POST* and immediately tells us what declarations $\overline{\tau_n\ n}$ to create. The compiled function call contains the reified versions of the precondition resources $\overline{m}$ as extra arguments and receives the reified postcondition resources $\overline{n}$ as extra return values. The reason each function f is renamed to f_{comp} during compilation is related to incall and outcall stubs and will become more clear in Section 5.4.4.

The FAPP rule is illustrated on lines 10-12 and lines 22-24. For lines 10-12 for example, $[subst_{\text{pre}}] == [a \mapsto a{+}1]$ and $[subst_{\text{post}}] == [a \mapsto a{+}1][result \mapsto x]$, where $[res : x]$ is substituted in the environment after the call. Given these substitutions, we can see that the preconditions indeed only differ in the chunk names m_2 versus m; hereby satisfying $\approx_{\text{Names}}$. The same holds for the postconditions (where the returned resource has already been omitted by CONSEQ on line 12). Lines 10-12 are hence compiled to line 5 in f_{comp}, where $add1_{\text{comp}}$ denotes the outcall stub for *add1*.

The RETURN rule forms a special case because return is not a source statement; it appears exactly once at the end of each function body. Since SEQ can only sequence source statements, the RETURN rule has to manually construct a new proof from a previous one. Conceptually, however, RETURN is a basic rule. Given a proof of *sstm*, the RETURN rule introduces the privileged $\overline{result}$ logical variables to the symbolic heap, equaling the returned expressions. The return compilation rule produces a target return statement, which additionally returns all reified resources $\overline{n}$. The CONSEQ rule reshapes the contract Q after the return into the function body's postcondition (in this phase, leaking resources is disallowed, because the set of reified resources $\overline{n}$ is already fixed). Lines 25-27 demonstrate the return rule and are compiled to line 11 in f_{comp}. No returned variables are added because f leaks its resources. Notice that line 27 follows from Return's postcondition $\{x == (c == 0\ ?\ a_1 + 1 : -a_1 + 1) * \mathit{result} == x\}$ by the Conseq rule.

5.4.3 Higher-Level Rules

Given a separation-logic proof of a function's body, constructed as in the previous subsections, we now introduce rules that define the notion of separation logic proof $\vdash$ for entire functions, components and source programs, as these higher-level structures are what we are most interested in compiling. The higher-level rules are ImplFVerif, ContFVerif, CompVerif and ProgVerif, presented in Figure 5.11 and discussed below. For all compilation-related judgments $\rightsquigarrow_{\mathrm{X}}$, we define the following tuple-based shorthand: $\vdash \overline{s_i} \rightsquigarrow_{\mathrm{X}} \overline{t_i} \triangleq \forall i. \vdash s_i \rightsquigarrow_{\mathrm{X}} t_i$.

First, we have the rules for implemented functions ImplFVerif and context functions ContFVerif. The main difference between these rules is that ContFVerif does not reference any function body, as expected, whereas ImplFVerif requires a proof of the function body to construct a function proof $\vdash$. The precondition environment of this proof is $[\overline{id_{\mathrm{arg}} : id_{\mathrm{arg}}}]$, since our separation logic contract preconditions always implicitly map the function arguments $\overline{id_{\mathrm{arg}}}$ to logical variables of the same names $\overline{id_{\mathrm{arg}}}$. Non-coincidentally, this environment is the starting environment in Figure 5.6. This initial environment explains how we can allow function contracts to be entirely logical assertions, but still reference function arguments $\overline{id_{\mathrm{arg}}}$.

The corresponding compilation rules both use an auxiliary judgment $\mathit{sfunc} \rightsquigarrow_{\mathrm{Decl}} \mathit{tfunc}$ that compiles a function declaration *sfunc* to a declaration *tfunc* by reifying all resources in the given pre- and postcondition into arguments and return values respectively, and compiling existing argument and return types using $\rightsquigarrow_{\mathrm{CompileType}}$. The rule ImplFVerif also changes any implemented function f's name to f_{comp} during compilation, again for stub-related reasons explained in Section 5.4.4. The proof of Figure 5.6 suffices to construct a proof $\vdash$ of f using ImplFVerif, which can then be compiled to the target-level function f_{comp} in Figure 5.5. The declaration of *add1*, on the other hand, is compiled to the declaration of *add1* using ContFVerif.

The component verification rule CompVerif allows verification and compilation of entire components. A component *scomp* has a proof $\vdash$ *scomp* if it is well-formed (denoted by $\vdash_{\mathrm{WF}}$ *scomp*, which includes some restrictions mentioned in Section 5.4.4) and if all its implemented and (exported and imported) context functions have proofs. A compiled component is constructed from the compilation of its functions. The compilation rules $\rightsquigarrow_{\mathrm{Incall}}$ and $\rightsquigarrow_{\mathrm{Outcall}}$ both subsume the ContFVerif rule, and

$$\frac{\begin{array}{c} \mathit{isfunc} = \overline{\tau_{\mathrm{ret}}}\ f(\overline{\tau_{\mathrm{arg}}\ \mathit{id}_{\mathrm{arg}}})//\mathrm{@pre}\ \mathit{PRE}\ //\mathrm{@post}\ \mathit{POST}\ \ \{\mathit{sstm}; \mathrm{return}\ \overline{\mathit{sexp}}\} \\ \mathit{isfunc} \leadsto_{\mathrm{Decl}} \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f(\overline{\tau'_{\mathrm{arg}}\ \mathit{id}_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m}) \\ \{\mathit{PRE}\}_{[\overline{\mathit{id}_{\mathrm{arg}}:\mathit{id}_{\mathrm{arg}}}]}\ \mathit{sstm}; \mathrm{return}\ \overline{\mathit{sexp}}\ \{\mathit{POST}\}_{\gamma} \leadsto p_1; \mathrm{return}\ \{\overline{\mathit{texp}}, \overline{n}\} \end{array}}{\vdash \mathit{isfunc} \leadsto \begin{array}{c} \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f_{\mathrm{comp}}(\overline{\tau'_{\mathrm{arg}}\ \mathit{id}_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m}) \\ \{p_1; \mathrm{return}\ \{\overline{\mathit{texp}}, \overline{n}\}\} \end{array}}\ (\textsc{ImplFVerif})$$

$$\frac{\begin{array}{c} \mathit{csfunc} = \overline{\tau_{\mathrm{ret}}}\ f(\overline{\tau_{\mathrm{arg}}\ \mathit{id}_{\mathrm{arg}}})//\mathrm{@pre}\ \mathit{PRE}\ //\mathrm{@post}\ \mathit{POST} \\ \mathit{csfunc} \leadsto_{\mathrm{Decl}} \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f(\overline{\tau'_{\mathrm{arg}}\ \mathit{id}_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m}) \end{array}}{\vdash \mathit{csfunc} \leadsto \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f(\overline{\tau'_{\mathrm{arg}}\ \mathit{id}_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m})}\ (\textsc{ContFVerif})$$

$$\frac{\begin{array}{c} \mathit{scomp} = \overline{\mathit{isfunc}}//\mathrm{@import}\ \overline{\mathit{csfunc}_{\mathrm{i}}}\ //\mathrm{@export}\ \overline{\mathit{csfunc}_{\mathrm{e}}} \\ \vdash_{\mathrm{WF}} \mathit{scomp} \qquad \vdash \overline{\mathit{csfunc}_{\mathrm{i}}} \leadsto_{\mathrm{Outcall}} \overline{\mathit{ctfunc}_{\mathrm{i}}, \mathit{stub}_{\mathrm{out}}} \\ \vdash \overline{\mathit{isfunc}} \leadsto \overline{\mathit{itfunc}} \qquad \vdash \overline{\mathit{csfunc}_{\mathrm{e}}} \leadsto_{\mathrm{Incall}} \overline{\mathit{ctfunc}_{\mathrm{e}}, \mathit{stub}_{\mathrm{in}}} \end{array}}{\vdash \mathit{scomp} \leadsto (\overline{\mathit{itfunc}}\ \overline{\mathit{stub}_{\mathrm{out}}}\ \overline{\mathit{stub}_{\mathrm{in}}})\ //\mathrm{@import}\ \overline{\mathit{ctfunc}_{\mathrm{i}}}\ //\mathrm{@export}\ \overline{\mathit{ctfunc}_{\mathrm{e}}}}\ (\textsc{CompVerif})$$

$$\frac{\mathit{sprog} = \overline{\mathit{scomp}}\ //\mathrm{@main} = \mathit{id} \qquad \vdash_{\mathrm{WF}} \mathit{sprog} \qquad \vdash \overline{\mathit{scomp}} \leadsto \overline{\mathit{tcomp}}}{\vdash \mathit{sprog} \leadsto \overline{\mathit{tcomp}}\ //\mathrm{@main} = \mathit{id}}\ (\textsc{ProgVerif})$$

Figure 5.11: Higher-level separation logic rules that can be extended to compilation rules.

additionally generate the incall and outcall stubs for exported and imported functions, respectively. Both these rules are discussed in the next subsection.

As an example, given the proofs of f and *add1* constructed in the previous paragraph, the CompVerif rule proves the source component in Figure 5.5, which has f as an internal function, an empty export list and the declaration of *add1* as the import list. The source component is compiled to the target component in the bottom half of Figure 5.5, where $\mathit{add1}_{\mathrm{comp}}$ is the outcall stub resulting from the application of $\leadsto_{\mathrm{Outcall}}$ on *add1*. In a real-life setting, we would have made f callable by including it in the export list of the source component, such that an incall stub f would have been created by $\leadsto_{\mathrm{Incall}}$ as well. We omitted this detail for simplicity's sake.

Finally, the program verification rule ProgVerif allows verification and compilation of entire programs. A program *sprog* has a proof $\vdash$ *sprog* if it is well-formed ($\vdash_{\mathrm{WF}}$ *sprog*, which e.g. states that every imported function should be exported by another component) and if all of its components have a proof. The compilation of a program is constructed from the compilations of its components.

5.4.4 Stub Compilation

As illustrated in Section 5.2, we require checking functions or stubs in our compiled code to enforce separation logic contracts at trust boundaries, both when receiving an untrusted incall to an exported function and performing an outcall to an untrusted imported function. A verified component of course trusts itself, requiring no stubs when internal calls are performed, as no trust boundary is crossed. We call functions that require the generation of stubs during compilation, i.e. functions that are imported or exported by any module, *boundary functions*.

This section discusses how our compiler generates outcall stubs specifically, by means of the Outcall compilation rule in Figure 5.12. The Incall compilation rule that generates the incall stubs for exported functions is an analogous but simpler version of Outcall and hence not detailed. No Outcall or Incall separation logic rule exists, since stubs are not part of the source code; separation logic contracts are enforced at trust boundaries by the separation logic proof itself.

The Outcall rule generates the outcall stubs for a component's imported functions by defining the previously mentioned compilation rule $\mathit{csfunc} \rightsquigarrow_{\mathrm{Outcall}} \mathit{ctfunc}, \mathit{stub}_{\mathrm{outcall}}$ that both compiles a context function *csfunc* to *ctfunc* using ContFVerif and generates an outcall stub $\mathit{stub}_{\mathrm{outcall}}$ for it. The latter is a wrapper around the outcall to f and reifies f's postcondition in the form of guard statements that check, after f has returned, whether it has upheld its postcondition.

Before generating outcall stubs for imported boundary functions, we make three assumptions on the form of the contracts of boundary functions. These assumptions allow us to easily generate both types of stubs by means of contract reification.

First, we only allow fixed-size, non-conditional array resources $n : \mathit{exp} \mapsto [\mathit{exp}_1, \ldots, \mathit{exp}_k]$ to appear in boundary function contracts. This allows us to do away with quantification in boundary contracts, making the reification of conditions in stubs easier, since every condition ranges over a predetermined set of variables. Since boundary contracts do not contain nested or conditional resources, pre- and postcondition symbolic heaps *PRE* and *POST* are separable into a spatial heap $\overline{m} : \mathit{PRE}_{\mathrm{s}}$ or $\overline{n} : \mathit{POST}_{\mathrm{s}}$, consisting of a sequence of separating-conjunction separated fixed-size array resources (whose names $\overline{n}$ and $\overline{m}$ we externalize in our notation), and a pure heap $\mathit{PRE}_{\mathrm{p}}$ or $\mathit{POST}_{\mathrm{p}}$, consisting of pure conditions. The Outcall rule will make handy use of this separability. Possible measures to weaken this restriction are future work.

Second, we require boundary functions to have *linear contracts*, in the sense that all argument names, all logical variable names in PRE_{s} and $POST_{\mathrm{s}}$ and the set of names $\overline{result}$ must be mutually distinct and cannot contain duplicates. This makes any otherwise implicit conditions in PRE_{s} and $POST_{\mathrm{s}}$ explicit in PRE_{p} and $POST_{\mathrm{p}}$, and hence easier to check. This assumption can be made without loss of generality, as non-linear contracts can easily be linearized. For example, the programmer-written contract for *add1* in Figure 5.5 is linearized to:

$$\begin{aligned}
&//@\mathrm{pre}\ m : a^{\mathrm{pre}} \mapsto_{\mathrm{int}} [a_1^{\mathrm{pre}}] * a^{\mathrm{pre}} == a \\
&//@\mathrm{post}\ n : a^{\mathrm{post}} \mapsto_{\mathrm{int}} [a_1^{\mathrm{post}}] * result == a_1^{\mathrm{post}} + 1 \\
&\qquad\qquad\qquad * a^{\mathrm{post}} == a * a_1^{\mathrm{post}} == a_1^{\mathrm{pre}}\ .
\end{aligned}$$

Lastly, we require boundary functions not to introduce any new logical variables (except for $\overline{result}$) in their pure heaps PRE_{p} and $POST_{\mathrm{p}}$. This will make the constraints in PRE_{p} and $POST_{\mathrm{p}}$ easier to reify into program-level guards, as all their logical variables either correspond to arguments, result variables or spatial values in the symbolic heap. For example, in the above linearized contract we can easily access the values for a, a^{pre}, a_1^{pre}, a^{post}, a_1^{post} and *result* in the compiled code. The restrictions imposed by these last two assumptions are included as part of the component well-formedness $\vdash_{\mathrm{WF}}$ *scomp* in CompVerif.

An outcall stub then needs to generate code in order to check any constraints present in the untrusted function's postcondition *POST*.

We first investigate what information from $\overline{m} : PRE_{\mathrm{s}}$ and $\overline{n} : POST_{\mathrm{s}}$ we need to reify to be able to check *POST*. Both $\overline{m} : PRE_{\mathrm{s}}$ and $\overline{n} : POST_{\mathrm{s}}$ are linear and hence use fresh variable names that can reappear in conditions in $POST_{\mathrm{p}}$. These variables hence need to be reified, i.e. declared and assigned the right values, so they can be used when reifying $POST_{\mathrm{p}}$'s conditions. Additionally, any constraints present in the linear spatial heap $\overline{n} : POST_{\mathrm{s}}$ need to be checked in the target language. Remember that an outcall stub solely checks the postcondition. The only information to check is that none of the reified resources $\overline{n}$ can be null, together with the fact that each reified resource n has its correct fixed length k. Both these checks need to be performed by a guard statement for each n. We need a way to reify the aforementioned checks *check*, declarations *decl* and assignments *assign* for a given spatial assertion $assert_{\mathrm{s}}$ (i.e. $\overline{m} : PRE_{\mathrm{s}}$ or $\overline{n} : POST_{\mathrm{s}}$). This is the function of the auxiliary compilation rule $assert_{\mathrm{s}} \rightsquigarrow_{\mathrm{s}} (check, decl, assign)$, defined by ResourceReify and SepConjCReify in Figure 5.12. The *checks* generated for $\overline{m} : PRE_{\mathrm{s}}$ are simply discarded.

Next, we investigate what information from PRE_{p} and $POST_{\mathrm{p}}$ we need to reify to check *POST*. Neither one is allowed to introduce fresh variables, due to the third assumption we made above. Therefore, no declarations or assignments will be reified; only checks. Since outcall stubs only check postconditions, we can disregard PRE_{p}.

$$\frac{}{exp \rightsquigarrow_{\mathrm{p}} \mathrm{guard}(exp)}\;(\textsc{ConditionReify}) \qquad \frac{C \rightsquigarrow_{\mathrm{p}} c_1 \qquad C' \rightsquigarrow_{\mathrm{p}} c_2}{C * C' \rightsquigarrow_{\mathrm{p}} c_1; c_2}\;(\textsc{SepConjPReify})$$

$$\frac{\begin{array}{c} \tau \rightsquigarrow_{\mathrm{CompileType}} \tau' \\ check = (\mathrm{guard}(n \mathrel{!=} \mathrm{null}); \mathrm{guard}(\mathrm{length}(n) == k)) \\ decl = (\tau *_0 x;\ \tau'\ x_1;\ \ldots;\ \tau'\ x_k) \\ assign = \\ (x = \mathrm{addr}(n); x_1 = n[0];\ \ldots;\ x_k = n[k-1]) \end{array}}{n : x \mapsto_{\tau} [x_1, \ldots, x_k] \rightsquigarrow_{\mathrm{s}} (check, decl, assign)}\;(\textsc{ResourceReify}) \qquad \frac{\begin{array}{c} C \rightsquigarrow_{\mathrm{s}} (c_1, d_1, a_1) \\ C' \rightsquigarrow_{\mathrm{s}} (c_2, d_2, a_2) \end{array}}{\begin{array}{c} C * C' \\ \rightsquigarrow_{\mathrm{s}} (c_1; c_2, d_1; d_2, a_1; a_2) \end{array}}\;(\textsc{SepConjCReify})$$

$$\frac{\begin{array}{c} f_i = \overline{\tau'_{\mathrm{ret}}}\ f\ (\overline{\tau_{\mathrm{arg}}\ id_{\mathrm{arg}}}) \\ //@\mathrm{pre}\ \overline{m} : PRE_{\mathrm{s}} * PRE_{\mathrm{p}}\ //@\mathrm{post}\ \overline{n} : POST_{\mathrm{s}} * POST_{\mathrm{p}} \\ p_i = \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f\ (\overline{\tau'_{\mathrm{arg}}\ id_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m}) \qquad \vdash f_i \rightsquigarrow p_i \\ \overline{m} : PRE_{\mathrm{s}} \rightsquigarrow_{\mathrm{s}} (\_, d_{\mathrm{pre}}, a_{\mathrm{pre}}) \qquad \overline{n} : POST_{\mathrm{s}} \rightsquigarrow_{\mathrm{s}} (cs_{\mathrm{post}}, d_{\mathrm{post}}, a_{\mathrm{post}}) \\ POST_{\mathrm{p}} \rightsquigarrow_{\mathrm{p}} cp_{\mathrm{post}} \\ stub_{\mathrm{outcall}} = \left\{ \begin{array}{l} \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f_{\mathrm{comp}}\ (\overline{\tau'_{\mathrm{arg}}\ id_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m})\{ \\ d_{\mathrm{pre}}; a_{\mathrm{pre}}; \overline{\tau'_{\mathrm{ret}}\ result}; \overline{\tau_{\mathrm{post}}\ n}; \\ (\overline{result, \overline{n}}) = f(\overline{id_{\mathrm{arg}}}, \overline{m}); \\ cs_{\mathrm{post}}; d_{\mathrm{post}}; a_{\mathrm{post}}; cp_{\mathrm{post}}; \\ \mathrm{return}\ (\overline{result, \overline{n}})\} \end{array} \right\} \end{array}}{\vdash f_i \rightsquigarrow_{\mathrm{Outcall}} p_i, stub_{\mathrm{outcall}}}\;(\textsc{Outcall})$$

Figure 5.12: Compilation rules for generating outcall stubs

All constraints present in $POST_{\mathrm{p}}$ can simply be reified to guard statements over the identical constraints, due to the three form assumptions on contracts we made above and since we already declared and assigned all variables occurring in these constraints when we reified $\overline{m} : PRE_{\mathrm{s}}$ and $\overline{n} : POST_{\mathrm{s}}$ in the previous paragraph. The auxiliary compilation rule $assert_{\mathrm{p}} \rightsquigarrow_{\mathrm{p}} check$, defined by ConditionReify and SepConjPReify in Figure 5.12, generates these checks *check* when given a pure assertion $assert_{\mathrm{p}}$ (i.e. $POST_{\mathrm{p}}$ in this case).

The Outcall compilation rule in Figure 5.12 integrates $\rightsquigarrow_{\mathrm{s}}$, $\rightsquigarrow_{\mathrm{p}}$ and ContFVerif to create the outcall stub $stub_{\mathrm{outcall}}$. Precondition-related declarations d_{pre} and assignments a_{pre} happen before the function call to f, since the reified resources $\overline{m}$ might be altered by f. Postcondition-related declarations d_{post}, assignments a_{post} and checks cs_{post} and cp_{post} happen after the call.

The reason the rule ImplFVerif renamed every function f to f_{comp} had everything to do with stubs. First of all, any incall stub generated for an exported source function f can now simply be called f and internally call the compiled target function f_{comp}, so that the names of a component's exported functions do not change during compilation. Conversely, outcall stubs for imported functions f are named f_{comp} as well, as Outcall demonstrates, so that the FApp rule does not need to know whether an internal or imported function is being called in order to derive the compiled function's name.

The outcall stub $add1_{\text{comp}}$ for *add1* in Figure 5.5 gives an example of the generated stub $stub_{\text{outcall}}$ in the Outcall compilation rule (compiled using the linearized contract above). As always, the stub has the same declaration (bar function name) as the function *add1* it wraps. The declarations d_{pre} can be found on line 2 and the assignments a_{pre} on line 3. The declarations d_{post} are generated on line 7, assignments a_{post} on line 8 and checks cs_{post} and cp_{post} on line 6 and lines 9-10, respectively.

5.5 Full Abstraction

This section summarizes the full abstraction proof for the compiler presented in Section 5.4. The full abstraction proof by itself takes up roughly 80 pages in the technical report, and is therefore too long and detailed to include in this paper. This section and Sections 5.6 and 5.7 hence summarize the essential concepts in an example-driven fashion, after which the reader should be able to digest the technical report, should they wish to read it.

In the following, notions relating to the source and target language are typeset in **green** and pink respectively. The first Subsection 5.5.1 formally defines both directions of full abstraction. Section 5.5.2 discusses why proving full abstraction for our compiler is useful, by highlighting interesting properties that our compiler preserves. Subsection 5.5.3 motivates the need for a back-translation and illustrates how both directions of full abstraction can be proved by proving equi-termination between source and target code. Subsection 5.5.4 further reduces these proofs of equi-termination to a proof of relatedness under specific adequate relations between source and target code.

5.5.1 Full Abstraction Definition

To define our notion of full abstraction, we require a notion of *behavioral equivalence*. As is standard in the literature, we define behavioral equivalence to be *contextual equivalence* [1, 121, 108, 45]. Terms x and x' are contextually equivalent, denoted $x \simeq_{\text{ctx}} x'$, if $\forall C : C[x] \Downarrow \Leftrightarrow C[x'] \Downarrow$ where $\Downarrow$ denotes termination of execution and C

is any program context with a hole that x and x' can be plugged into. Both x and x' are either source or target components in our case. A context C consists of two parts in both source and target languages: a *component context* $\mathfrak{C}_s$ or $\mathfrak{C}_t$, which is a sequence of components, and a *main function identifier*, denoted by the metavariable *id*, identifying the main function to execute when starting program execution. A context is hence denoted $(\mathfrak{C}, id)$ and an entire program $\mathfrak{C}[x]//@\mathrm{main} = id$. In our source language, the notion of *plugging* from the contextual equivalence definition above also requires (in addition to the program well-formedness constraints denoted by $\vdash_{\mathrm{WF}}$ *scomp* in Figure 5.11 in the previous section) that given the source component proof $\vdash \mathrm{s}$ and the context $(\mathfrak{C}_s, id)$, a program proof $\vdash \mathfrak{C}_s[\mathrm{s}]//@\mathrm{main} = id$ exists. The notion of plugging in the target language solely requires program well-formedness, expressed by a similar judgment $\vdash_{\mathrm{WF}}$ *tcomp* for target programs, defined in the technical report.

Full abstraction is then defined as the reflection and preservation of contextual equivalence $\simeq_{\mathrm{ctx}}$ [1]. Given source components s and s' and target components t and t', we have that compilation is fully abstract iff $\vdash \mathrm{s} \rightsquigarrow \mathrm{t}$ and $\vdash \mathrm{s}' \rightsquigarrow \mathrm{t}'$ implies that $(\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}' \Leftrightarrow \mathrm{s} \simeq_{\mathrm{ctx}} \mathrm{s}')$. This statement depends on the chosen proofs $\vdash$ of s and s', but has to hold for *any* such choice. Notice how our formulation of full abstraction does not make a distinction between code that gets stuck and code that diverges. In other words, diverging source code could in theory be compiled to target code that gets stuck. This is, however, not a real concern; since our compiler does not alter control flow, it should be easy to prove that it preserves divergence and stuck-ness individually, if so desired.

Fully abstract compilation proofs are usually split up in a correctness proof direction $\Rightarrow$ that states (by contraposition) that non-equivalent source programs should yield non-equivalent target programs and a security proof direction $\Leftarrow$ that (by contraposition) states that any non-equivalence in the target programs should already have been there in the source programs, and hence attackers have no more power in the absence of contracts than they do in their presence. Both proof directions are summarized by the following equations:

$$\forall\, \mathrm{s}, \mathrm{s}', \mathrm{t}, \mathrm{t}'. \vdash \mathrm{s} \rightsquigarrow \mathrm{t} \;\wedge \vdash \mathrm{s}' \rightsquigarrow \mathrm{t}' \Rightarrow (\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}' \Rightarrow \mathrm{s} \simeq_{\mathrm{ctx}} \mathrm{s}') \qquad (\textsc{Correctness})$$

$$\forall\, \mathrm{s}, \mathrm{s}', \mathrm{t}, \mathrm{t}'. \vdash \mathrm{s} \rightsquigarrow \mathrm{t} \;\wedge \vdash \mathrm{s}' \rightsquigarrow \mathrm{t}' \Rightarrow (\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}' \Leftarrow \mathrm{s} \simeq_{\mathrm{ctx}} \mathrm{s}') \qquad (\textsc{Security})$$

5.5.2 The Guarantees Offered by Full Abstraction

This section illustrates the meaning of the **Security** direction of the compiler full-abstraction proof. Concretely, we show a few examples for which the full abstraction proof implies preservation of some safety and confidentiality properties. Note that we *assume* the equivalence of the source programs in this section, based on an intuitive understanding of our separation logic; in this paper, we have not developed any tools to reason about contextual equivalence of verified source programs, as this issue is mostly orthogonal to secure compilation.

We discuss examples where preservation of useful safety and confidentiality properties follows from our results, not necessarily strictly from the full abstraction property alone. In other words, some results are also based on relatively basic inspection of the compiler and proof structure, in addition to full abstraction of the compiler. Consequently, we cannot use these examples to link full abstraction to the alternative secure compilation criteria of Abate et al. [4]. Additionally, for reasons of simplicity, in this section we will assume the stronger type of contextual equivalence that distinguishes between terminating and stuck programs and that was discussed in the previous section.

First, under the above assumptions about contextual equivalence, the full abstraction *proof* is sufficiently powerful to show preservation of some robust safety properties [152], a class of unary security properties that express safety of a program interacting with an untrusted context. Concretely, following Swasey et al. [152], guard statements can be used to encode desirable safety properties that should hold at certain points during execution[1]. If one can show the equivalence of programs that are identical up to guard statements in the source language, meaning that no (accessible) guard statements fail, then our compiler will ensure that the corresponding compiled programs are still equivalent under an arbitrary target context.

This, however, is insufficient to prove preservation of the robust safety property encoded by the guard: the compiler might well map all terminating programs to diverging programs and vice versa, preserving full abstraction in the process. This *horizontal* view of equivalence is a well-known limitation of using full abstraction, that we already discussed in Section 1.3. As we discussed there, we need vertical knowledge to be able to use full abstraction in practice. Luckily, this vertical relationship between source and target programs is often present in the full abstraction proof. In our case, the vertical equi-termination result from Figure 5.18 suffices to know that our compiler will not misbehave in this way, and that the robust safety property encoded by the guard is indeed preserved.

Figure 5.13 illustrates this approach for a simple program that demonstrates that an adversary (represented by a context function g with a trivial precondition) cannot change the value of encapsulated stack variables. This is expressed by checking, using a guard statement, that the value of local variable x has not changed across the call to g. Similar results hold for heap-allocated variables, and in more complicated settings with richer interactions with g, as long as the variable is not passed to the untrusted context. This example illustrates that our compiler upholds the integrity guarantees

[1] Note that we do not *require* guard statements to express robust safety, but failure is a simple way of creating a context-visible side-effect.

of the separation logic frame rule of Figure 5.7; g can only alter the values of variables mentioned in its contract. Note that our frame rule is somewhat non-standard, in that it also allows framing parts of the environment γ. In this case, $[x : 0]$ is framed. For heap-allocated variables, the framing would be more standard, as the regular points-to chunk $\gamma(x) \mapsto 0$ would be framed off (along with parts of γ).

Program 1	$\simeq_{\text{ctx}}$	**Program 2**	**Context**

void f( )
//@pre true
//@post true {
  int x = 0;
  g( );
  guard(x == 0)
}
		void f( )
		//@pre true
		//@post true {
		  int x = 0;
		  g( )
		}
			void g( )
			//@pre true
			//@post _

Figure 5.13: Illustration of how contextual equivalence can express a robust safety property: equivalence of **Program 1** with guard statements and **Program 2** without them specifies that the guard never fails. The integrity of local variable x is hence upheld.

Program 1	$\simeq_{\text{ctx}}$	**Program 2**	**Context**

void f( )
//@pre true
//@post true {
  int x = g( );
  guard(x == 0)
}
		void f( )
		//@pre true
		//@post true {
		  int x = g( );
		}
			void g( )
			//@pre true
			//@post result == 0

Figure 5.14: Illustration of how contextual equivalence can enforce contracts: equivalence of **Program 1** with guard statements and **Program 2** without them specifies that the guard never fails, and hence the postcondition holds.

As an additional form of robust safety, functional aspects of contracts are also preserved. This can be demonstrated by providing g with a contract that states that, for example, its return value is equal to 0, and checking this condition in one of two programs but not the other, cfr. Figure 5.14.

The aforementioned contract checking work of Agten et al. [7] was already capable of preserving robust safety properties as part of their verified component hardening transformation. However, they lacked support for binary properties, notably 2-hypersafety properties such as confidentiality. Figure 5.15 shows a simple example of how contextual equivalence can be used to specify confidentiality of the local stack frame. Similar results hold for heap-allocated variables, and richer interactions with the context. Concretely, the example illustrates that full abstraction enforces the confidentiality aspects of the frame rule from Figure 5.7; variables that are not present in the contract of an untrusted function (g in this case), should not be able to influence its execution.

Program 1	$\simeq_{\text{ctx}}$	**Program 2**	**Context**

void f( )
//@pre true
//@post true {
  int x = 0;
  g( )
}
		void f( )
		//@pre true
		//@post true {
		  int x = 1;
		  g( )
		}
			void g( )
			//@pre true
			//@post _

Figure 5.15: Illustration of how contextual equivalence can enforce a confidentiality property: equivalence of **Program 1** and **Program 2** specifies that the value of local variable x cannot influence context behavior.

Program	**Context**

void example3_down(int* x)
//@pre n: $x \mapsto_{\text{int}} [x_1]$
//@post n: $x \mapsto_{\text{int}} [x_2]$ {
  *x = 42; //This write is redundant
  f( );
  *x = 13
}
	void f( )
	//@pre true
	//@post true

Figure 5.16: Translation of Jung et al. [77]'s `example3_down` function to our setting. This example illustrates a compiler optimization where a redundant write to the local variable x can be removed, because the function f is not granted access to x.

One use case for our fully abstract compiler is that we can prove that sound compiler optimizations in the source language (i.e., source-level equivalences), do not create additional vulnerabilities in the target language. The type of source-level equivalences that Jung et al. [77] prove for simple compiler optimization passes in Rust, could be fed to our compiler (after translation to our verified C setting), to prove that the same equivalences hold when using linear capabilities. Hence, the source-level optimization did not create additional target-level vulnerabilities. As an example, Figure 5.16 shows a translation of Jung et al.'s `example3_down`-function to our setting. Because our languages lack closures, we consider f to be a regular, statically known, context function.

Just as in Jung et al.'s setting, this program is contextually equivalent to one where the first write has been removed. Since our compiler is fully abstract, we know that this dead store elimination will not introduce additional vulnerabilities at the linear capability level. Differences with the Rust setting are that we do not have a notion of unsafe code at the source level (our notion of unsafety comes from lowering the abstraction level during compilation), and that we do not currently support shared, read-only resources for memory, nor do we have the RustBelt-style lifetimes required to support shared borrows in our logic [78].

In Section 5.9, we speculate about the fact that the proof techniques we use should allow proving preservation of higher-arity security properties as well. We have not yet thought of potential use cases for this stronger result.

5.5.3 Full Abstraction as Source-to-Target Equi-Termination

We first dissect the **Correctness** direction above. To prove $\mathbf{s} \simeq_{\text{ctx}} \mathbf{s}'$, we need to prove (by definition) that $\mathfrak{C}_s[\mathbf{s}]//@\text{main} = id \Downarrow \Leftrightarrow \mathfrak{C}_s[\mathbf{s}']//@\text{main} = id \Downarrow$, given any source context $(\mathfrak{C}_s, id)$ such that we can construct proofs $\vdash \mathfrak{C}_s[\mathbf{s}]//@\text{main} = id$ and $\vdash \mathfrak{C}_s[\mathbf{s}']//@\text{main} = id$ from $\vdash \mathbf{s}$ and $\vdash \mathbf{s}'$. If we can prove that verified code and its compilation equi-terminate, i.e. if it holds for any contexts $(\mathfrak{C}_s, id)$ and $(\mathfrak{C}_t, id)$ and any components $\mathbf{s}$ and t (with $\vdash \mathbf{s} \rightsquigarrow \mathsf{t}$) that:

$$\begin{array}{l}\vdash \mathfrak{C}_s[\mathbf{s}]//@\text{main} = id \rightsquigarrow \mathfrak{C}_t[\mathsf{t}]//@\text{main} = id \Rightarrow \\ (\mathfrak{C}_s[\mathbf{s}]//@\text{main} = id \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathsf{t}]//@\text{main} = id \Downarrow)\end{array} \qquad (\textsc{Comp-}\Updownarrow)$$

then we can prove $\mathbf{s} \simeq_{\text{ctx}} \mathbf{s}'$. The reason is that we know from $\mathsf{t} \simeq_{\text{ctx}} \mathsf{t}'$ that $\mathfrak{C}_t[\mathsf{t}]//@\text{main} = id$ and $\mathfrak{C}_t[\mathsf{t}']//@\text{main} = id$ equi-terminate, and hence that:

$$\begin{array}{l}\mathfrak{C}_s[\mathbf{s}]//@\text{main} = id \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathsf{t}]//@\text{main} = id \Downarrow \Leftrightarrow \\ \mathfrak{C}_t[\mathsf{t}']//@\text{main} = id \Downarrow \Leftrightarrow \mathfrak{C}_s[\mathbf{s}']//@\text{main} = id \Downarrow.\end{array}$$

We would like to repeat the above process for the **Security** direction, i.e. prove $\mathsf{t} \simeq_{\text{ctx}} \mathsf{t}'$ through some form of equi-termination between source and target code. To prove $\mathsf{t} \simeq_{\text{ctx}} \mathsf{t}'$, we need to prove (by definition) that

$$\mathfrak{C}_t[\mathsf{t}]//@\text{main} = id \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathsf{t}']//@\text{main} = id \Downarrow$$

given *any* target context $(\mathfrak{C}_t, id)$ and source component proofs $\vdash \mathbf{s}$ and $\vdash \mathbf{s}'$. There is, however, one problem: since we start from a target context $(\mathfrak{C}_t, id)$ rather than a source context $(\mathfrak{C}_s, id)$, we cannot use our compiler to construct an equi-terminating source program for us. Simply inverting the compilation function is impossible, since it is not a bijection; the compiler's range is a strict subset of the target language. Hence, we need a new transformation, this time from target to source, to create equi-terminating source code, starting from any target-context $(\mathfrak{C}_t, id)$ and a source component proof $\vdash \mathbf{s}$. This target-to-source code transformation is called the *back-translation*, denoted $\vdash \mathbf{s}, (\mathfrak{C}_t, id) \rightsquigarrow_b \vdash \mathfrak{C}_s[\mathbf{s}]//@\text{main} = id$, and is a standard tool in full abstraction proofs. The proof $\vdash \mathbf{s}$ is necessary because the back-translated context $(\mathfrak{C}_s, id)$ needs to result in a sound program proof $\vdash \mathfrak{C}_s[\mathbf{s}]//@\text{main} = id$ (since $\mathbf{s} \simeq_{\text{ctx}} \mathbf{s}'$ requires verified code). To back-translate individual target statements $tstm$, no such proof is required and we hence simply write $tstm \rightsquigarrow_b \vdash sstm$.

Having introduced a back-translation, we can repeat the process used to prove **Correctness**. If we can prove that target code and its back-translation equi-terminate, i.e. if it holds for any contexts $(\mathfrak{C}_s, id)$ and $(\mathfrak{C}_t, id)$ and any components $\mathbf{s}$ and t (with $\vdash \mathbf{s} \rightsquigarrow \mathrm{t}$) that:

$$\begin{array}{l} \vdash \mathbf{s}, (\mathfrak{C}_t, id) \rightsquigarrow_b \vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id \;\Rightarrow \\ (\mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathrm{t}]//@\mathrm{main} = id \Downarrow) \end{array} \qquad (\textbf{BT-}\Updownarrow)$$

Then $\mathrm{t} \simeq_{\mathrm{ctx}} \mathrm{t}'$. The reasoning is analogous to the one for **Correctness** above.

For the sake of brevity, we introduce short notations for the compilation of general source code $\mathfrak{s}$ and the back-translation of target contexts $\mathfrak{C}_t$:

- The target code $[\![\vdash \mathfrak{s}]\!]$ denotes the result t of the compilation $\vdash \mathfrak{s} \rightsquigarrow \mathrm{t}$.
- The source context (without proof) $\langle\!\langle \vdash \mathbf{s}, (\mathfrak{C}_t, id) \rangle\!\rangle$ denotes the context $\mathfrak{C}_s$ in the back-translation $\vdash \mathbf{s}, (\mathfrak{C}_t, id) \rightsquigarrow_b \vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id$. To avoid notational clutter, we usually simply write $\langle\!\langle \mathfrak{C}_t \rangle\!\rangle$ when $\vdash \mathbf{s}$ is clear from context.

5.5.4 Proof Decomposition: Relational View

This subsection provides a more detailed account of the proofs of **Correctness** and **Security** by further decomposing their proof obligations **Comp-**$\Updownarrow$ and **BT-**$\Updownarrow$ from the previous subsection. The proof schemata in Figures 5.17 and 5.18 (inspired by the schemata of Devriese et al. [45]) illustrate this decomposition graphically. The equi-termination in both **Comp-**$\Updownarrow$ and **BT-**$\Updownarrow$ is proved by using two auxiliary relations R and S, for the correctness and security directions of full abstraction, respectively. Both R and S are binary relations that relate source language states to target language states during execution. These states are of the same form as the program states in the operational semantics; $\langle \overline{s}, h \rangle \mid \overline{c}$. Again, $\overline{c}$ is either a sequence of partially executed function bodies, or an entire program.

These relations internally make use of *simulation relations* [see e.g. 34], to capture the lock-step execution of source and target code. However, R and S are not technically simulation relations themselves (see Section 5.7). In this section, we consider both relations to be black boxes.

One important caveat should already be made regarding the source language states: since our compilation is separation-logic-driven, the target language states mirror the states of the separation logic *proof* in the source language, and not just the states of the executing source *code* itself. For example, at any given point during execution of code and its compilation, the linear capabilities present in the heap and stack in the target language will correspond to the separation logic resources present in the proof of the current source code. An analogous argument holds for the back-translation; the linear capabilities in the target language are reflected as separation logic resources in the source language at any point during execution, and hence require the inclusion of a proof into the source-language states.

As an example of why raw, unverified source code does not suffice to define R (or S), consider the following single-statement verified source program $\vdash \mathbf{s}$:

$$\{n : a \mapsto_{\text{int}} [2]\}_{\cdot[a:a]} \quad \mathrm{a}[0] = 3 \; \{n : a \mapsto_{\text{int}} [3]\}_{\cdot[a:a]}$$

and its compilation $[\![\vdash \mathbf{s}]\!] \equiv \mathrm{n}[0] = 3$. In the correctness setting, i.e. when relating $\vdash \mathbf{s}$ and $[\![\vdash \mathbf{s}]\!]$ through R, we need to relate the contents of the linear capability n in the target to the separation logic resource n in the source. If we solely used the raw source code without proof as the source state in R, i.e. the program $\mathrm{a}[0] = 3$, then it would be impossible to know what part of the source heap the target linear capability n corresponds to, because we erased the connection between n and a by erasing the verification proof. Additionally, if R would not constrain the contents of both to correspond and be equal to the single element 2 (in the target-level stack and the separation logic proof, respectively), simulation would get stuck if e.g. a conditional statement were encountered that checked whether $a[0] == 2$ in the source language (and hence whether $n[0] == 2$ in the target). Of course, in addition to enforcing this correspondence between source proof and target-level capabilities, R and S will also need to relate the concrete stack and heap in the source language to the current logical state in the precondition. More details about this can be found in the technical report.

In order to relate source and target states as they execute, we need a notion of separation logic proof that evolves along with the executing source code. We obtain this by *lifting* the source-level operational semantics to the verified source code. This new *lifted operational semantics* is detailed in the technical report. The definition relies on the property of *proof preservation* (the analogue to type preservation in type systems, see e.g. [128]). The property states the following: if we have a transition $\langle \overline{\mathbf{s}}, \mathbf{h} \rangle \mid \overline{\mathbf{c}} \hookrightarrow \langle \overline{\mathbf{s}'}, \mathbf{h}' \rangle \mid \overline{\mathbf{c}'}$ in the non-lifted operational semantics, and proofs $\overline{\vdash \mathbf{c}}$, then the resulting program $\overline{\mathbf{c}'}$ is also provable, i.e. we can construct proofs $\overline{\vdash \mathbf{c}'}$. Using this lemma, the lifted operational semantics essentially lets verified programs $\vdash \mathbf{c}$ step to $\vdash \mathbf{c}'$.

To better understand the lifted semantics, let us consider the execution of $\vdash \mathbf{s}$ and $[\![\vdash \mathbf{s}]\!]$ in the above example. Assuming appropriate stacks and heaps, both $\mathbf{s}$ (the source program without proof) and $[\![\vdash \mathbf{s}]\!]$ evaluate to a single skip statement in one step (cfr. the ArrayMut rule in Figure 5.4). Proof preservation updates the proof of $\vdash \mathbf{s}$ to a proof of the resulting skip statement:

$$\{n : a \mapsto_{\text{int}} [3]\}_{\cdot[a:a]} \quad \mathrm{skip} \; \{n : a \mapsto_{\text{int}} [3]\}_{\cdot[a:a]}$$

Note how we have "executed" the precondition of $\vdash \mathbf{s}$ to match the now executed array mutation.

Now that we understand how R and S relate source states of the form $\langle \overline{\mathbf{s}}, \mathbf{h} \rangle \mid \overline{\vdash \mathbf{c}}$ (where $\overline{\vdash \mathbf{c}}$ is either a monolithic, verified source program or a sequence of verified, partially executed function bodies) with target states of the form $\langle \overline{\mathsf{s}}, \mathsf{h} \rangle \mid \overline{\mathsf{c}}$, let us take a close look at Figures 5.17 and 5.18. Note how their visual similarity illustrates the similarities between the two proof directions. For compactness' sake, the main function specification $//@main = id$ has been left out of the source and target program descriptions in both schemata, for example abbreviating $\vdash \mathfrak{C}_{\mathfrak{s}}[\mathbf{s}]//\mathbf{@main} = \mathbf{id}$ to $\vdash \mathfrak{C}_{\mathfrak{s}}[\mathbf{s}]$ and $\mathfrak{C}_{\mathsf{t}}[\mathsf{t}]//\mathsf{@main} = \mathsf{id}$ to $\mathfrak{C}_{\mathsf{t}}[\mathsf{t}]$. The proof steps are denoted by arrows $\Rightarrow$, where a ✓ denotes a given and **?** a proof obligation. The contextual equivalence we need to prove has been boxed, and is situated across from the given contextual equivalence. The proof in both proof schemata starts at the left side of the dashed $\overset{?}{\Rightarrow}$ and traces the entire circle before arriving at its right side. For both correctness and security, all proof steps $\Rightarrow$ are explained by either the definition of contextual equivalence $\simeq_{\mathbf{ctx}}$, or one of a set of three auxiliary lemmas. These three lemmas are similar between correctness and security and numbered **(1)**, **(2)** and **(3)** in both. Notice how the number-annotated proof steps, considered in isolation, indeed constitute a decomposition of the proof obligations **Comp-⇕** and **BT-⇕** in the respective figures. For example, starting from the top left corner in Figure 5.17, consecutively applying **(1)**, **(2)** and **(3)** and ending up in the bottom left corner, corresponds to the left-to-right implication in **Comp-⇕**. Starting from the bottom right and moving upwards results in the right-to-left implication direction. We now discuss the role of both contextual equivalence and the three lemmas in order.

The arrows annotated with $\simeq_{\mathbf{ctx}}$ and $\simeq_{\mathsf{ctx}}$ denote an application of the definition of source- and target-level contextual equivalence, respectively. The universal quantification over (well-formed) contexts $(\mathfrak{C}_{\mathbf{s}}, \mathit{id})$ and $(\mathfrak{C}_{\mathsf{t}}, \mathit{id})$ is left implicit in the unfolding of the definition. For correctness (and similarly for security), the implication in the proof obligation $\mathfrak{C}_{\mathfrak{s}}[\mathbf{s}] \Downarrow \overset{?}{\Rightarrow} \mathfrak{C}_{\mathfrak{s}}[\mathbf{s}'] \Downarrow$ is sufficient to prove contextual equivalence in the source language. The other direction follows by symmetry.

The **Coherence** lemma **(1)** states that source programs in the regular operational semantics (i.e. without their proofs) and the same source programs in the lifted operational semantics (i.e. including their proofs) equi-terminate. This lemma allows adding proofs to source programs and conversely stripping them away, all the while preserving termination. This conversion is necessary since $\simeq_{\mathbf{ctx}}$ is defined using the regular operational semantics, whereas the relations S and R make use of the lifted semantics.

The **Compatibility** **(2)** and **Adequacy** **(3)** lemmas are used in combination to prove equi-termination between a source program and its compilation in the correctness case, and between a target program and its back-translation in the security case. **Compatibility** proves that any source program and its compilation are related by R under the empty stack and heap for correctness, and that any target program

$$\begin{tikzcd}
 & \boxed{\mathrm{s} \simeq^{?}_{\text{ctx}} \mathrm{s}'} \arrow[d, Leftrightarrow, "\simeq_{\text{ctx}}"] & \\
\mathfrak{C}_{\mathrm{s}}[\mathrm{s}] \Downarrow \arrow[rr, Rightarrow, dashed, "?"] \arrow[d, Rightarrow, "(1)"] & {} & \mathfrak{C}_{\mathrm{s}}[\mathrm{s}'] \Downarrow \\
\vdash \mathfrak{C}_{\mathrm{s}}[\mathrm{s}] \Downarrow \arrow[d, Rightarrow, "(2)+(3)"] & & \vdash \mathfrak{C}_{\mathrm{s}}[\mathrm{s}'] \Downarrow \arrow[u, Rightarrow, "(1)"'] \\
\llbracket \vdash \mathfrak{C}_{\mathrm{s}} \rrbracket \left[\llbracket \vdash \mathrm{s} \rrbracket \right] \Downarrow \arrow[rr, Rightarrow, "\checkmark"] & {} \arrow[d, Leftrightarrow, "\simeq_{\text{ctx}}"] & \llbracket \vdash \mathfrak{C}_{\mathrm{s}} \rrbracket \left[\llbracket \vdash \mathrm{s}' \rrbracket \right] \Downarrow \arrow[u, Rightarrow, "(2)+(3)"'] \\
 & \llbracket \mathrm{s} \rrbracket \simeq^{\checkmark}_{\text{ctx}} \llbracket \mathrm{s}' \rrbracket &
\end{tikzcd}$$

(1) $\vdash \mathit{sprog} \Downarrow \Leftrightarrow \mathit{sprog} \Downarrow$

(Coherence)

(2) $\vdash \mathfrak{C}_{\mathrm{s}}[\mathrm{s}]//\mathrm{@main} = \mathit{id} \rightsquigarrow \quad \Rightarrow$

$\llbracket \vdash \mathfrak{C}_{\mathrm{s}} \rrbracket \left[\llbracket \vdash \mathrm{s}' \rrbracket \right] //\mathrm{@main} = \mathit{id}$

$(\langle \cdot, \cdot \rangle \mid \vdash \mathfrak{C}_{\mathrm{s}}[\mathrm{s}]//\mathrm{@main} = \mathit{id}\,)\ R$

$(\langle \cdot, \cdot \rangle \mid \llbracket \vdash \mathfrak{C}_{\mathrm{s}} \rrbracket \left[\llbracket \vdash \mathrm{s} \rrbracket \right] //\mathrm{@main} = \mathit{id}\,)$

(Compatibility)

(3) $(\langle \cdot, \cdot \rangle \mid \vdash \mathit{sprog})\ R\ (\langle \cdot, \cdot \rangle \mid \mathit{tprog}) \quad \Rightarrow$

$\vdash \mathit{sprog} \Downarrow \Leftrightarrow \mathit{tprog} \Downarrow$

(Adequacy)

Figure 5.17: **Correctness** proof outline.

$$\begin{tikzcd}
 & \mathrm{s} \simeq^{\checkmark}_{\text{ctx}} \mathrm{s}' \arrow[d, Leftrightarrow, "\simeq_{\text{ctx}}"] & \\
\langle\!\langle \mathfrak{C}_{\mathrm{t}} \rangle\!\rangle[\mathrm{s}] \Downarrow \arrow[rr, Rightarrow, "\checkmark"] & {} & \langle\!\langle \mathfrak{C}_{\mathrm{t}} \rangle\!\rangle[\mathrm{s}'] \Downarrow \arrow[d, Rightarrow, "(1)"] \\
\vdash \langle\!\langle \mathfrak{C}_{\mathrm{t}} \rangle\!\rangle[\mathrm{s}] \Downarrow \arrow[u, Rightarrow, "(1)"'] & & \vdash \langle\!\langle \mathfrak{C}_{\mathrm{t}} \rangle\!\rangle[\mathrm{s}'] \Downarrow \arrow[d, Rightarrow, "(2)+(3)"] \\
\mathfrak{C}_{\mathrm{t}} \left[\llbracket \vdash \mathrm{s} \rrbracket \right] \Downarrow \arrow[u, Rightarrow, "(2)+(3)"'] \arrow[rr, Rightarrow, dashed, "?"] & {} \arrow[d, Leftrightarrow, "\simeq_{\text{ctx}}"] & \mathfrak{C}_{\mathrm{t}} \left[\llbracket \vdash \mathrm{s}' \rrbracket \right] \Downarrow \\
 & \boxed{\llbracket \mathrm{s} \rrbracket \simeq^{?}_{\text{ctx}} \llbracket \mathrm{s}' \rrbracket} &
\end{tikzcd}$$

(1) $\vdash \mathit{sprog} \Downarrow \Leftrightarrow \mathit{sprog} \Downarrow$

(Coherence)

(2) $\vdash \mathrm{s}, (\mathfrak{C}_{\mathrm{t}}, \mathit{id}) \rightsquigarrow_{\mathrm{b}} \quad \Rightarrow$

$\vdash \langle\!\langle \mathfrak{C}_{\mathrm{t}} \rangle\!\rangle[\mathrm{s}]//\mathrm{@main} = \mathit{id}$

$(\langle \cdot, \cdot \rangle \mid \vdash \langle\!\langle \mathfrak{C}_{\mathrm{t}} \rangle\!\rangle[\mathrm{s}]//\mathrm{@main} = \mathit{id}\,)\ S$

$(\langle \cdot, \cdot \rangle \mid \mathfrak{C}_{\mathrm{t}} \left[\llbracket \vdash \mathrm{s} \rrbracket \right] //\mathrm{@main} = \mathit{id}\,)$

(Compatibility)

(3) $(\langle \cdot, \cdot \rangle \mid \vdash \mathit{sprog})\ S\ (\langle \cdot, \cdot \rangle \mid \mathit{tprog}) \quad \Rightarrow$

$\vdash \mathit{sprog} \Downarrow \Leftrightarrow \mathit{tprog} \Downarrow$

(Adequacy)

Figure 5.18: **Security** proof outline.

and its back-translation are related by S under the empty stack and heap for security. **Adequacy** finishes the combined equi-termination argument by stating that source and target programs related by S or R equi-terminate (from the empty stack and heap). The proof of **Adequacy** follows straightforwardly from the fact that S and R internally make use of simulation relations. This will be clarified further in Section 5.7.

5.6 Proving security: the back-translation

Similarly to how Section 5.4 introduced compilation using Figure 5.5, this section will introduce the back-translation by means of an example, that builds on top of the compilation example. The goal of this section is to illustrate how the back-translation of this example satisfies one specific instance of **BT-**$\Updownarrow$ from the previous section, in the process highlighting the key concepts behind the back-translation. Concretely, we will back-translate a target-level implementation of the context function *add1* from Figure 5.5 and provide intuitions for why the back-translation and the original implementation equi-terminate. The concrete implementation of *add1* that we will back-translate is given in the bottom-right of Figure 5.20.

Let us identify what concrete instance of **BT-**$\Updownarrow$ we have to prove here. The verification of f from Figure 5.5 acts as our verified component $\vdash \mathbf{s} = \vdash (\ f \ //@\text{import}\ add1\ )$. Note the slight abuse of notation, where we use the name of a function (f) for its entire implementation. Consequently, f_{comp} and $add1_{\text{comp}}$ together act as our compiled target component $\mathrm{t} = [\![\vdash \mathbf{s}]\!] = f_{\text{comp}}\ add1_{\text{comp}}\ //@\text{import}\ add1$. Our target context $(\mathfrak{C}_{\mathrm{t}}, id)$ then consists of the target function *add1*'s implementation in Figure 5.20 and an arbitrary choice for the main function *id*.

Since the main function in our formalization is not allowed to have arguments, needs to have void as its return type, and needs to be exported by some component, we cannot have f_{comp} nor *add1* as a main function. Instead, we could for example add an exported function *main* to $\vdash \mathbf{s}$, solely serving as a wrapper for f, with an implementation as shown in Figure 5.19.

```
void main()
//@pre true
//@post true {
   int* a; a = malloc(2 * sizeof(int));
   a[0] = 0; a[1] = 1;
   int res; res = f(a);
   return }
```

Figure 5.19: Example implementation of a main function wrapping f from Figure 5.5.

This function *main* is compiled by our compiler to a function $\mathit{main}_{\mathrm{comp}}$, and since it is an exported function, an incall stub (without guard statements, since the precondition of *main* is true) *main* is generated, which is the target main function and would in turn have to be back-translated as well. However, to not needlessly clutter our example, we simply assume the main function in both source and target languages to be called *main*, respectively *main*, and do not explicitly represent or back-translate this main function anywhere.

In other words, $(\mathfrak{C}_{\mathrm{t}}, \mathit{id}) = ((\mathit{add1}\ //\mathrm{@export}\ \mathit{add1}\ ),\ \mathit{main})$, where we will not explicitly write *main* in t. Notice how t and $(\mathfrak{C}_{\mathrm{t}}, \mathit{id})$ together form a sound target-program, as required by the definition of contextual equivalence $\mathsf{t} \simeq_{\mathrm{ctx}} \mathsf{t}'$ in Section 5.5.3. To prove this specific instance of (**BT-**$\Updownarrow$), we have to back-translate the example context $(\mathfrak{C}_{\mathrm{t}}, \mathit{id})$ to a context $(\mathfrak{C}_{\mathrm{s}}, \mathit{id})$ such that $\vdash \mathfrak{C}_{\mathrm{s}}[\mathbf{s}]//\mathrm{@main} = \mathit{id}$ is a valid separation logic proof, and $\mathfrak{C}_{\mathrm{s}}[\mathbf{s}]//\mathrm{@main} = \mathit{id}$ and $\mathfrak{C}_{\mathrm{t}}[\mathsf{t}]//\mathrm{@main} = \mathit{id}$ equi-terminate. As we will see, $\mathfrak{C}_{\mathrm{s}}$ will contain both a back-translation $\mathit{add1}_{\mathrm{bt}}$ of *add1* and a back-translation *add1* of the outcall stub $\mathit{add1}_{\mathrm{comp}}$, resulting in the source context

$$(\mathfrak{C}_{\mathrm{s}}, \mathit{id}) = (\langle\!\langle \mathfrak{C}_{\mathrm{t}} \rangle\!\rangle, \mathit{id}) = ((\mathit{add1}\ \mathit{add1}_{\mathrm{bt}}\ //\mathrm{@export}\ \mathit{add1}\ ),\ \mathit{main})$$

where, again, we will not explicitly write *main* in $\vdash \mathbf{s}$.

Notice how the implementation of the context function *add1* in Figure 5.20 upholds the source-level contract that the verified component $\vdash \mathbf{s}$ expects of *add1* in Figure 5.5: it reads the first element of resource *m*, increments and returns it, together with resource *m*, without altering the address of *m* or its contents. The context $(\mathfrak{C}_{\mathrm{t}}, \mathit{id})$ is *not* required to behave properly like this! It might also add 2 instead, change the contents of the resource *m*, etc., effectively ignoring the expectations of $\vdash \mathbf{s}$ and causing execution to get stuck at the guard statements of the outcall stub $\mathit{add1}_{\mathrm{comp}}$. This extra freedom of the target context to misbehave, and the requirement for guard statements to detect such misbehavior, is at the core of the full abstraction proof: the security proof direction states that we can reinterpret (i.e. back-translate) even possibly misbehaving contexts as equi-terminating, verified source contexts that are incapable of breaking the verification guarantees of $\vdash \mathbf{s}$. Section 5.6.3 will further illustrate this point, by briefly demonstrating the back-translations of a few misbehaving implementations of *add1*.

The remainder of this section introduces the back-translation incrementally, introducing key concepts gradually. First, Section 5.6.1 starts off with a naive backtranslation of *add1* that does not generalize to arbitrary code. After pointing out some problems in generalizing this back-translation, Section 5.6.2 introduces a version that works for any target code not using nested pointer types (such as int$**$) in the target language. As mentioned, Section 5.6.3 briefly investigates the back-translation of misbehaving contexts. Finally, Section 5.6.4 sketches what the most general back-translation looks like by investigating the back-translation of nested pointers.

5.6.1 Naive back-translation

This subsection constructs a naive version of the back-translation of *add1* from Figure 5.20, and discusses the results in the **Source** row of this same figure. The back-translation is *naive* because it assumes a statically-known size of 1 for all target capabilities (the *naive assumption*), and does not support back-translating nested pointer types. These assumptions are unproblematic for *add1*.

Outcall Stub | Context

Source (back-translation)

```
int add1(int* a)
//@pre m: a ↦_int [a_1]
//@post n: a ↦_int [a_1] ∗ result == a_1 + 1 {
   int* m; m = a;
   int* a^pre; int a_1^pre;
   a^pre = m; a_1^pre = m[0];
   int result; int* n;
   (result,n) = add1_bt(a,m);
   guard(n != null);
   int* a^post; int a_1^post;
   a^post = n; a_1^post = n[0];
   guard(result == a_1^post + 1);
   guard(a^post == a); guard(a_1^post == a_1^pre);
   return result }
```

```
univ_contr_{int*_0}(x) ≜ true
univ_contr_int(x) ≜ true
univ_contr_{int*}(x) ≜ (x != null) ?
        n : x ↦_int [l_1]
```

```
int add1_bt(int* a, int* m)
//@pre  univ_contr_{int*_0}(a) ∗ univ_contr_{int*}(
        m)
//@post univ_contr_int(result_1) ∗
      univ_contr_{int*}(result_2) {
   int b;
   guard(m != null);
   b = m[0];
   return (b + 1,m) }
```

Target

```
(int,int*) add1_comp(int*_0 a,int* m)
   int*_0 a^pre; int a_1^pre;
   a^pre = addr(m); a_1^pre = m[0];
   int result; int* n;
   (result,n) = add1(a,m);
   guard(n!=null); guard(length(n) == 1);
   int*_0 a^post; int a_1^post;
   a^post = addr(n); a_1^post = n[0];
   guard(result == a_1^post + 1);
   guard(a^post == a); guard(a_1^post == a_1^pre);
   return (result,n) }
```

```
(int,int*) add1(int*_0 a,int* m){
   int b;
   b = m[0];
   return (b + 1,m)}
```

Figure 5.20: Illustrative example: naive back-translation of a context that implements *add1*.

We first define the back-translation of types and expressions. Target types that can result from compilation of source types, i.e. int, length-0 capabilities $\tau_s *_0$ and tuples (τ_t^*), are simply back-translated inversely to how they are compiled. However, linear capabilities are reified resources and did not originally exist in the source language, so we have to come up with a way to represent them.

Target linear capabilities $l^{[a,b]}$ inherently contain both an address l and a length $b-a+1$, as discussed in Section 5.3.3. To extract these, the target language contains built-in addr and length functions. Pointers in the source language are of the form (l, i), are non-linear and do not have built-in length information and the source language does not have (or need) addr or length functions. Fortunately, the naive back-translation assumes all target pointers to have a statically-known length of 1, so there is no need to keep any length information in the source language. It is hence possible to simply back-translate linear capabilities of type $\tau*$ to source-level pointers $\tau'*$, where τ' is obtained by recursively back-translating τ. The pointer $\tau'*$ simultaneously represents the back-translated address, since $\tau'*$ is non-linear and allows for pointer arithmetic. Both the length and the address information of each capability are hence retained during back-translation. These back-translations of target types are formalized by the judgment $\tau \leadsto_{\text{InvCompileType}} \tau'$, dual to $\tau \leadsto_{\text{CompileType}} \tau'$ from Section 5.4.2, that recursively back-translates target types as follows:

$$\frac{}{\text{int} \leadsto_{\text{InvCompileType}} \text{int}}\ (\textsc{InvCompileInt}) \qquad \frac{\tau' \leadsto_{\text{InvCompileType}} \tau}{\tau'*_0 \leadsto_{\text{InvCompileType}} \tau*}\ (\textsc{InvCompileSrcPtr})$$

$$\frac{\tau_1 \leadsto_{\text{InvCompileType}} \tau'_1 \quad \dots \quad \tau_k \leadsto_{\text{InvCompileType}} \tau'_k}{(\tau_1, \dots, \tau_k) \leadsto_{\text{InvCompileType}} (\tau'_1, \dots, \tau'_k)}\ (\textsc{InvCompileTuple}) \qquad \frac{\tau \leadsto_{\text{InvCompileType}} \tau'}{\tau* \leadsto_{\text{InvCompileType}} \tau'*}\ (\textsc{InvertCapability})$$

Section 5.6.2 will scrap the naive assumption and will therefore have to introduce a more involved back-translation for pointer types, retaining length information.

Unfortunately, the back-translated linear capabilities will not automatically behave linearly. Therefore, our back-translation needs to simulate their linear behavior. Extra statements have to be added during the back-translation to imitate the target-language erasure of capabilities. For example, when back-translating the assignment $x = y$ with y of type int$*$, an erasure assignment $y =$ null will be added in the source. Similarly, once we support back-translating nested pointers in Section 5.6.4, back-translating the assignment $x = n[2]$ with n of target type int$**$ produces an erasure assignment $n[2] =$ null in the source. Additionally, since the target language gets stuck when the same target capability is used twice in one statement, the back-translation then has to artificially get stuck as well. For example, when back-translating the assignment $x = (y, y)$, with y as before, the source language needs to add a statement guard(false) to ensure equi-termination.

The back-translation of expressions *texp*, denoted $\textit{texp}_\text{b}$, is now easy to define, since target and source expressions only differ in the addr and length functions mentioned before. The back-translation hence maps addr(*texp*) to $\textit{texp}_\text{b}$ (the pointer doubles as address, since it is copyable) and length(*texp*) to ($\textit{texp}_\text{b}$!= null) (all non-null capabilities are assumed to have length one, whereas null has length 0). All other cases proceed structurally.

With these prerequisites out of the way, we can examine the back-translation $\mathfrak{C}_s$ of the target context $\mathfrak{C}_t$, shown in the top row of Figure 5.20. As mentioned, this back-translation consists of two separate parts; the expected back-translation *add1*$_{\text{bt}}$ of the target function *add1* on the right, but also a back-translation *add1* of the previously generated stub *add1*$_{\text{comp}}$ on the left. Conceptually, the universal contract of *add1*$_{\text{bt}}$ captures how the use of linear capabilities in the target language restricts the behavior of the context function *add1*, whereas guards in *add1* will enforce further functional conditions that f expects of *add1*. The function *add1* is derived from *add1*$_{\text{bt}}$, using a separate back-translation for stubs that we will motivate below. First, let us investigate how regular target functions such as *add1* are back-translated.

$$\text{int b;} \quad \rightsquigarrow_b$$

$$\text{b = m[0];} \quad \rightsquigarrow_b$$

$$\text{return (b + 1,m)} \quad \rightsquigarrow_b$$

$$\{\text{univ\_contr}_{\text{int}*_0}(a) * \text{univ\_contr}_{\text{int}*}(m)\}.[a{:}a][m{:}m]$$
$$\text{int b;}$$
$$\{\sim\}.[a{:}a][m{:}m][b{:}0]$$
$$\{\text{univ\_contr}_{\text{int}*_0}(a) * \text{univ\_contr}_{\text{int}*}(m) * \text{univ\_contr}_{\text{int}}(b)\}.[a{:}a][m{:}m][b{:}b]$$
$$\{\text{true} * (m \mathrel{!=} \text{null})\ ?\ m_{\text{chunk}} : m \mapsto_{\text{int}} [l_1] * \text{true}\}.[a{:}a][m{:}m][b{:}b]$$
$$\text{guard(m != null);}$$
$$\{m_{\text{chunk}} : m \mapsto_{\text{int}} [l_1] * 0 < 1\}.[a{:}a][m{:}m][b{:}b]$$
$$\text{b = m[0];}$$
$$\{\sim\}.[a{:}a][m{:}m][b{:}l_1]$$
$$\{\text{univ\_contr}_{\text{int}*_0}(a) * \text{univ\_contr}_{\text{int}*}(m) * \text{univ\_contr}_{\text{int}}(b)\}.[a{:}a][m{:}m][b{:}b]$$
$$\text{return (b + 1,m)}$$
$$\{\text{univ\_contr}_{\text{int}}(\mathit{result}_1) * \text{univ\_contr}_{\text{int}*}(\mathit{result}_2)\}.$$

Figure 5.21: Separation logic proof of the body of *add1*$_{\text{bt}}$ from the naive back-translation example.

Essentially, our goal is to understand Figure 5.21: the back-translation equivalent of the proof of f_{comp} in Figure 5.6. The notation $\sim$ is used to denote an unaltered symbolic heap. A first question is what the separation-logic contract of the back-translated function *add1*$_{\text{bt}}$ should be. The desired contract is the one $\vdash \mathbf{s}$ expects for *add1* in Figure 5.5, so that the resulting source program $\mathfrak{C}_s[\mathbf{s}]//@\text{main} = \mathit{id}$ has a sound verification. However, since the target context *add1* can freely misbehave as we saw earlier, proving this contract will in general be impossible. Additionally, if the context contains functions that are not imported by $\vdash \mathbf{s}$, there are no restrictions on their contract whatsoever.

The solution is to employ the most general admissible contract for the backtranslation and adapt it to the expected contract separately (see below). This contract will express the permissions associated with target-language capabilities as separation logic resources in the source. Combining the resources represented by all arguments of a back-translated function gives us its precondition and the resources represented by its result are the postcondition. We call this type of contract a *universal contract* and define it in the next paragraph. Lines 2-3 of *add1*$_{\mathrm{bt}}$ in Figure 5.20 show the universal contract for $add1_{\mathrm{bt}}$.

Universal contracts $\mathrm{univ\_contr}_{\tau_{\mathrm{t}}}$ are predicates on logical expressions *exp*. They are indexed by the target type τ_{t} of id_{p}, since this type determines the target-language permissions associated with the variable. Universal contracts are separation logic assertions and hence cannot take program variables like id_{p} as a direct argument. The universal contract for a program variable id_{p} of type τ_{t} is obtained by applying $\mathrm{univ\_contr}_{\tau_{\mathrm{t}}}$ to the variable's logical interpretation $\gamma(id_{\mathrm{p}})$.

We now present a simplified definition for universal contracts, which we will expand upon in Sections 5.6.2 and 5.6.4:

Definition 5.1 ($\mathrm{univ\_contr}_{\tau_{\mathrm{t}}}(exp)$)**.**

$$\begin{aligned}
\mathrm{univ\_contr}_{\tau_{\mathrm{t}}}(exp) &\triangleq \mathrm{true}\ \mathit{if}\ \tau_{\mathrm{t}} = \mathrm{int}\ \mathit{or}\ \tau_{\mathrm{t}} = \tau_{\mathrm{s}}{*_0} \\
\mathrm{univ\_contr}_{(\tau_1,\dots,\tau_k)}(exp) &\triangleq \mathrm{univ\_contr}_{\tau_1}(exp.1) * \dots * \mathrm{univ\_contr}_{\tau_k}(exp.k) \\
\mathrm{univ\_contr}_{\tau_{\mathrm{t}}*}(exp) &\triangleq (exp \mathrel{!=} \mathrm{null})\ ?\ \exists l_1.\, n : exp \mapsto_{\tau_{\mathrm{s}}} [l_1] \\
&\qquad \text{given that } \tau_{\mathrm{t}} \rightsquigarrow_{\mathrm{InvCompileType}} \tau_{\mathrm{s}} \text{ and } n \text{ fresh}
\end{aligned}$$

The case for target-capabilities $\tau_{\mathrm{t}}*$ is the only non-trivial one. It states that a linear capability is either the null-pointer, or that it has length one (per the naive assumption made before) and allows access to its unspecified contents l_1.

Universal contracts are now used to back-translate each target-level statement to a block of verified source code. Both the separation-logic pre- and postcondition of such blocks consist of the separating conjunction of universal contracts for all declared target-level program variables. The universal contract will hence monotonically increase throughout the proof; if a *tstm* declares a set of variables V_{tstm}, if variables $\mathrm{V}_{\mathrm{pre}}$ were previously declared and if $tstm \rightsquigarrow_{\mathrm{b}} \vdash sstm$ holds, then *sstm* has as contract (omitting type subscripts)

$$\{\mathrm{univ\_contr}(\gamma_{\mathrm{pre}}(\mathrm{V}_{\mathrm{pre}}))\}_{\gamma_{\mathrm{pre}}}\ sstm\ \{\mathrm{univ\_contr}(\gamma_{\mathrm{post}}(\mathrm{V}_{\mathrm{pre}} \cup \mathrm{V}_{tstm}))\}_{\gamma_{\mathrm{post}}}$$

One of the main efforts in defining the back-translation is proving that the above Hoare triple indeed holds for all single-statement back-translation rules. Figure 5.21 demonstrates this block-level proof; the three statements (including return) of *add1* are back-translated to the three annotated blocks on lines 1-4, 4-10 and 10-12 in the proof.

These separate proof blocks offer the advantage that the back-translation can be proven modularly, on a block-per-block basis, since we already know that the last block's postcondition and the next blocks's precondition will correspond. At the start of function execution, only the arguments id_{arg} have been declared; the precondition of a back-translated function is hence univ_contr($\overline{id_{\text{arg}}}$), as demonstrated on line 1 of Figure 5.21. The postcondition is an exception, since the caller only cares about the privileged $\overline{result}$ variables and resources over them. The universal postcondition is hence univ_contr($\overline{result}$), as demonstrated on line 12 of Figure 5.21. This postcondition is achieved through ConSeq once the function's body (without the return statement) is proven.

Since each back-translated block of code has to start and end in a universal contract, each block consists of three separate phases. These three phases are summarized in Figure 5.22. To illustrate this figure, we investigate the back-translation of the array lookup on line 2 in *add1*, i.e. the middle block on lines 4-10 in Figure 5.21.

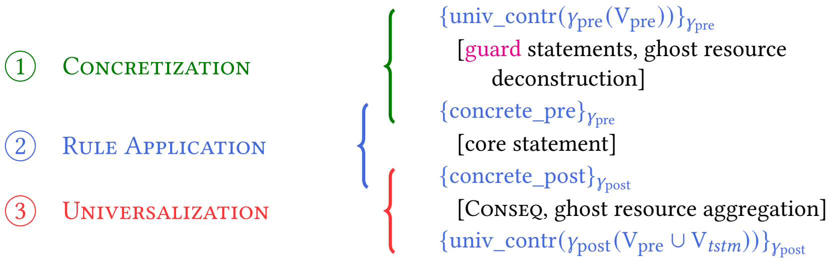

① Concretization
② Rule Application
③ Universalization
$\{\text{univ\_contr}(\gamma_{\text{pre}}(\text{V}_{\text{pre}}))\}_{\gamma_{\text{pre}}}$
[guard statements, ghost resource deconstruction]
$\{\text{concrete\_pre}\}_{\gamma_{\text{pre}}}$
[core statement]
$\{\text{concrete\_post}\}_{\gamma_{\text{post}}}$
[ConSeq, ghost resource aggregation]
$\{\text{univ\_contr}(\gamma_{\text{post}}(\text{V}_{\text{pre}} \cup \text{V}_{tstm}))\}_{\gamma_{\text{post}}}$

Figure 5.22: Illustration of the general three-phase structure of back-translation rules.

First, to match the precondition of the ArrayLkup separation logic rule, we need a resource different from the null pointer, and we need to know that our index (0 here) is within the bounds of our array. This last fact follows automatically from the naive assumption. Since the universal contract does not provide us guarantees about the pointer m not being null, lines 4-7 add this condition through a guard and derive the necessary preconditions for ArrayLkup using ConSeq. Failing this inserted guard statement would make the back-translated program get stuck. This is desired behavior, preserving equi-termination, since the target operational semantics would also get stuck in this faulty case. To summarize, this first phase, called Concretization in Figure 5.22, starts from the universal contract and transforms it into the precondition concrete_pre of the separation logic rule we actually want to apply. As we will discuss in Section 5.6.2, this phase will also include transformations on ghost resources in the general case.

Second, the core rule ArrayLkup is applied on lines 7-9, leaving us with the concrete postcondition concrete_post. This second phase is called Rule Application in

Figure 5.22. Finally, on lines 9-10, Conseq is applied to forget the information added through the guard statement and the array lookup, making the postcondition universal again. This third phase is called Universalization in Figure 5.22 and transforms concrete_post back into a universal contract. As with Concretization, transformations on ghost resources can be required in more complex cases. In general, it is non-trivial to see that the Conseq rule will always suffice to prove Universalization. In the technical report, this fact is proven for each back-translation rule individually, by making use of Theorems 5 and 6.

The back-translations of other statements follows the same structure of Figure 5.22. For some simple statements, the first phase might be empty. Such is the case for e.g. the return block on lines 10-12 of Figure 5.21.

The aforementioned simulation of linearity in the source language has so far been swept under the rug in this discussion. Concretely, the Universalization phase inserts erasure statements for linear capabilities, whereas the Concretization phase makes sure that guard(false) statements are inserted when linear capabilities would otherwise be duplicated.

We can now back-translate regular target functions, but the resulting universal contracts do not match the concrete contracts that the source context $\vdash \mathbf{s}$ expects, e.g. the contract for *add1* in Figure 5.5 does not match the universal contract of $add1_{\mathrm{bt}}$ in Figure 5.20. When f performs an outcall to *add1*, the gap between the preconditions of *add1* and $add1_{\mathrm{bt}}$ needs to be bridged: the back-translated argument m needs to be constructed and the universal contracts univ_contr of m and a need to be satisfied, starting from the concrete precondition of *add1*. Conversely, when *add1* returns control to f afterwards, we need to find a way to transform the universal postcondition of $add1_{\mathrm{bt}}$ into the concrete postcondition of *add1* that f expects.

When we had a similar mismatch between guarantees and expectations on trust boundaries during compilation, we introduced stubs to enforce contracts at the target level. Enforcing a contract is exactly what we need to do here, but now at the source level. We therefore reexamine the outcall stub $add1_{\mathrm{comp}}$ (semi-transparently repeated in Figure 5.20) that $\vdash \mathbf{s}$ generates for *add1*, based on the contract it expects *add1* to uphold. This outcall stub contains reified checks (i.e. the guard statements on line 6 and lines 9-10) enforcing all conditions present in the postcondition of *add1*. In other words, if we can somehow back-translate the outcall stub $add1_{\mathrm{comp}}$ and insert it between f and $add1_{\mathrm{bt}}$, the back-translated guard statements should correctly add the missing concrete conditions to $add1_{\mathrm{bt}}$'s universal postcondition. Additionally, but less crucially, this back-translated outcall stub will have to connect the arguments and preconditions of the two functions. Given that the back-translated outcall stub needs to convert $add1_{\mathrm{bt}}$'s universal postcondition into f's concrete one and vice versa for preconditions, the back-translation cannot make use of universal contract blocks, as it did for $add1_{\mathrm{bt}}$. Stubs hence make use of an alternative, second, back-translation, which *does* retain concrete information across back-translated statements.

The back-translated outcall stub, called *add1* to match the name used by f, is shown in the top left corner of Figure 5.20. It mostly consists of straightforward back-translations of the individual statements of $add1_{\text{comp}}$, with a few caveats, mainly caused by the fact that f is a regular verified source function, whereas $add1_{\text{comp}}$ is a back-translated function that mimics a target function. The non-obvious aspects are the following:

- Notice how the guard statement checking the length of n on line 6 of $add1_{\text{comp}}$ is absent on line 9 of *add1*. This discrepancy is caused by the naive assumption, which allows us to know beforehand that n has length 1. This distinction between the functions is a clear hint that our current back-translation schema is not sufficiently general. It will naturally disappear in the next section, when we lift the naive assumption.
- A proof of the contract of *add1* needs to be constructed, to prove soundness of the back-translation. Contrary to the regular back-translation, no universal contract blocks are created for each back-translated statement, since we *want* to make a non-modular proof. This implies that only the RULE APPLICATION phase for each back-translated block is kept. An example symbolic execution of the interesting part of *add1*, describing how the back-translated guard statements transform a universal contract into a concrete postcondition, can be found in Figure 5.23.
- The function f is unaware of the back-translated reified resource m, whereas $add1_{\text{bt}}$ expects m as an argument. This value hence has to be declared and assigned on line 4 of *add1*, using the information about the logical resource m present in the precondition.
- The back-translated guard statements in *add1* guarantee equi-termination between source and target language when $add1_{\text{bt}}$ misbehaves, since they mirror the guards in the target-level outcall stub $add1_{\text{comp}}$.

Because the example's back-translation (including source-level stubs) forms a sound separation-logic proof and closely mimics the target language, this concrete instantiation of equation (**BT-⇕**) will indeed hold, as we set out to illustrate at the start of this section.

5.6.2 The regular back-translation

In this section, we generalize the back-translation of *add1*, introduced in Section 5.6.1, by lifting the naive assumption made there. Concretely, we discard the assumption that each target-level linear capability should have size one, and extend the back-translation to allow for linear capabilities of arbitrary size. In fact, the size of linear

$(\ldots)$

(result,n) = add1$_{\mathrm{bt}}$(a,m);

$\{\mathrm{univ\_contr}_{\mathrm{int}*_0}(\mathit{result})$
$* \ \mathrm{univ\_contr}_{\mathrm{int}*}(n, 1)$
$\}_{\bullet[a:a][a_1^{\mathrm{pre}}:a_1][\mathit{result}:\mathit{result}][n:n]}$

guard(n != null);

$\{\mathrm{true} * n_{\mathrm{chunk}} : n \mapsto_{\mathrm{int}} [l_1]\}_{[\sim]}$

int* $\mathrm{a}^{\mathrm{post}}$; int $\mathrm{a}_1^{\mathrm{post}}$;

$\{n_{\mathrm{chunk}} : n \mapsto_{\mathrm{int}} [l_1]\}_{[\sim][a^{\mathrm{post}}:0][a_1^{\mathrm{post}}:0]}$

$\mathrm{a}^{\mathrm{post}}$ = n; $\mathrm{a}_1^{\mathrm{post}}$ = n[0];

$\{n_{\mathrm{chunk}} : n \mapsto_{\mathrm{int}} [l_1]\}_{\bullet[\sim][a^{\mathrm{post}}:n][a_1^{\mathrm{post}}:l_1]}$

guard(result == $\mathrm{a}_1^{\mathrm{post}}$ + 1);

guard($\mathrm{a}^{\mathrm{post}}$ == a); guard($\mathrm{a}_1^{\mathrm{post}}$ == $\mathrm{a}_1^{\mathrm{pre}}$);

$\{n_{\mathrm{chunk}} : n \mapsto_{\mathrm{int}} [l_1] * \mathit{result} == l_1 + 1$
$* \ n == a * l_1 == a_1\}_{[\sim]}$

$\{n_{\mathrm{chunk}} : a \mapsto_{\mathrm{int}} [a_1] *$
$\mathit{result} == a_1 + 1\}_{[\mathit{result}:\mathit{result}]}$

return result

$\{n : a \mapsto_{\mathrm{int}} [a_1] *$
$\mathit{result} == a_1 + 1\}_{[\mathit{result}:\mathit{result}]}$

Figure 5.23: Excerpt from the separation logic proof of the body of *add1* from the naive back-translation example.

capabilities may not be statically determined, and we have to take this into account in the back-translation. For example, *add1* might be invoked by other functions than f, which may hand it a capability m with an arbitrary size. The current universal contract for m in Figure 5.20 clearly does not allow for this case.

In addition to this first generalization, we will reformulate the definition of universal contracts to use range instead of array resources. This reformulation is required for the back-translation of nested pointers, the discussion of which we defer to Section 5.6.4.

The back-translation of our example-context, *add1*, is updated to reflect both changes. The results are presented in Figures 5.24 and 5.25, which generalize Figures 5.20 and 5.21 respectively. We first revisit the back-translation of types and expressions, before redefining universal contracts.

Now that linear capabilities $l^{[a,b]}$ are no longer assumed to always have length 1 in the target language, an information discrepancy between source and target pointers arises. The naive back-translation of target pointers to source pointers in the InvertCapability rule made us forget their length $b - a + 1$. We add length information to back-translated pointers by introducing a form of *fat pointer* scheme. We back-translate each linear capability of type $\tau*$ to a pair $(\tau'*, \mathrm{int})$. The first element $\tau'*$ is a pointer to contents of type τ' (the recursive back-translation of type τ). Again, $\tau'*$ simultaneously represents the pointer's address. The second int element is the externalized length of the capability. The null-pointer null is back-translated to the fat pointer $(\mathrm{null}, 0)$. In summary, we update the judgment $\tau \rightsquigarrow_{\mathrm{InvCompileType}} \tau'$, by redefining its InvertCapability rule as follows:

$$\frac{\tau \rightsquigarrow_{\mathrm{InvCompileType}} \tau' \\ (\textsc{InvertCapability})}{\tau* \rightsquigarrow_{\mathrm{InvCompileType}} (\tau'*, \mathrm{int})}$$

The simulation of linear behavior in the source language remains mostly unaltered. The only difference is that linear erasure should now make use of fat pointers. When e.g. back-translating the assignment $x = n[2]$ with n of target type int$**$ from before, an erasure assignment $n[2] = (\text{null}, 0)$ has to be added in the source (remember that $(\text{null}, 0)$ is the fat null-pointer).

The back-translation of expressions also largely remains the same, except for the length and address functions, which we created our fat pointer scheme for. The back-translation hence maps addr$(texp)$ to $texp_{\text{b}}.1$ (the address is the first part of the fat pointer), length$(texp)$ to $texp_{\text{b}}.2$ (the length is the second part of the fat pointer) and *null* to $(\text{null}, 0)$ (as mentioned before). All other cases are still the identity.

We now examine the updated back-translation $add1_{\text{bt}}$ of *add1* in Figures 5.24 and 5.25. The differences with Figure 5.20 have been highlighted in Figure 5.24.

Source (back-translation)

Outcall Stub

```
1  int add1(int* a)
2  //@pre m: a ↦_int [a_1]
3  //@post n: a ↦_int [a_1] * result == a_1 + 1 {
4     (int*,int) m; m = (a,1);
5     int* a^pre; int a_1^pre;
6     a^pre = m.1; a_1^pre = m.1[0];
7     //@collect m
8     int result; (int*,int) n;
9     (result,n) = add1_bt(a,m);
10    guard(n != (null,0)); guard(n.2 == 1);
11    //@flatten n_chunk;
12    int* a^post; int a_1^post;
13    a^post = n.1; a_1^post = n.1[0];
14    guard(result == a_1^post + 1);
15    guard(a^post == a); guard(a_1^post == a_1^pre);
16    return result }
```

Context

```
univ_contr_int*0 (x) ≜ true
univ_contr_int (x) ≜ true
univ_contr_int* (x) ≜ (x != (null, 0)) ? ∃l.
   (n : [x.1 + i ↦_int l[i] | 0 ≤ i < length(l)]
   * length(l) == x.2)
```

```
1  (int,(int*,int)) add1_bt(int* a, (int*,int) m)
2  //@pre  univ_contr_int*0(a) * univ_contr_int*(m)
3  //@post univ_contr_int(result_1) *
4      univ_contr_int*(result_2) {
5    int b;
6    guard(m != (null,0)); guard(0 ≤ 0 < m.2);
7    //@split m_chunk[1]; //@flatten m^0_chunk;
8    b = m.1[0];
9    //@collect m^0,flat_chunk; //@join m^0_chunk m^1+_chunk;
10   return (b + 1,m) }
```

Figure 5.24: Illustrative example: back-translating a context that implements *add1*. The differences with Figure 5.20 have been highlighted.

A first thing to note is the introduction of the guard statements on line 10 of *add1* and line 6 of $add1_{\text{bt}}$. Since we discarded the naive assumption, we have to manually check whether $add1_{\text{bt}}$ returns a capability of size 1 as specified by the contract of *add1*. In this way, these guards reintroduce length information to the universal contract. With the addition of this length guard, *add1* now reflects all guard statements of $add1_{\text{comp}}$, as expected.

Secondly, fat pointers cause minor differences. Lines 4, 6, 8, 10 and 13 of *add1* and lines 1, 6 and 8 of $add1_{\text{bt}}$ have been adjusted to accommodate the fat pointer scheme.

Finally, as mentioned, the pointer case in the universal contracts on lines 2-3 of $add1_{\text{bt}}$

int b; $\rightsquigarrow_{\text{b}}$

b = m[0]; $\rightsquigarrow_{\text{b}}$

return (b + 1,m) $\rightsquigarrow_{\text{b}}$

$\{\text{univ\_contr}_{\text{int}*_0}(a) * \text{univ\_contr}_{\text{int}*}(m)\}._{[a:a][m:m]}$

int b;

$\{\sim\}._{[a:a][m:m][b:0]}$

$\{\text{univ\_contr}_{\text{int}*_0}(a) * \text{univ\_contr}_{\text{int}*}(m) * \text{univ\_contr}_{\text{int}}(b)\}._{[a:a][m:m][b:b]}$

$\{\text{true} * (m.1 \mathrel{!=} (\text{null}, 0))\ ?\ (\text{length}(l) == m.2 * m_{\text{chunk}} : [m.1 + i \mapsto_{\text{int}} l[i] \mid 0 \le i < \text{length}(l)]) * \text{true}\}._{[a:a][m:m][b:b]}$

guard(m != (null,0)); guard(0 ≤ 0 < m.2);

$\{m_{\text{chunk}} : [m.1 + i \mapsto_{\text{int}} l[i] \mid 0 \le i < \text{length}(l)] * \text{length}(l) == m.2 * 0 < m.2\}._{[a:a][m:m][b:b]}$

//@split m_{chunk}[1]; //@flatten $\text{m}^{0}_{\text{chunk}}$;

$\{m^{0,\text{flat}}_{\text{chunk}} : m.1 \mapsto_{\text{int}} l[0] * m^{1+}_{\text{chunk}} : [m.1 + i \mapsto_{\text{int}} l[i] \mid 1 \le i < \text{length}(l)] * \text{length}(l) == m.2 * 0 < m.2\}._{[a:a][m:m][b:b]}$

b = m.1[0];

$\{\sim\}._{[a:a][m:m][b:l[0]]}$

//@collect $\text{m}^{0,\text{flat}}_{\text{chunk}}$; //@join $\text{m}^{0}_{\text{chunk}}$ $\text{m}^{1+}_{\text{chunk}}$;

$\{m_{\text{chunk}} : [m.1 + i \mapsto_{\text{int}} l[i] \mid 0 \le i < \text{length}(l)] * \text{length}(l) == m.2\}._{[a:a][m:m][b:b]}$

$\{\text{univ\_contr}_{\text{int}*_0}(a) * \text{univ\_contr}_{\text{int}*}(m) * \text{univ\_contr}_{\text{int}}(b)\}._{[a:a][m:m][b:b]}$

return (b + 1,m)

$\{\text{univ\_contr}_{\text{int}}(result_1) * \text{univ\_contr}_{\text{int}*}(result_2)\}.$

Figure 5.25: Separation logic proof of the body of $add1_{\text{bt}}$ from the back-translation example.

has been adjusted to allow non-statically sized capabilities, and to use range resources. The new definition looks as follows:

Definition 5.2 ($\text{univ\_contr}_{\tau_{\text{t}}}(exp)$-bis)**.**

$$\text{univ\_contr}_{\tau_{\text{t}}}(exp) \triangleq \text{true } \mathit{if}\ \tau_{\text{t}} = \text{int } \mathit{or}\ \tau_{\text{t}} = \tau_{\text{s}}*_0$$

$$\text{univ\_contr}_{(\tau_1,\ldots,\tau_k)}(exp) \triangleq \text{univ\_contr}_{\tau_1}(exp.1) * \ldots * \text{univ\_contr}_{\tau_k}(exp.k)$$

$$\text{univ\_contr}_{\tau_{\text{t}}*}(exp) \triangleq exp \mathrel{!=} (null, 0)\ ?$$

$$\exists l.\ (n : [exp.1 + i \mapsto_{\tau_{\text{s}}} l[i] \mid 0 \le i < \text{length}(l)] * \text{length}(l) == exp.2)$$

$$\text{given that } \tau_{\text{t}} \rightsquigarrow_{\text{InvCompileType}} \tau_{\text{s}} \text{ and } n \text{ fresh}$$

The case for target-capabilities $\tau_{\text{t}}*$ now states that a linear capability is either the fat null-pointer $(\text{null}, 0)$, or that we have a range resource that allows us to access each element $l[i]$ of the capability (without knowing anything about the value of $l[i]$, hence the existential quantification over l), where $exp.1$ is the capability's address and $exp.2$ its length. The fat-pointer scheme appears here, because the universal contracts are used to describe the permissions associated with back-translated τ_t*-typed variables id_p, and these back-translated variables are fat pointers. Again, the CONSEQ rule will allow us to introduce or eliminate the universal quantification.

Note that if we solely wanted to support non-statically sized capabilities, we might as well have defined the τ_t*-case as follows:

$$exp \mathrel{!=} (null, 0)\ ?\ \exists l.\ (n : exp.1 \mapsto_{\tau_s} l * \mathrm{length}(l) == exp.2)$$

However, this formulation would not be sufficiently general to handle the back-translation of nested pointers in Section 5.6.4, since it does not allow for nested universal contracts. Specifically, when back-translating nested linear capabilities, each element of the back-translated capability is again a pointer whose permissions are described by a universal contract. This will require a recursive universal contract call inside the range resource in the pointer case above.

The use of range resources in universal contracts necessitates some changes in the different back-translation phases of Figure 5.22. Since the separation logic rules that are applied in the RULE APPLICATION phase require array resources in their pre- and postcondition, we need to convert the range resources from the universal contract to array resources before applying the rule, and back afterwards. This is what happens on lines 7 and 9 of *add1*$_{\mathrm{bt}}$ in Figure 5.24; the resource we have for the fat source pointer *m* is a range expression, but we need an array resource to apply ARRAYLKUP on line 8. Afterwards, we need a range resource again to satisfy the universal contract. Lines 7-13 in Figure 5.25 perform this conversion between array and range resources in both directions. A length-1 range resource m^0_{chunk} (not explicitly shown) is split from m_{chunk} and flattened to the array resource $m^{0,\mathrm{flat}}_{\mathrm{chunk}}$ on lines 7-9. The rule ARRAYLKUP can now be applied on lines 9-11. Afterwards, the resource is recollected to reobtain the range resource m^0_{chunk} and rejoined to the rest of the range resource on lines 11-13. In other words, the CONCRETIZATION phase for each back-translated statement uses split and flatten statements to convert range resources to array resources, and the UNIVERSALIZATION phase will use collect and join statement to do the inverse.

A similar conversion between array and range resources is now required in the back-translated stub *add1*, since the function *f* uses array resources, whereas *add1*$_{\mathrm{comp}}$ uses range resources. Lines 7 and 11 of *add1* switch between the two representations, similarly to lines 7-13 of Figure 5.25. Line 11 of *add1* can simply be inserted (together with the new guard on line 10) into the proof of Figure 5.23 to make it go through with the new definition of universal contracts.

In the general case, lines 7 and 11 of *add1* will not suffice to convert between a range and array representation of resources. The reason for this is that the universal contract consists of a range resource containing length-1 array resources. If, e.g., the resource *m* in the precondition of *add1* were to have a length greater than 1, split statements would be required before the collect on line 7 of *add1*. The technical report defines procedures to perform this conversion in the general case.

5.6.3 Back-translating misbehaving contexts

Having defined a more general back-translation, this section briefly investigates how the back-translation handles alternative, misbehaving implementations of *add1*. Notice that swapping out the implementation of *add1* does not affect the back-translated stub *add1*. These misbehaving target contexts will always get stuck; either due to a failing guard statement or due to the operational semantics. Since these alternative implementations of *add1* and their back-translations have to satisfy a specific equi-terminating instance of **BT-⇕**, their back-translation should get stuck as well. The different types of misbehavior are listed in Figure 5.26, and illustrated by means of a possible implementation of the body of *add1*. They can be subdivided into three main categories; functional misbehavior, out of bounds accesses and breaking the restrictions of linearity. We now discuss these in order.

First, *add1* *functionally misbehaves* when it does not satisfy one or more non-spatial conditions that *f* expects *add1* to uphold in its postcondition. This will cause the failure of one or more guard statements in both *add1* and *add1*$_{\text{comp}}$, ensuring equi-termination. In the concrete example from Figure 5.26, the variable *b* is decremented

Type of Misbehavior	**Example Body**

Functional	int b;
	b = m[0];
	return (b − 1,m)

Out of bounds	int b;
	b = m[1];
	return (b + 1,m)

Breaching linearity:
• Storing
	int b;
	b = m[0];
	int* n; n = m;
	return (b + 1,m)}

• Duplicating	int b;
	b = m[0];
	(int*,int*) n; n = (m,m);
	m = n.1;
	return (b + 1,m)

Figure 5.26: Examples of different classes of misbehaving implementations of *add1*.

instead of incremented, causing the guard statement on line 14 of *add1* in Figure 5.24 and the corresponding guard on line 9 of *add1*$_{\text{comp}}$ in Figure 5.5 to fail. Similarly, the example could have set the value of $m[0]$ to 0 before returning, reduced the bounds of m or tried to return a different linear capability altogether, each time making a different pair of guard statements fail.

Secondly, *out of bounds accesses* happen when *add1* reads from or writes to linear capabilities outside their intended bounds. The example in Figure 5.26 contains a read from index 1 of m, causing ArrayMut from Figure 5.4 to get stuck when f_{comp} provides a value for m with a length of 1. On the source level, the second guard statement on line 6 of *add1*$_{\text{bt}}$ in Figure 5.24 (now enforcing $0 \leq 1 < m.2$) would ensure equi-termination.

Lastly, *add1* can try to *breach linearity guarantees* and keep a copy of the linear pointer m, either by trying to *store* it for later use, or by using multiple copies of m in one statement, thereby trying to *duplicate* m.

The storing example in Figure 5.26 stores the value of m in n for later use. Currently, n in this example is modeled as local state for simplicity reasons. However, it would be more useful for a malicious context to store linear capabilities in context-global state. Our target language model does not include such global state, but it could be simulated by passing global state around using an additional parameter.

The third line of the example is back-translated to (int*,int) n; n = m; m = (null,0). The last statement emulates erasure and ensures that both the first guard statement on line 10 of *add1* and the corresponding guard in *add1*$_{\text{comp}}$ fail. Notice that, if the postcondition of *add1* did not require the return of the resource n, then the guard statements on line 10 of *add1* would not have been generated, and the storing example would not have been problematic.

The duplication example in Figure 5.26 duplicates the value of m in n, before perhaps storing it for later use or causing aliasing in the return value. The third line of the example is back-translated to ((int*,int),(int*,int)) n; guard(false); n = (m,m); m = (null,0). Remember that a guard(false)-statement is inserted when back-translating code that attempts to duplicate a linear capability. The guard statement ensures equi-termination, emulating the target-level semantics getting stuck.

5.6.4 Back-translating nested pointers

The back-translation in Section 5.6.2 did not yet support back-translating nested pointer types. This section will fill this gap, by defining universal contracts for back-translated, nested pointers and investigating how they are used in statements. Notice that back-translated nested pointers can never appear in the universal contracts of back-translated boundary functions, since Section 5.4.4 required boundary function contracts to solely contain array resources, which cannot result in nested pointers after compilation. Although alleviating this restriction is future work, this currently means that the conversion between range and array resources in back-translated stubs does not need to be generalized in this section.

We now investigate what the universal contract for a back-translated pointer of target type int∗∗ looks like. The general case can easily be derived from this, but contains some additional uninteresting clutter, to do with resource names n. The τ_t*-case of the universal contracts from Section 5.6.2 contains a range resource

$$n : [\mathit{exp}.1 + i \mapsto_{\tau_s} l[i] \mid 0 \leq i < \mathrm{length}(l)]$$

that in turn contains array resources to access each individual element of the pointer represented by *exp*. Unfortunately, this does not represent the permissions carried by nested pointers. The reason is that $l[i]$ itself is not necessarily a permissionless value with universal contract true, but rather a value of a type that carries its own permissions, again described by a universal contract. In our case where $\tau_t* = \mathrm{int}**$ and $\mathrm{int}** \rightsquigarrow_{\mathrm{InvCompileType}} ((\mathrm{int}*, \mathrm{int})*, \mathrm{int})$, the logical list element $l[i]$ represents a value of type $(\mathrm{int}*, \mathrm{int})$, that we should again define an *inner* universal contract for, using a range resource. The only difference between the outermost and inner universal contracts was highlighted in Section 5.3.1 already; the nested universal contracts do not require chunk names, as range resources are reified as a whole during compilation.

Given these observations, we can define universal contracts for int∗∗-pointers as follows:

Definition 5.3 ($\mathrm{univ\_contr}_{\mathrm{int}**}(\mathit{exp})$).

$$\mathrm{univ\_contr\_inner}_{\mathrm{int}*}(\mathit{exp}) \triangleq \mathit{exp} \mathrel{!=} (\mathit{null}, 0)\ ?$$
$$\exists l.\ ([\mathit{exp}.1 + j \mapsto_{\mathrm{int}} l[j] \mid 0 \leq j < \mathrm{length}(l)] * \mathrm{length}(l) == \mathit{exp}.2)$$
$$\mathrm{univ\_contr}_{\mathrm{int}**}(\mathit{exp}) \triangleq \mathit{exp} \mathrel{!=} (\mathit{null}, 0)\ ?\ \exists l.\ (n : [\mathit{exp}.1 + i \mapsto_{(\mathrm{int}*,\mathrm{int})} l[i]\ *$$
$$\mathrm{univ\_contr\_inner}_{\mathrm{int}*}(l[i]) \mid 0 \leq i < \mathrm{length}(l)] * \mathrm{length}(l) == \mathit{exp}.2)$$
$$\text{given that } n \text{ fresh}$$

The interesting aspects have been highlighted; notice the nesting of the inner contracts $\mathrm{univ\_contr\_inner}_{\mathrm{int}*}$ inside the outer contract $\mathrm{univ\_contr}_{\mathrm{int}**}$ and the single range resource name n. This nested contract structure would not have been possible using a single resource name n if universal contracts used array resources instead. This is the motivation for the reformulation in terms of range resources in Section 5.6.2. The above structure is easily generalized to arbitrary target types τ_t, by allowing inner contracts to contain more deeply nested inner contracts.

Excerpt from Figure 5.24

```
5 int b;
6 guard(m != (null,0)); guard(0 ≤ 0 < m.2);
7 //@split $m_{chunk}$[1]; //@flatten $m^{0}_{chunk}$;
8 b = m.1[0];
9 //@collect $m^{0,flat}_{chunk}$; //@join $m^{0}_{chunk}$ $m^{1+}_{chunk}$;
```

Back-translation of int∗ b; b = m[0]

```
1 (int∗,int) b;
2 guard(m != (null,0)); guard(0 ≤ 0 < m.2)
3 //@split $m_{chunk}$[1]; //@flatten $m^{0}_{chunk}$;
4 b = m.1[0]; m.1[0] = null;
5 //@collect $m^{0,flat}_{chunk}$; //@join $m^{0}_{chunk}$ $m^{1+}_{chunk}$;
```

Figure 5.27: Back-translation of line 2 of *add1* with and without nested pointers.

The back-translation of statements containing nested pointers happens very similarly to the non-nested examples we saw before, except that more intricate emulation of linearity in the source language is often required. To illustrate this, we rewrite line 2 of *add1* in Figure 5.20 with m now of type int∗∗, obtaining int∗ b; b = m[0]. This line gets back-translated to the code on the right in Figure 5.27. Notice how small the differences with the original back-translation on the left are: only lines 1 and 4 differ, because b is now a linear value itself, instead of a duplicable integer. The proof is also very similar to the one from Figure 5.25, except that on line 10 of this figure, we would need to consume the inner universal contract for $m.1$ at index 0 to derive that the new value of b satisfies the universal contract of its type int∗. Since universal contracts are linear, we need the erasure of $m.1[0]$ on line 4 of Figure 5.27 to re-establish the universal contract of $m.1$ at index 0 afterwards.

5.7 Simulation Relations

This section takes the black-box relations for correctness and security, R and S, from Section 5.5.4, and decomposes both in Sections 5.7.1 and 5.7.2 respectively. Formulated differently, this section provides more details on how to prove the **Adequacy** proof step in Figures 5.17 and 5.18.

5.7.1 Decomposing R

We decompose R first, since its decomposition is easier. The reason is that the proof of **Comp-**$\Updownarrow$ only involves compiled components, whereas **BT-**$\Updownarrow$ requires simulating both compiled and back-translated components.

To illustrate this section, we need a source program and its compilation. We assume the source component from Figure 5.5 as our verified source component $\vdash \mathbf{s}$, and an arbitrary verifiable source context $(\mathfrak{C}_{\mathrm{s}}, \mathit{id})$ that implements and exports *add1*, with the main function *main* from Figure 5.19 as our main function *id*. Again, we ignore

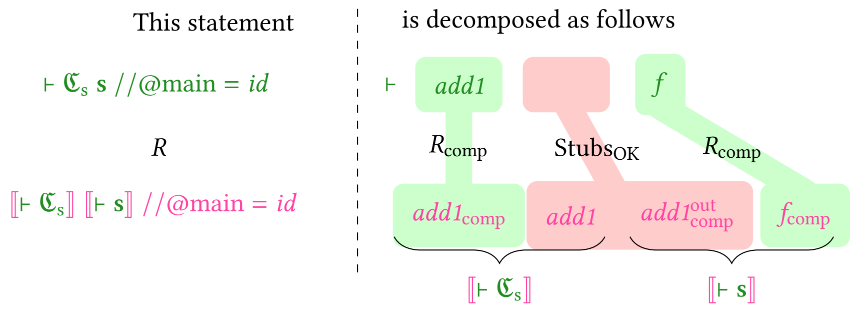

This statement
$\vdash \mathfrak{C}_{\mathrm{s}}\ \mathbf{s}\ //@\mathrm{main} = id$
R
$\llbracket \vdash \mathfrak{C}_{\mathrm{s}} \rrbracket\ \llbracket \vdash \mathbf{s} \rrbracket\ //@\mathrm{main} = id$
is decomposed as follows
$\vdash$
$add1$
f
R_{comp}
$\mathrm{Stubs}_{\mathrm{OK}}$
R_{comp}
$add1_{\mathrm{comp}}$
$add1$
$add1^{\mathrm{out}}_{\mathrm{comp}}$
f_{comp}
$\llbracket \vdash \mathfrak{C}_{\mathrm{s}} \rrbracket$
$\llbracket \vdash \mathbf{s} \rrbracket$

Figure 5.28: Visual representation that illustrates the decomposition of R (inspired by the schemata of Devriese et al. [45]).

the actual implementation of *main* and its compilation *main* in respectively $\vdash \mathbf{s}$ and t to avoid uninteresting clutter. Including compiled components, we then have the following (we repeat the same notational abuse from Section 5.6 where the names of functions can be used to represent their entire implementation):

$$\begin{aligned}
\mathbf{s} &= f\ //@\mathrm{import}\ add1 \\
(\mathfrak{C}_{\mathrm{s}}, id) &= ((add1\ //@\mathrm{export}\ add1\,),\ \ main) \\
\llbracket \vdash \mathbf{s} \rrbracket &= f_{\mathrm{comp}}\ add1^{\mathrm{out}}_{\mathrm{comp}}\ //@\mathrm{import}\ add1 \\
(\mathfrak{C}_{\mathrm{t}}, id) &= (\llbracket \vdash \mathfrak{C}_{\mathrm{s}} \rrbracket, id) = ((add1\ add1_{\mathrm{comp}}\ //@\mathrm{export}\ add1\,),\ \ main)
\end{aligned}$$

The lower two lines follow from the CompVerif rule of our compiler in Figure 5.11. Note how we added the superscript out to the generated outcall stub $add1^{\mathrm{out}}_{\mathrm{comp}}$, to distinguish it from the compiled context function $add1_{\mathrm{comp}}$. The context function *add1* is the generated incall stub.

The statement we need to prove as part of **Compatibility** is shown on the left in Figure 5.28. The right side of this figure shows how this statement, once proven, implies equi-termination of our source and target programs, or in other words, proves **Adequacy**. We now focus on explaining this right part.

When simulating our compiled code, a distinction has to be made between execution within the individual source components $\mathbf{s}$ and $\mathfrak{C}_{\mathrm{s}}$, and execution when an outcall stub is called or is being returned from, i.e. when a transition between $\mathbf{s}$ and $\mathfrak{C}_{\mathrm{s}}$ (or back) occurs. Notice that an outcall stub is always called first, and then execution transitions to the incall stub of the component that is being called. When returning, the order is inverted. In general, we will refer to this sequence of either two calls or two returns as a *component switch*.

The reason R (and S) is not a simulation relation itself (as we mentioned in Section 5.5.4) is that R does not consider states to be related during in- or outcalls, but rather consists of two separate parts; an actual simulation relation R_{comp} and a *connecting lemma* Stubs_{OK} to link up different instances of R_{comp} across component switches.

First, R consists of a simulation relation R_{comp} that relates source code to its compilation and models how the compiler produces equi-terminating code in the target language. The relation R_{comp} is solely used to reason within a single domain of trust, i.e. within a single source and target component (hence the comp subscript) during execution. The simulation halts right before a component switch occurs. More concretely, the technical report proves that R_{comp} satisfies the following definition of a *forward*, *strong* simulation relation (inspired by the definition of a *Simulation relation with multiple matching steps* in Chlipala [34]):

Definition 5.4 (source-to-target forward simulation relation). *Given a relation R_{comp} relating source states $\mathbf{st} = \langle \overline{\mathbf{s}}, \mathbf{h} \rangle \mid \overline{\vdash \mathbf{c}}$ to target states $\mathsf{st} = \langle \overline{\mathsf{s}}, \mathsf{h} \rangle \mid \overline{\mathsf{c}}$. If the following 2 properties hold, then R_{comp} is a source-to-target forward simulation relation:*

1. *The first property is used to guarantee equi-termination in the proof of* **Adequacy**, *when we know that the source program terminates:*

   $$\forall\, P, Q, \mathbf{s}, \mathbf{h}, \mathsf{s}, \mathsf{h}, \mathsf{c}.\ (\langle \mathbf{s}, \mathbf{h} \rangle \mid \{P\}\ \text{return}\ \{Q\})\ R_{\text{comp}}\ (\langle \mathsf{s}, \mathsf{h} \rangle \mid \mathsf{c}) \Rightarrow \mathsf{c} = \text{return}$$

   It states that a terminated source statement (i.e. a single return statement) must correspond to a terminated target statement.
2. *The second property is the inductive part of the simulation relation:*

   $$\forall\, \mathbf{st}, \mathsf{st}, \mathbf{st}'.\ \mathbf{st}\ R_{\text{comp}}\ \mathsf{st} \land \mathbf{st} \hookrightarrow \mathbf{st}' \Rightarrow \exists\, \mathsf{st}'.\ \mathsf{st} \hookrightarrow^{+} \mathsf{st}' \land \mathbf{st}'\ R_{\text{comp}}\ \mathsf{st}'$$

   Note that this condition requires the target operational semantics $\hookrightarrow$ to perform at least one step, denoted $\hookrightarrow^{+}$. Remember that the source level makes use of the lifted operational semantics.

Second, R also requires proving a connecting lemma Stubs_{OK}. This lemma essentially states that if the source and target states are related by R_{comp} before a component switch, they will still be related after the switch, and R_{comp} can hence continue simulating. Additionally, Stubs_{OK} does not allow target code to get stuck while executing code in an in- or outcall stub, since source code does not have any stub code to execute, and this would otherwise break equi-termination. Fortunately, R only relates correctly-behaving programs that live up to their contracts, and the guards will never fail. The situation will be different in the security direction, where we consider potentially misbehaving contexts.

Skipping the execution of the main functions *main* and *main* that simply perform set-up and pass control to f and f_{comp}, the right side of Figure 5.28 now ensures equi-termination as follows; execution starts off on the far right in the function f (f_{comp} in the target), and R_{comp} simulates (using the second property in the above definition) until f_{comp} is about to perform an outcall to $\mathit{add1}^{\text{out}}_{\text{comp}}$. At this point, the Stubs_{OK} lemma is applied, guaranteeing us that we can bypass the stubs $\mathit{add1}^{\text{out}}_{\text{comp}}$ and *add1* in the middle and resume simulation under R_{comp} at the left side, in the function *add1* ($\mathit{add1}_{\text{comp}}$ in the target). Once *add1* is about to return, the Stubs_{OK} lemma is again applied to make the inverse transition back to f. When execution terminates in *main* at the source level, the first property in the above definition is applied, thereby proving that the target program has also terminated (in *main* in this case).

5.7.2 Decomposing S

Having expanded our toolbox in the previous section, we now study the decomposition of the relation S. As stated before, this decomposition is slightly more involved, since both compiled and back-translated components are present in the statement of **BT-⇕**.

To illustrate this section, we reuse the example described in the introduction of Section 5.6 (including the *main* function defined there, the fact that we keep the main functions implicit, and the abuse of notation for functions). To reiterate, we had the following:

$$\begin{aligned}
\mathbf{s} &= (f \ //@\text{import } \mathit{add1}\ )\\
(\mathfrak{C}_{\text{t}}, \mathit{id}) &= ((\mathit{add1}\ //@\text{export } \mathit{add1}\ ),\ \mathit{main})\\
[\![\vdash \mathbf{s}]\!] &= f_{\text{comp}}\ \mathit{add1}_{\text{comp}}\ //@\text{import } \mathit{add1}\\
(\mathfrak{C}_{\text{s}}, \mathit{id}) &= (\langle\!\langle \mathfrak{C}_{\text{t}} \rangle\!\rangle, \mathit{id}) = ((\mathit{add1}\ \mathit{add1}_{\text{bt}}\ //@\text{export } \mathit{add1}\ ),\ \mathit{main})
\end{aligned}$$

Figure 5.29 is similar to Figure 5.28, showing the **Compatibility** statement on the left and how it implies adequacy on the right. We now focus on explaining this right part.

In this case, we need to make a distinction between not two, but four different modes of execution. First off, there are two different regular, intra-component modes of execution; either execution is happening within the source component **s** and its compilation, or within the component $\mathfrak{C}_{\text{t}}$ and its back-translation. Furthermore, two different transitions can now be made; either **s** performs an outcall to the context, or the context performs an incall to **s**. Since f is not an exported function, this second scenario cannot occur in our simple example. Unlike in Section 5.7.1, incalls and outcalls do not occur in pairs, since the target context does not result from compilation and hence does not generate its own stubs. Another distinction with Section 5.7.1 is that in- and outcalls now execute code in both the source and target languages, since stubs such as $\mathit{add1}_{\text{comp}}$ are back-translated into the source context.

Again, S as introduced in Section 5.5.4 was not really a simulation relation, but rather consisted of the four aforementioned parts: two simulation relations R_{comp} and S_{comp}

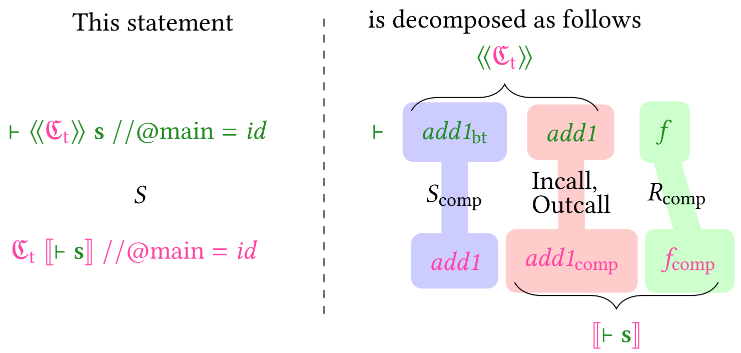

This statement
is decomposed as follows
$\langle\langle \mathfrak{C}_t \rangle\rangle$
$\vdash \langle\langle \mathfrak{C}_t \rangle\rangle$ s //@main = id
S
$\mathfrak{C}_t$ $[\![\vdash$ s$]\!]$ //@main = id
⊢
$add1_{bt}$
add1
f
S_{comp}
Incall, Outcall
R_{comp}
add1
$add1_{comp}$
f_{comp}
$[\![\vdash$ s$]\!]$

Figure 5.29: Visual representation that illustrates the decomposition of S (inspired by the schemata of Devriese et al. [45]).

(with R_{comp} as defined before) and two connecting lemmas Incall and Outcall.

First, S consists of two different simulation relations R_{comp} and S_{comp}, both used to reason within a single domain of trust, i.e. halting before a component switch. The relation R_{comp}, discussed in the previous section, is used to reason about code and its compilation. On the other hand, the relation S_{comp} relates target code to its back-translation, and models how the back-translation produces equi-terminating code in the source language.

Since S_{comp} performs a target-to-source simulation of back-translated code in terms of target code, whereas R_{comp} performed source-to-target simulation of compiled code in terms of the original source code, the technical report defines a second, different notion of *forward, strong* simulation relation. The relation S_{comp} is then proven to satisfy this notion. The second version of simulation we use is defined as follows (again inspired by the definition of a *Simulation relation with multiple matching steps* in Chlipala [34]):

Definition 5.5 (target-to-source forward simulation relation). *Given a relation S_{comp} relating source states* $\mathbf{st} = \langle \bar{\mathbf{s}}, \mathbf{h} \rangle \mid \overline{\vdash \mathbf{c}}$ *to target states* $\mathsf{st} = \langle \bar{\mathsf{s}}, \mathsf{h} \rangle \mid \bar{\mathsf{c}}$ *. If the following 2 properties hold, then S_{comp} is a target-to-source forward simulation relation:*

1. *The first property is used to guarantee equi-termination in the proof of* **Adequacy***, when we know that the target program terminates:*

   $$\forall\, \mathbf{s}, \mathbf{h}, \mathbf{c}, \mathsf{s}, \mathsf{h}.\ (\langle \mathbf{s}, \mathbf{h} \rangle \mid \mathbf{c})\ S_{\text{comp}}\ (\langle \mathsf{s}, \mathsf{h} \rangle \mid \text{return}) \Rightarrow \exists\, P, Q.\ \mathbf{c} = \{P\}\ \text{return}\ \{Q\}$$

   It states that a terminated target statement (i.e. a single return statement) must correspond to some valid proof of a terminated source statement.

2. *The second property is the inductive part of the simulation relation:*

$$\forall\, \mathbf{st}, \mathrm{st}, \mathbf{st}'.\ \mathbf{st}\ S_{\mathrm{comp}}\ \mathrm{st} \wedge \mathrm{st} \hookrightarrow \mathrm{st}' \Rightarrow \exists\, \mathbf{st}'.\ \mathbf{st} \hookrightarrow^{+} \mathbf{st}' \wedge \mathbf{st}'\ S_{\mathrm{comp}}\ \mathrm{st}'$$

Note that this condition again requires the source operational semantics $\hookrightarrow$ *to perform at least one step, denoted* $\hookrightarrow^{+}$*. Remember that the source level makes use of the lifted operational semantics.*

Second, S also requires proving two connecting lemmas Incall and Outcall, used for incalls and outcalls respectively. The Outcall lemma states that if R_{comp} holds in the verified component $\mathbf{s}$, we can perform an outcall to the context, and after executing the outcall stubs in both source and target language, S_{comp} will hold in the context. Similarly, when returning from the outcall, R_{comp} will still hold in $\mathbf{s}$. A crucial difference with last section, is that execution is now allowed to get stuck in the outcall stubs, as long as it gets stuck in both source and target language, preserving equi-termination. The Incall lemma makes similar claims, but for incalls.

The right side of Figure 5.29 illustrates the decomposition of S. Interestingly, note how *add1*$_{\mathrm{comp}}$ is part of the compiled, verified component in the target language, whereas its back-translation *add1* is part of the source context. Equi-termination is proved as follows (again ignoring the execution of the main functions *main* and *main* in our example); execution starts off on the far right in the function f (f_{comp} in the target), and R_{comp} simulates (similarly to what we saw in the previous subsection) until f and f_{comp} are about to perform an outcall to the pair of outcall stubs *add1* and *add1*$_{\mathrm{comp}}$.

In order to be able to apply the Outcall lemma here (and similarly for Incall), both the source and target code must reach their respective outcalls to *add1* and *add1*$_{\mathrm{comp}}$ simultaneously during simulation (and similarly, return from them simultaneously). This follows easily from the definitions of R_{comp} and S_{comp} in the technical report. Consequently, the Outcall lemma can be applied, guaranteeing that either execution gets stuck in both stubs, or we can bypass the stubs in the middle and resume simulation under S_{comp} at the left side, in the function *add1*$_{\mathrm{bt}}$ (*add1* in the target). Once *add1*$_{\mathrm{bt}}$ is about to return, the Outcall lemma is again applied to make the inverse transition back to f.

Proof Conclusion *In the previous Sections 5.5, 5.6 and 5.7, we discussed the main intuitions behind the full abstraction proof of our compiler, including the back-translation, in an example-driven way. These sections should provide the reader with sufficient anchoring points to understand the full proof in the aforementioned technical report [164], in case they are interested in the more formal and detailed approach.*

5.8 Discussion and future work

This section first provides more detail on two challenges in making our compiler more broadly applicable, and subsequently discusses the benefits of semantically deriving our separation logic rules, instead of stating them syntactically.

5.8.1 Gradual Verification

Function signatures are modified by our compiler, as apparent from its definition in Section 5.4: additional parameters and/or return values that represent the memory resources that are transferred are added. Additional effort is hence still required by third-party developers to produce code that follows our target-level calling convention. Two scenarios are possible.

First of all, a developer could write verified code themselves and compile it using our compiler, gaining the same secure compilation guarantees that our compiled code does. Although this could be realistic in some settings, it goes against our original goal of allowing interaction of our compiled code with arbitrary, non-verified attacker code.

Secondly, the developer could write unverified code in the target language. This code must then be written to call and be called with the modified function signatures. This might be realistic for applications like the video player with codec plugin described in the introduction. However, we would also like to support a form of gradual verification, where we can take a large, unverified codebase, verify the critical parts and securely combine them with the rest. This type of use case is currently only supported when boundary functions solely use integer arguments and return values and do not receive or return memory resources, since the declaration of such functions is not altered during compilation. Even with this strong restriction, our strong security results might still be useful in some scenarios.

We plan to explore two ideas for extending our approach to large, partially-verified codebases: either based on the use of an automatic verifier on the unverified code, like Smallfoot [21], Space Invader [48, 20], Infer [27] or SLAyer [22], or on a kind of universal contract for unverified code in terms of a pure predicate similar to the lowval predicate of Swasey et al. [152]. Such an approach could be valuable in practice, as many large code bases contain small, isolated components whose security is of high value and for which the verification effort might be realistic and cost-effective.

5.8.2 Extending the Source Language

A second direction we want to expand our work in, is to extend the compiler itself. As mentioned in the introduction, this paper contains but the first steps towards a practically applicable secure compilation scheme. Notably, the source language only consisted of simple resources in the separation logic, had a simple type system and featured restrictions on the form of boundary contracts. We now discuss some ideas for extensions in these three directions in order. We do believe all suggested extensions to be within reach.

Resources In this paper, we support only two kinds of spatial predicates, describing array and range resources. We believe our approach can be extended to a more general form of predicates, by relying on a notion of capability *sealing*. Support for such predicates would also allow us to formalize memory de-allocation, i.e., a free-statement. The difficult part in supporting free is that the authority to deallocate a block of memory needs to be represented separately from the authority to access the memory (i.e. our array points-to predicate). This is because the latter can be subdivided, but the former should not be, since most memory allocators rely on the entire block being deallocated together. To accommodate this, separation logics like VeriFast represent the authority to deallocate memory with a special *malloc resource* abstract predicate. We could do the same and compile this resource in the same way as the discussed general predicates.

Type system As mentioned in the introduction, the most obvious feature missing from the source language is support for recursive data types, e.g. in the form of C-like structs. We believe the type system will scale in parallel with the introduction of resources to represent more complex permissions, e.g. *struct* types would be introduced in parallel with the general predicate resources discussed above. The general resource reification principles demonstrated in the current submission would remain the same.

Boundary contract restrictions Loosening boundary contract restrictions corresponds to loosening the constraints on calls to untrusted code. It seems possible to allow non-fixed-length array resources and range resources to appear in boundary contracts, by reifying (nested) foreach loops in our stubs, given some proof changes. Reification of foreach loops could also be used to allow quantifiers over finite domains in boundary contracts (although efficiency remains an open question here).

Supporting a more intricate logic internally Another thing we intend to investigate, is whether we can support a richer logic within trusted components, while retaining the current, more limited expressivity in the boundary contracts. For example, the popular Iris separation logic [80] supports invariants and custom ghost state, which we might want to use internally, but not share with the adversary through boundary functions, as this would likely require us to reify all involved logical constructs (e.g. masks and view shifts). One idea is to introduce a second type of *internal* points-to chunk that cannot be shared with untrusted code, nor mentioned in boundary contracts. This new type of points-to chunk would enjoy the full expressivity of the logic, but could only be used privately, whereas the pre-existing points-to chunk would only be usable in logical expressions of limited expressivity (e.g. it could not appear inside invariants), but could be mentioned in boundary contracts. Separation logic state corresponding to internal points-to chunks might then be safely erasable by the compiler, at the cost of copying internal values over from internal non-linear capabilities to external linear capabilities when internal state needs to be shared with the adversary. The viability of this approach remains to be investigated.

5.8.3 Semantic Separation Logic Rules

The separation logic rules presented in Section 5.4 are *syntactic* in nature; they are presented as axioms in the logic, without any formal justification. In other words, they are assumed to be part of the *trusted computing base (TCB)* of the source language. To decrease the TCB and make our approach more foundational, it would be worthwhile to lift these rules out of the TCB. There are two ways to do so:

- Perform a proof of *adequacy* for our current rules, proving that the syntactic rules are adequate with respect to the operational semantics.
- Derive the rules *semantically* from the operational semantics, on top of some appropriate program logic. Preferably, this program logic would contain a built-in proof of adequacy for any Hoare triples derived this way. For example, the Iris program logic framework [79, 80, 89] (and corresponding proof assistant, implemented in Coq, in case we mechanize our results) would likely be a good fit to perform these derivations in, as it has proofs of adequacy for its weakest precondition judgment.

The advantage of deriving rules semantically over our current approach seems to be two-fold; it would be possible to derive a back-translation that does not have to be syntactic in nature, and the Concretization phase in Figure 5.22 is likely not required. We briefly discuss both advantages in the following two paragraphs.

Because of the syntactic nature of our separation logic rules, it is currently impossible to “look under the hood" and derive what we call a *semantic back-translation.* By semantic back-translation, we mean a back-translation where not each individual back-translated statement is proven to preserve universal contracts by a derived separation logic rule, but rather, it suffices to prove that the overall back-translated code respects universal contracts (similar to the proofs of semantic type safety in RustBelt [78] or the semantic proofs of wrapper contracts that Sammler et al. [136] use). In this setting, we would still have some notion of "universal contract", but now defined in terms of some underlying base logic. It is not unlikely, however, that it would still be easiest to derive semantic verification rules for individual code blocks based on universal contracts, as we currently do, rather than try to construct an end-to-end proof of a function's universal contract directly. This would need to be investigated further.

As for the necessity of the Concretization phase in Figure 5.22; this too has to do with the fact that our separation logic rules are derived syntactically. What we illustrate in Figure 5.22 is essentially a syntactic derivation of a proof rule for the back-translation of each type of target statement that operates on the shapes of resources (i.e. universal contracts), instead of using some concrete pre- and postcondition like the rules we presented in Section 5.4. This new, universal rule is derived by proving that each back-translated block can be proven to respect a contract that consists of universal contracts of local variables. If our rules were to be derived semantically, it would be possible to derive these rules in a more direct way, without going through the concrete syntactic-style rules. It seems likely that we would be able to use information from the operational semantics themselves, instead of using the explicit "guard" statements in the Concretization phase, to derive rules that operate on universal contracts. This would reduce the Concretization phase to only inserting a "guard(false)" statement to simulate linearity where necessary.

5.9 Related work

Our work builds on three research lines with a long and rich history: capability machines, separation logic and full abstraction. It is not feasible to give complete surveys of these three research lines here, so we just provide some pointers to key papers. For an excellent introduction to separation logic and references, we refer to O'Hearn [116].

Capability machines have been studied for decades. Levy [94] provides a good survey of early systems. With the increased need for security and fine-grained protection, there is a renewed interest in these machines, or in generalizations where the hardware can track even more metadata. Two influential recent systems are the CHERI system developed in Cambridge [176, 33], and the SAFE machine developed within the CRASH/SAFE project [87, 41, 40]. Linear capabilities have already been implemented in the latter. Skorstengaard et al. [144] have used them in a secure calling convention

`StkTokens`, and an early design for their implementation in CHERI is in the latest CHERI ISA Spec [175].

To formalize secure compilation, we use the property of fully abstract compilation [1], like many previous results [e.g., 2, 57, 121, 108, 45, 144]. We refer to Patrignani et al. [122] for an overview of the field. Recent research has investigated other formal characterisations of secure compilation: robust safety preservation [152, 124], trace-preserving compilation [125] and robust hyperproperty preservation [60]. Although we only prove fully abstract compilation, it is important to understand that most of our proof consists of the construction of the back-translation and its properties, and those parts could be immediately reused to prove many of the alternative properties.

The fact that our back-translation depends just on the context, not on the compiled program, suggests that our compiler actually also satisfies the property that Garg et al. [60] call Relational Hyperproperty Preservation (RrHP). Technically, our backtranslation and its use of universal contracts is reminiscent of the use of universal types and universal embeddings in previous work [108, 45].

Our work is also related to the body of work on contract enforcement, where the enforcement of higher-order contracts, and the assignment of blame on contract violations has received significant attention. A recent Functional Pearl [46] provides an in-depth discussion of this line of work. Bader et al. [16] recently demonstrated how dynamic checking of Hoare logic contracts can be obtained using the general AGT framework for gradual typing [59].

Directly related to our work are other approaches to dynamic checking of separation logic. The main challenge for such dynamic techniques is the enforcement of framing. Nguyen et al. [109] use a heap coloring technique and run time checks at every method invocation and field access in unverified code to check framing. The performance overhead of this approach is substantial, and it is limited to safe languages such as Java. Agten et al. [7] were the first to propose a contract checking approach for C, but, as we discussed in the Introduction, their approach is not fully abstract, it only guarantees integrity: safety properties expressed in separation logic assertions within a verified module are guaranteed to hold at run time in the presence of an unverified context, but confidentiality properties are lost. Building further on Agten et al. [7]'s work, van Ginkel et al. [156] developed a separation-logic-based specification language for Intel SGX enclaves, that allows the automatic generation of contract checking functions at the enclave's trust boundaries. The contract checking approach of van Ginkel et al. [156] can be translated to the capability domain as a coarse-grained approach where a single CHERI compartment would be used to encapsulate one trust domain, rather than having fine-grained linear capabilities. This will work best for simple component interfaces that few complex data structures pass through, as any pointers passed through the interface need to be deep-copied.

Lastly, the notion of universal contracts introduced here has implicitly been used by other papers to describe the most general constraints that arbitrary adversarial code satisfies. More concretely, Skorstengaard et al. [142, 144] encode the guarantees obtained when executing adversarial assembly code in the fundamental theorem of their logical relations. The semantic type systems defined by Jung et al. [78] and Sammler et al. [136] are similarly used to specify the behavior of arbitrary untrusted code. The lowval predicate, used to describe safely shareable values, defined by Swasey et al. [152], again serves a similar purpose. More generally, the guarantees obtained when syntactically restricting adversaries can be seen as an instance of parametricity. Note that parametricity should be interpreted broadly here to refer to the use of logical relations to describe semantic properties that follow from syntactic restrictions in the language. This includes not just System F's parametric polymorphism [130], but also many other semantic properties like sequentiality in PCF [141], capability safety in object capability languages [142, 152] or purity in dependently-typed languages with effects [126].

5.10 Conclusion

We have explored a fundamentally new approach for the dynamic checking of separation logic contracts. Our approach relies on hardware support for linear capabilities, a form of unforgeable and non-copyable memory pointers. A proof-directed compiler represents separation logic memory resources as linear capabilities and relies on the information in the proof to compile source code pointer dereferences to dereferences of the correct linear capability. We formalized and proved the correctness of our approach by showing that our compiler from verified source code to unverified target code is fully abstract.

Acknowledgements

This research is partially funded by the Research Fund KU Leuven, by the Research Foundation - Flanders (FWO) under grant number G0G0519N and by the Air Force Office of Scientific Research under award number FA9550-21-1-0054. Thomas Van Strydonck holds a PhD Fellowship of the Research Foundation - Flanders (FWO).

Chapter 6

Future Work and Conclusions

I believe that capabilities represent a next step in the security arms race between attackers and defenders, and will raise the security bar at the lowest levels of abstraction. They complement the advantages of process-based isolation, and have the potential to relieve the process abstraction of its role as a clunky security primitive in applications where more fine-grained security is desired. In a world where code bases increase in size and software interaction and networking consistently become more important, protection primitives that provide fine-grained compartmentalization and spatial memory safety at a low performance overhead steadily grow more appealing. The timing seems right, and the CHERI machine offers a seemingly low-overhead implementation of hardware capabilities, avoiding the complexity and inefficiency issues that prevented adoption of capability architectures in the 70s [181].

That being said, the practical applicability of capabilities remains to be thoroughly tested. The CHERI project has already contributed an impressive software stack as part of the research endeavors, but industry has not been able to experiment with capabilities yet. Hence, the interest from multiple hardware vendors, led by Arm and their Morello project, is a big step forward towards the validation and (hopefully) adoption of capabilities. These projects have the potential to break the cycle of chicken and egg that plagues hardware primitives and tooling based on these new primitives: by providing a hardware platform to code for and by involving software giants in the project, capabilities will receive the shot at validation they missed out on historically.

Along with the practical adoption, capabilities present many exciting opportunities for formal research, a fact that this PhD thesis aims to exemplify. The explicit representation of authority makes capabilities very flexible as a security primitive, but does render it all the more important and difficult to ensure that code manipulating them executes correctly and securely. We experienced this trade-off first hand while designing the secure interrupts and the local attestation mechanism in Chapter 4, and while developing code to use these features. In conclusion, capabilities are a prime candidate for formalization of security guarantees.

Within this formal realm, the universal contract approach we demonstrated seems particularly promising, since it permits us to reason about both trusted and untrusted code, and their interaction across trust domains. This allows deriving whole-system guarantees while only verifying a small, security-critical part of the system, as Chapter 3 illustrated. If performed on a real-world capability architecture, this would immediately result in hard guarantees that only require minimal verification effort. Through secure compilation techniques, the level at which the programming and reasoning occurs can be lifted from tedious, low-level assembly to more abstract source languages. The end goal would be to obtain whole-system security guarantees for a realistic capability architecture, while only requiring verification of a smaller piece of relatively high-level security-critical source code.

In the rest of this section, we summarize our accomplishments in these two research directions that were also highlighted in the introduction: formal reasoning about capability machines and secure compilation to capability back-ends. We take a step back to draw conclusions for each, and discuss the future prospects of the work. We do not repeat conclusions for individual chapters, and attempt to focus on the bigger picture. Section 6.1 concludes the work on formal reasoning about capability machines, and focuses on scaling universal contracts to more realistic architectures with complex features. Special attention goes to formal reasoning about enclaved execution, as this ongoing work aims to formally underpin the work presented in Chapter 4. Section 6.2 then discusses the chapter on secure compilation to capability architectures, and ponders the types of source languages one might want to compile securely to capability machines. Finally, Section 6.3 provides some high-level perspectives as a parting conclusion to the thesis.

6.1 Formal reasoning about capability machines

Chapters 2 and 3 illustrated the Cerise framework for formal reasoning about capability architectures, and showed how it can be extended to accommodate new features (MMIO in this case) with relatively limited effort. Chapter 4 introduced the conceptual building blocks to implement enclaved execution on a capability architecture, but did not yet formalize the security arguments that were presented. Section 6.1.1 below will discuss the addition of enclaved execution to the Cerise model in more detail. In other work I was involved in that was not included in this thesis, Georges et al. [62] have also used an extension of Cerise to reason about stack safety, to wit, well-bracketed control flow and local state encapsulation.

When reasoning formally about capability code, we are both interested in developing a program logic to reason about concrete code, as well as developing a methodology for reasoning about unknown, untrusted code. To do the latter, the Cerise model and its extensions make heavy use of the previously described universal contract methodology: a security contract for untrusted code is defined, that semantically captures an upper limit of the behavior that untrusted code might exhibit. As the name suggests, this contract holds over arbitrary untrusted code, and in the case of a capability machine, is essentially a specification of the capability safety guaranteed by the ISA.

Our current work has mostly explored *novel* formalizations of universal contracts that capture different notions of capability-based protection, and ways of reasoning about concrete code interacting with these contracts. To do this, concrete reasoning has to interface with the meta-theorems (such as the fundamental theorem) proven about the capability ISA. One thing we have not yet focused on is scaling this methodology; the end goal of this research is to extend the current approach to realistic (capability) architectures. We now expand upon the discussion from Section 2.9 and discuss three different directions of future research required to achieve this goal: proof automation, using more realistic models, and support for additional features. Extending the methodology to more abstract source languages is a fourth type of scaling, which is discussed in the next section.

Proof automation

Our models so far consisted of simple capability machines with concise operational semantics (on the order of dozens of instructions). In these models, performing most of the reasoning manually is feasible, though already laborious. That being so, we have developed a naive symbolic executor in Coq for the verification of straight-line (no jumps), concrete machine code. This cuts down on a lot of reasoning, but still requires the triples for single machine instructions to be derived manually. To scale our models to realistic ISAs, we will need *automation* both at the level of concrete (potentially non-straight-line) code, as well as in the derivation of triples for instructions. The goal of automation is to cut down on the tedious parts of reasoning, to allow focusing attention on the inherently difficult aspects that require human intervention.

Additionally, real-life ISAs undergo frequent changes, both in the design phase as e.g. Bauereiss et al. [18] report for the Morello project, as well as when instructions or features are added. Automation is hence necessitated by maintainability; it helps minimize the effort required to reprove formal results under such changes. As noted by Huyghebaert et al. [66], maintainable universal contracts could be used as specifications for the security guarantees offered by ISAs, as a supplement to the formal specification of the ISA semantics. This would improve over the state of the art, where security guarantees are usually specified in prose.

In the literature, multiple projects have already started investigating solutions to different facets of automated reasoning about ISAs. First, the aforementioned works of Nienhuis et al. [110] and Bauereiss et al. [18] illustrate different degrees of automation when proving intra-protection-domain monotonicity of capability authority on the full-scale Sail models (after extraction to Isabelle/HOL [111]) of respectively CHERI-MIPS (6k LoS) and ARM Morello (62k LoS in ASL, Arm's Architecture Specification Language, before extraction to Sail). In both cases, their proofs make use of an abstract semantics, which defines traces of interactions with memory and registers that satisfy some security properties. Each instruction is proven to satisfy such an abstract trace, and the security properties that hold over the abstract trace are in turn proven to imply capability monotonicity, completing the proof. Second, the Katamaran tool defines a framework for the generation of sound symbolic-execution-based verifiers [84, 83]. This methodology has been instantiated to Sail[1], and its automation is illustrated on a simple capability machine ISA called MinimalCaps [66]. Although this approach is yet to be illustrated on a full-scale ISA, it shows promise both for the verification of capability safety (in the style of the fundamental theorems shown in this thesis) as demonstrated by MinimalCaps, as well as for the verification of concrete code. Lastly, Islaris is a tool that combines the Isla symbolic executor for Sail [15] with automated reasoning about execution traces in Iris to verify concrete Armv8-A and RISC-V machine code running on top of the full Sail model, including system features [137].

Using more realistic models

All of these approaches aim for maximally realistic models: they avoid defining new, bespoke models for the operational semantics, since these would take a lot of time to craft, and because their validity with respect to the actual processor specification cannot easily be established. Instead, they start from complete, *authoritative* models defined in the Sail language, that have themselves been validated against the architectures they model. For example, Bauereiss et al. [18] report that the ASL-to-Sail translation of the Morello model was validated by comparing Sail's generated C simulator to an internal Arm test suite. Conversely, Isla was used to generate additional test cases from the Sail specification automatically. As another example, the CHERI-RISC-V model internally makes use of a RISC-V model that has been adopted by the RISC-V Foundation as the official, formal RISC-V specification that other implementations are compared against [133]. Additionally, the emulators resulting from these Sail models have been used to successfully boot various operating systems, as a practical form of validation [14]. It would hence make sense for the Cerise

[1] At the moment, specifications are written in μSail, a deep embedding in Coq of a subset of Sail, but the goal is to perform automatic extraction from Sail in the future.

approach to be applied to one of the authoritative Sail models in the future, possibly through one of the automation methodologies outlined above.

Supporting missing features

As a last future work track, we would like to explore the formalization of missing features in the Cerise model. Some of the real-world features that we have not explored are virtual memory and address translation, Direct Memory Access (DMA), interrupts and concurrency (including different memory models). Simply trying to implement some of these features in a capability setting is already challenging on its own; for example, the CHERI ISA spec has not yet settled on the best way to combine capabilities and DMA while staying true to the capability philosophy, and the same holds true for capabilities and virtual memory [175]. Once we decide on a satisfactory capability-style design, we can attempt to formally integrate these features into the Cerise model. As part of this last track, we are currently exploring different ways to model the enclaved execution design from Chapter 4 in the formalism of Cerise. The following subsection discusses the high-level concepts underlying this work.

In parallel, we would also like to test the applicability of the Cerise methodology to different architectures and security primitives. For example, in ongoing work Sander Huyghebaert and collaborators are formalizing the guarantees of RISC-V's PMP (Physical Memory Protection) feature as a universal contract, and using Katamaran to prove the fundamental theorem.

6.1.1 A formal account of enclaved execution

The Cerise work allowed reasoning about spatial memory safety and compartmentalization, but in order to reason about enclaved execution, we need to extend it with support for secure communication, attestation (both local and remote) and temporal memory safety. The latter is required in order to guarantee unique ownership of the enclave's memory footprint at the start of execution. The following subsections sketch the high-level ideas behind all three required extensions.

Secure communication

Sealed capabilities can be added to Cerise to symbolically encrypt local communication. Since encryption is inherently symbolic, we do not need to make any simplifying assumptions about the cryptographic primitives involved in local attestation. Hence, the unknown code running on the capability machine automatically corresponds to a standard Dolev-Yao attacker [49]. To model sealed capabilities, we draw inspiration from the work of Sumii and Pierce [150] on binary logical relations for a cryptographic lambda calculus. They defined a *relational environment* φ; a mapping from individual keys (otypes in our case) to relations over sealed values. All values sealed with a key k are required to be related by $\varphi(k)$, ensuring that valid functions cannot unseal non-related values.

We consider the unary case first, but extension to the binary case should be reasonably straightforward. In our case, each individual otype can be linked to a predicate $P : \mathrm{Cap} \rightarrow \mathit{iProp}$ (with *iProp* the universe of Iris propositions) over machine words, that specifies what words can be sealed with that specific seal. Consequently, any capabilities that are sealed with this seal and later unsealed, are required to uphold P. For example, assume that for an enclave's signing seal o_sign it holds that $P_{\mathrm{sign}} \triangleq \lambda c.\, c = (p, b, b+1, b) * \boxed{\exists n.\, b \mapsto 2 \cdot n}$, i.e. all signed capabilities point to a single, even integer value in their range. Then, receiving a capability signed with o_sign allows one to derive that its single-value contents must be even. A notion of causality between messages can be embedded into these predicates by using e.g. nonces. To link o_sign to the correct enclave's P_{sign}, reasoning about attestation will be required, as we discuss in the next subsection.

To encrypt remote communication, more standard cryptographic primitives must be added to Cerise. Since the remote case was not yet considered in Chapter 4, the design has to first be extended to allow linking ownership of the private parts of otypes to access to cryptographic keys for remote communication. Once this has been achieved, similar predicates can be defined that constrain the values encrypted or signed by specific cryptographic keys. Assumptions will need to be made to map the cryptographic primitives onto a symbolic model of cryptography, or alternatively, a probabilistic model in the style of Abadi and Plotkin [2] will need to be considered. The MMIO-infrastructure developed in Chapter 3 can likely be reused to verify that messages that are signed with a specific cryptographic key and sent onto the network, satisfy a certain desired protocol. The fact that properties can be enforced over observable effects makes the remote case ultimately more useful than the local one, but both cases seem to be similar on a technical level.

Attestation

From an abstract point of view, attestation enables the attesting party (either a local or remote entity) to know that the attested code will behave in a predictable, desired way during future interactions. As is often the case in the literature, there are two different methods to phrase this property: by explicitly stating what the set of desired behaviors is in terms of a protocol, and proving that the admitted behaviors satisfy the protocol, or by relating the behavior of the enclaved code to code that is known to admit correct behavior. This last technique is essentially a form of secure compilation.

Whichever approach we choose, to the best of our knowledge, no whole-system proofs about attestation have been realized at the level of the machine's semantics.

For the protocol-based approach, the otype predicates we described in the last section can be used as protocol specifications during local attestation. Protocols P_{sign} and P_{enc} can be attached to respectively the signing and encryption seal of an enclave. Local attestation then aims to verify that a given otype o indeed corresponds to the signing or encryption otype of an enclave of interest, such that P_{sign} and P_{enc} can be used to reason about capabilities sealed with o. This link can be made, because the attesting party knows the enclave identity for the enclave, and can use the `EStoreId` instruction to check whether o matches the expected identity. If a match is found and if the hash function is assumed to be collision-free, then the enclave must have been initialized correctly, and the predicates P_{sign} and P_{enc} can be used in reasoning. Of course, the enclave's concrete code will have to be proven to always uphold P_{sign} and P_{enc} for its signing and encryption seal, and the fundamental theorem will need to reason about the safety of enclaves that respect these predicates.

For the remote case, additional machinery is required: as mentioned in the last section, MMIO traces can be reused to specify enclave behavior. The rest of the reasoning should then be quite similar to the local case, where certain protocols can be connected to cryptographic keys for remote communication. All messages sent using this key need to satisfy the protocol. In other words, any code that has access to the cryptographic key (i.e. the code that has access to the private parts of the sealed capabilities) needs to be verified to uphold this protocol. Again, under the no-collision-assumption, this allows us to derive information about the correct initialization and future communication of an enclave, once remote attestation is successful.

We now briefly consider the secure compilation approach. In this space, Noorman et al. [115] have defined a property called *authentic execution* and provided a semi-formal proof sketch that their applications running on the Sancus TEE respect a unary version of this property. Concretely, they require the input-output behavior of enclaved, compiled MSP430 code to contextually refine the input-output behavior of the original C source code. All traces observed in practical execution should already have been observable while running the deployed code in a setting where no notion of enclaves or a network attacker exists. Their proof does consider replay and reordering attacks on messages sent between different enclaves, as part of a Dolev-Yao attacker model, but starts from the assumption that attestation has happened successfully and assumes a correct compiler. We would like to go one step further, and reason about the attestation process itself, and how one can reconstruct information about the executing code from successful attestation, as well as mechanize the whole proof.

To avoid the complexities of working with C code, and the orthogonal aspects involved in proving the compilation secure, another option is to define *overlay semantics* in the style of Skorstengaard et al. [144]; a second reinterpreted semantics for the same machine, that inherently provides the properties we want the secure compiler to preserve. This has the advantage that the source code stays close to the machine level, while the resulting compiler might still be reusable in a future secure compilation chain. In our case, the source language should offer *inherently secure* notions of enclaves, enclave identity and attestation.

Temporal memory safety

As mentioned in Chapter 4, at least two different approaches exist that guarantee unique ownership of memory capabilities at enclave initialization time: a memory sweep to find overlapping capabilities, and the use of linear (non-duplicable) capabilities. The literature provides inspiration on how to formalize the reasoning in either approach.

For the memory sweep, we draw inspiration from the work of Hur and Dreyer [65]. They have illustrated how one can reason about garbage collection in the context of logical relations by employing a *logical* memory layer on top of the physical memory, that is unaffected by garbage collection. In the logical memory, pointers are never deallocated or moved, preserving monotonicity of the logical relations. A mapping between logical and physical memory is tracked, along with a notion of satisfaction of a physical memory by a logical view of it. Under the assumption that the garbage collector respects this satisfaction relation, one can continue to reason about code using abstract, logical addresses, even in the presence of periodic garbage collector activations.

The idea would be to model our memory sweep in a similar fashion, where the mapping from logical to physical addresses is the identity (since we will not be moving capabilities in memory), except for a version number that tracks which logical version each physical address is currently at. Each logical address contains an additional *version* number to indicate what version it is, and an invariant is enforced that guarantees that only the current version will ever be reachable through separation logic resources for the logical registers. When a new enclave is initialized and the memory sweep finishes successfully, meaning that no aliases of the enclave's memory footprint are in circulation, then the version of the logical address is increased, and new logical separation logic resources can be created for said memory. Since we know the memory sweep was successful and the old physical memory was hence unreachable (apart from the copy used to initiate the memory sweep), this operation will respect the correspondence between logical and physical memory.

For the alternative approach with linear capabilities, we can draw inspiration from the StkTokens work of Skorstengaard et al. [144]. Although this development has not been mechanized, it provides a logical relation for a capability machine containing linear capabilities. It seems likely that modeling linear capabilities in the Cerise model would be more concise than the formalization of Skorstengaard et al., since linearity is easily modeled in a separation logic, and Iris helps avoid many of the technical details involved in the definition of Worlds. At a technical level, linear capabilities would cause the logical relation to no longer be persistent, since the $\mathcal{V}$ relation would then need to express non-duplicable ownership of linear capabilities.

6.2 Secure compilation

Although the formal reasoning as outlined in the previous section provides whole-system security guarantees over systems code or other security-critical code in the presence of untrusted adversaries, the fact that it requires code to be written and reasoned about at the assembly level is often impractical. In order to allow reasoning in a more abstract source language, researching secure compilation to capability architectures is required.

The backwards compatibility of existing source languages and their compilers is important to achieve general adoption in a top-down way. Conversely, the development of new, intermediate languages with increasingly high-level abstractions that provide bottom-up support for capabilities and fit this new back-end well is also an important goal. Another advantage to the latter approach is the possibility of proving compiler security for a relatively simple compiler, rather than a multi-pass, real-world project consisting of millions of LoC.

Chapter 3 could potentially be used as the basis for a source language in the bottom-up approach: since para-passthrough wrappers have real-world use in e.g. hypervisors [140], it would be useful to develop a more abstract language (with functions, variables and types rather than registers and capabilities) to develop these security wrappers in. This could provide programmers with immediate formal benefits, and constitute a first step towards general source-level abstractions that grant whole-system guarantees. Our goal here is similar to the whole-system correctness results that Erbsen et al. [52] demonstrate for a small, embedded RISC-V system and a compiler from Bedrock2, a minimal, C-like language, to RISC-V assembly. However, owing to the protection of capabilities and the universal contracts that model them, we would only require verification of the security-critical software components, not the whole code base. One difference with Erbsen et al. is that we do not explicitly verify the processor's implementation versus its formal specification, but rather, start from authoritative, formal ISA models and leave the CPU's verification out of scope, as a responsibility for the chip manufacturer.

Chapter 5 can be seen as an instantiation of the broader goal of secure compilation, with separation-logic-verified C-like code as a source language, where we prove that the source language's separation logic resources can be represented fully abstractly by linear capabilities in the target language. Section 5.8 already contained an extensive conclusion discussing the compiler.

As an additional remark, we note that the secure compiler from Chapter 5 can likely be split into two separate compiler passes. A first pass would enforce the functional aspects of the separation logic contracts at trust boundaries and compile from our verified C-like language to an intermediate C-like language with capabilities where untrusted components can only have a universal separation logic contract. Conceptually, this is similar to Swasey et al. [152]'s lowval predicate, or to a typed version of Sammler et al. [136]'s sandboxes, where sandboxed untrusted code is only allowed to interact with trusted code at the **any** type (a sort of unitype containing all values). A second pass would simply erase the contracts to arrive at the original target language.

Carefully performing this separation of passes might have resulted in a simpler and more reusable compiler security proof. Notably, since the first compiler pass involves both a verified source and target, if we were to mechanize our development as discussed in Section 5.8.3, then adequacy of the separation logic might help simplify the proofs. Additionally, it might be possible to formalize the simulation relations for the first pass as contextual refinements in the style of ReLoC [58], such that correctness and security can be proven within the logic itself.

6.3 Conclusion

In an ideal scenario, the Morello project is highly successful and proves once and for all that capabilities are viable security primitives, both in terms of cost and as a programming model. Capability architectures are widely adopted by different vendors, and integrated into a variety of existing architectures. In this case, there are multiple grand opportunities for formal research. First, as we described before, the security features of these novel architectures need to be formalized. Researchers need to investigate the best way to model these features and verify code that uses them, as well as ways to scale the size of the modeled architectures and codebases. The former was the subject of Chapters 2 to 4, whereas the latter was briefly discussed in Section 6.1.

Secondly, research needs to be conducted into compilers from existing and novel source languages to these new back-ends, and into ways of proving them secure. Historically, CHERI has focused on backwards compatibility with C and C++ code bases, because large bodies of code (especially systems code) have been written in these languages, and they are considered archetypal unsafe languages. The abundance of C/C++ systems code is one of the reasons CHERI-support was introduced into the

Clang compiler first. For example, the FreeBSD operating system that now runs on top of CHERI, was written in C. This is one of the reasons we studied secure compilation from verified C-like code in Chapter 5.

Other, safe, languages are being considered as compiler source languages as well. Notably, a capability-style extension of WebAssembly has been proposed in order to enforce memory safety within sandboxes [47] and Wei Sheng Sim [177] adapted the Rust-LLVM compiler to work with a CHERI back-end as part of their master's thesis. Alternatively, as we hinted at in Section 6.2, building bottom-up abstractions on top of the existing back-end is a viable alternative to more easily achieve verifiable whole-system guarantees for a language that allows, for example, the development of security wrappers. These whole-system guarantees for relatively low-level languages would be especially interesting for the development of systems and operating system software, as well as security-critical applications.

Even if the results of the Morello project turn out to be less unanimously positive, there is merit to this work. The general universal contract methodology as described in Section 1.2 applies at different levels of abstraction (higher-level examples are e.g. the OCPL logic [152], or applications of parametricity) and for different ISAs and security primitives (e.g. RISC-V PMP, virtual memory protection). Additionally, capabilities need not be implemented at the hardware level, as the OCPL logic and many other existing capability systems and languages illustrate. I therefore believe that many of the insights gathered in this thesis could be applied in these other settings as well.

Bibliography

[1] Martín Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, 1999.

[2] Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.*, 15(2):8:1–8:29, 2012.

[3] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. When good components go bad: formally secure compilation despite dynamic compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, 1351–1368. Association for Computing Machinery, 2018.

[4] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: exploring robust property preservation for secure compilation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 256–25615, 2019.

[5] Carmine Abate, Matteo Busi, and Stelios Tsampas. Fully abstract and robust compilation: and how to reconcile the two, abstractly. In Hakjoo Oh, editor, *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Proceedings*, volume 13008 of *Lecture Notes in Computer Science*, pages 83–101. Springer, 2021.

[6] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 334–344. IEEE Computer Society, 1998.

[7] Pieter Agten, Bart Jacobs, and Frank Piessens. Sound modular verification of C code executing in an unverified context. In *Symposium on Principles of Programming Languages*, POPL '15, pages 581–594. ACM, 2015.

[8] Tiago Alves and Don Felton. TrustZone: integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.

[9] AMD. AMD SEV-SNP: strengthening VM isolation with integrity protection and more. *White paper*, Jan. 2020.

[10] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[11] Arm Limited. Arm architecture reference manual supplement - morello for a-profile architecture. `https://developer.arm.com/documentation/ddi0606/latest`, 2020. [Online; accessed 14-02-2022].

[12] Arm Limited. ARM morello program. `https://www.arm.com/architecture/cpu/morello`, 2019. [Online; accessed 21-02-2022].

[13] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Jon French, Kathryn E. Gray, Gabriel Kerneis, Neel Krishnaswami, Prashanth Mundkur, Robert Norton-Wright, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. The Sail instruction-set architecture (ISA) specification language, 2013–2019.

[14] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages*, 3(POPL):71:1–71:31, Jan. 2019.

[15] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: integrating full-scale isa semantics and axiomatic concurrency models. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 303–316. Springer International Publishing, 2021.

[16] Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science. Springer International Publishing, 2018.

[17] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: static driver verification with under 4% false alarms. In Roderick Bloem and Natasha Sharygina, editors, *International Conference on Formal Methods in Computer-Aided Design*, pages 35–42. IEEE, 2010.

[18] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. Verified security for the morello capability-enhanced prototype arm architecture. In *Programming Languages and Systems*, pages 174–203. Springer International Publishing, 2022.

[19] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: design and implementation of nested virtualization. In *OSDI*. USENIX Association, Oct. 2010.

[20] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *International Conference on Computer Aided Verification*, pages 178–192. Springer, 2007.

[21] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*. International Symposium on Formal Methods for Components and Objects, Lecture Notes in Computer Science, pages 115–137. Springer, Berlin, Heidelberg, 2005.

[22] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAyer: memory safety for systems-level code. In *Computer Aided Verification*, pages 178–183. Springer, 2011.

[23] Bluespec. Bluespec company. https://bluespec.com/, 2003. [Online; accessed 20-07-2021].

[24] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. A multipurpose formal RISC-V specification, Apr. 2021. arXiv: 2104.00762 [cs].

[25] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: tiny trust anchor for tiny devices. In *Design Automation Conference*, DAC '15, pages 1–6. Association for Computing Machinery, June 2015.

[26] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing trustzone with user-space enclaves. In *NDSS*, 2019.

[27] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[28] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327. ACM, 1994.

[29] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, pages 431–447. ACM, 2016.

[30] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. *J. Autom. Reason.*, 61(1-4):141–189, 2018.

[31] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. Driverguard: virtualization-based fine-grained protection on I/O flows. *ACM Trans. Inf. Syst. Secur.*, 16(2):6:1–6:30, 2013.

[32] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N. M. Watson. CHERI JNI: Sinking the Java Security Model into the C. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 569–583. ACM, 2017.

[33] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: architectural support for a memory-safe C abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 117–130, 2015.

[34] Adam Chlipala. Formal reasoning about programs. 2017. URL: http://adam.chlipala.net/frap.

[35] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 1–12. Association for Computing Machinery, 2015.

[36] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[37] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[38] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: minimal hardware extensions for strong software isolation. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16*, pages 857–874. USENIX Association, 2016.

[39] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. Cheriabi: enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural*

Support for Programming Languages and Operating Systems, ASPLOS '19, 379–393. Association for Computing Machinery, 2019.

[40] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. *Journal of Computer Security*, 24(6):689–734, 2016.

[41] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy, SP 2015*, pages 813–830. IEEE Computer Society, 2015.

[42] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3):143–155, Mar. 1966.

[43] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. *SIGPLAN Not.*, 43(3):103–114, 2008.

[44] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016*, pages 147–162. IEEE, 2016.

[45] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *Symposium on Principles of Programming Languages, POPL 2016*, pages 164–177, 2016.

[46] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. Oh lord, please don't let contracts be misunderstood (functional pearl). In *International Conference on Functional Programming, ICFP 2016*, pages 117–131, 2016.

[47] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19. Association for Computing Machinery, 2019.

[48] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[49] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[50] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012.

[51] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. CapablePtrs: Securely compiling partial programs using the pointers-as-capabilities principle. In *34th IEEE Computer Security Foundations Symposium, CSF 2021*, pages 1–16. IEEE, 2021.

[52] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, 604–619. Association for Computing Machinery, 2021.

[53] Lawrence Esswood. *CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor.* PhD thesis, University of Cambridge, 2020.

[54] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305. ACM, 2017.

[55] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal safety for CHERI heaps. In *IEEE Symposium on Security and Privacy*. IEEE, May 2020.

[56] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In Karin Petersen and Willy Zwaenepoel, editors, *USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151. ACM, 1996.

[57] Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *Symposium on Principles of Programming Languages, POPL '13*, pages 371–384, 2013.

[58] Dan Frumin, Robbert Krebbers, and Lars Birkedal. ReLoC: a mechanised relational logic for fine-grained concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 442–451, 2018.

[59] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Principles of Programming Languages*, pages 429–442. ACM, 2016.

[60] Deepak Garg, Catalin Hritcu, Marco Patrignani, Marco Stronati, and David Swasey. Robust hyperproperty preservation for secure compilation (extended abstract), 2017. arXiv: 1710.07309.

[61] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu. French. In *Journées Francophones des Langages Applicatifs 2021*. Institut de Recherche en Informatique Fondamentale, Apr. 2021.

[62] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proceedings of the ACM on Programming Languages*, 5(POPL):6:1–6:30, Jan. 2021.

[63] Jean-Yves Girard. Une extension de l'interpretation de gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *Studies in Logic and the Foundations of Mathematics*. Volume 63, pages 63–92. Elsevier, 1971.

[64] LLVM Developer Group. Clang: a c language family frontend for llvm. https://clang.llvm.org/, 2007. [Online; accessed 18-02-2022].

[65] Chung-Kil Hur and Derek Dreyer. A kripke logical relation between ml and assembly. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, 133–146. Association for Computing Machinery, 2011.

[66] Sander Huyghebaert, Steven Keuchel, and Dominique Devriese. Semi-automatic verification of isa security guarantees in the form of universal contracts. *Workshop on the Security of Software/Hardware Interfaces (SILM)*, 2021.

[67] Sander Huyghebaert, Thomas Van Strydonck, Steven Keuchel, and Dominique Devriese. Uninitialized capabilities. arXiv: 2006.01608 [cs].

[68] Gianluca Insolvibile. Garbage collection in C programs. *Linux J.*, 2003(113):7, Sept. 2003.

[69] Iris Team. The Iris documentation and Coq development. 2021. URL: https://iris-project.org.

[70] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, 271–282. Association for Computing Machinery, 2011.

[71] Bart Jacobs and Frank Piessens. The VeriFast program verifier. *CW Reports*, 2008.

[72] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *Programming Languages and Systems*. Volume 6461, Lecture Notes in Computer Science, pages 304–311. Springer Berlin Heidelberg, 2010.

[73] Limin Jia, Shayak Sen, Deepak Garg, and Anupam Datta. A logic of programs with interface-confined code. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 512–525, 2015.

[74] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. Efficient Tagged Memory. In *IEEE International Conference on Computer Design (ICCD)*. IEEE, Nov. 2017.

[75] Nicolas Joly, Saif ElSherei, and Saar Amar. Security analysis of CHERI ISA. `https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf`, 2020. [Online; accessed 15-02-2022].

[76] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo De Amorim, Boris Eng, and Benjamin C. Pierce. Beyond good and evil: formalizing the security guarantees of compartmentalizing compilation. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 45–60, 2016.

[77] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for rust. *Proc. ACM Program. Lang.*, 4(POPL), 2019.

[78] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, Dec. 2017.

[79] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *International Conference on Functional Programming*, ICFP 2016, 256––269. Association for Computing Machinery, 2016.

[80] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

[81] Bernhard Kauer, Paulo Veríssimo, and Alysson Neves Bessani. Recursive virtual machines for advanced security mechanisms. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, pages 117–122. IEEE Computer Society, 2011.

[82] Paul Kehrer. 2021 in memory unsafety - Apple's operating systems. `https://langui.sh/2021/12/13/apple-memory-safety/`, 2021. [Online; accessed 15-02-2022].

[83] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. Katamaran project. https://katamaran-project.github.io/, 2020. [Online; accessed 22-03-2022].

[84] Steven Keuchel, Georgy Lukyanov, and Dominique Devriese. Katamaran: semi-automated verification of ISA specifications. June 2020. URL: https://pldi20.sigplan.org/details/rems-deepspec-2020/7/Katamaran-semi-automated-verification-of-ISA-specifications.

[85] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[86] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *ACM Symposium on Operating Systems Principles 2009*, pages 207–220. ACM, 2009.

[87] Thomas F. Knight, Jr., André DeHon, Andrew Sutherland, Udit Dhawan, Albert Kwon, and Sumit Ray. SAFE ISA (version 3.0 with interrupts per thread), 2012.

[88] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: a security architecture for tiny embedded devices. In *European Conference on Computer Systems*, EuroSys '14, pages 1–14. ACM, Apr. 2014.

[89] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 696–723. Springer Berlin Heidelberg, 2017.

[90] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020*, 38:1–38:16. ACM, 2020.

[91] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. Cryptographic capability computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, 253–267. Association for Computing Machinery, 2021.

[92] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, 42–54. Association for Computing Machinery, 2006.

[93] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE, Jan. 2016.

[94] Henry M. Levy. *Capability-Based Computer Systems*. en. Digital Press, 1984.

[95] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: a two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 409–420. USENIX Association, June 2014.

[96] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Krügel, and Giovanni Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *NDSS*, 2017.

[97] Donald MacKenzie and Garrel Pottinger. Mathematics, technology, and trust: formal verification, computer security, and the U.S. military. *IEEE Ann. Hist. Comput.*, 19(3):41–59, 1997.

[98] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. Hardware-based Trusted Computing Architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, Mar. 2018.

[99] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010*, pages 125–140. IEEE Computer Society, 2010.

[100] A. Theodore Markettos, John Baldwin, Ruslan Bukin, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Position paper: defending direct memory access with CHERI capabilities, 2020.

[101] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. Trustvisor: efficient TCB reduction and attestation. In *31st IEEE Symposium on Security and Privacy, S&P 2010*, pages 143–158. IEEE Computer Society, 2010.

[102] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for TCB minimization. In Joseph S. Sventek and Steven Hand, editors, *Proceedings of the 2008 EuroSys Conference*, pages 315–328. ACM, 2008.

[103] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[104] Microchip Technology Inc. SAM D5x/E5x Family Data Sheet. en, 2019.

[105] Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, 2006.

[106] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: hardware for safe and secure manual memory management and full memory safety. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 189–200, 2012.

[107] Peter G. Neumann. Fundamental Trustworthiness Principles in CHERI. en. In *New Solutions for Cybersecurity.* Jan. 2018.

[108] Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming, ICFP 2016*, pages 103–116, 2016.

[109] Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking for separation logic. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference*, pages 203–217, 2008.

[110] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*, May 2020.

[111] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[112] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: a simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and InHerItance*, pages 9–16, 2013.

[113] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. In *USENIX Security Symposium*, pages 479–494, 2013.

[114] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: a low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, July 2017.

[115] Job Noorman, Tobias Mühlberg, and Frank Piessens. Authentic execution of distributed event-driven applications with a small tcb. eng. In volume 10547, pages 55–71. Livraga, G, Springer; Heidelberg, DE, 2017.

[116] Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). In *Software Safety and Security - Tools for Analysis and Verification*, pages 286–318. 2012.

[117] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2), 2018.

[118] Charles Papon. SpinalHDL, A Scala based HDL. `https://github.com/SpinalHDL/SpinalHDL`, 2015. [Online; accessed 10-02-2022].

[119] Charles Papon. VexRiscv, A FPGA friendly 32 bit RISC-V CPU implementation. `https://github.com/SpinalHDL/VexRiscv`, 2017. [Online; accessed 10-02-2022].

[120] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *31st IEEE Symposium on Security and Privacy, S&P 2010*, pages 414–429. IEEE Computer Society, 2010.

[121] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37(2), Apr. 2015.

[122] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: a survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, Feb. 2019.

[123] Marco Patrignani, Dominique Devriese, and Frank Piessens. On modular and fully-abstract compilation. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 17–30, 2016.

[124] Marco Patrignani and Deepak Garg. Robustly safe compilation or, efficient, provably secure compilation, 2018. arXiv: `1804.00489`.

[125] Marco Patrignani and Deepak Garg. Secure compilation and hyperproperty preservation. In *Computer Security Foundations Symposium*, pages 392–404. IEEE, Aug. 2017.

[126] Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. A reasonably exceptional type theory. *Proceedings of the ACM on Programming Languages*, 3(ICFP):108:1–108:29, July 2019.

[127] Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens. Sound formal verification of linux's USB BP keyboard driver. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 210–215. Springer, 2012.

[128] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002. 656 pages. Google-Books-ID: ti6zoAC9Ph8C.

[129] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE, 2002.

[130] John C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing*, pages 513–523. North Holland, 1983.

[131] Alexander Richardson. *Complete Spatial Safety for C and C++ Using CHERI Capabilities*. en. PhD thesis, University of Cambridge, Computer Laboratory, 2020.

[132] Sail-CHERI-RISC-V, CHERI-RISC-V model in Sail. `https://github.com/CTSRD-CHERI/sail-cheri-riscv`, 2019. [Online; accessed 10-02-2022].

[133] Sail-RISC-V, RISC-V model in Sail. `https://github.com/CTSRD-CHERI/sail-riscv`, 2019. [Online; accessed 23-03-2022].

[134] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. Heapcheck: low-cost hardware support for memory safety. *ACM Trans. Archit. Code Optim.*, 19(1), 2022.

[135] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[136] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.*, 4(POPL):32:1–32:32, 2020.

[137] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: verification of machine code against authoritative ISA semantics. In *43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2022.

[138] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, 552–561. Association for Computing Machinery, 2007.

[139] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast capability system. *SIGOPS Oper. Syst. Rev.*, 33(5):170–185, 1999.

[140] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. BitVisor: a thin hypervisor for enforcing I/O device security. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09. ACM, Mar. 2009.

[141] Kurt Sieber. Reasoning about sequential functions via logical relations. *Applications of categories in computer science*, 177:258–269, 1992.

[142] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities - provably safe stack and return pointer management. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018*, pages 475–501, 2018.

[143] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. *ACM Transactions on Programming Languages and Systems*, 42(1):5:1–5:53, Dec. 2019.

[144] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.*, 3(POPL):19:1–19:28, Jan. 2019.

[145] Open source project. The freebsd project. https://www.freebsd.org/, 1993. [Online; accessed 18-02-2022].

[146] SpinalHDL, Cryptography libraries. https://github.com/SpinalHDL/SpinalCrypto, 2017. [Online; accessed 10-02-2022].

[147] Raoul Strackx and Frank Piessens. Fides: selectively hardening software application components against kernel-level or process-level malware. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12*, pages 2–13. ACM, 2012.

[148] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. en. In *Security and Privacy in Communication Networks*, Lecture Notes, pages 344–361. Springer, Berlin, Heidelberg, Sept. 2010.

[149] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*, pages 161–172. ACM, 2004.

[150] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. English. *Journal of Computer Security*, 11(4):521–554, 2003.

[151] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Higher-order concurrent abstract predicates. *Modular specification and verification for higher-order languages with state*:108, 2012.

[152] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA):89:1–89:26, Oct. 2017.

[153] Gavin Thomas. A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/, 2019. [Online; accessed 15-02-2022].

[154] Tools for BlueSpec HDL. https://github.com/B-Lang-org/bsc, 2020. [Online; accessed 10-02-2022].

[155] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *European Conference on Computer Systems*, pages 1–14. Association for Computing Machinery, Apr. 2014.

[156] Neline van Ginkel, Raoul Strackx, and Frank Piessens. Automatically generating secure wrappers for SGX enclaves from separation logic specifications. In Bor-Yuh Evan Chang, editor, *Programming Languages and Systems*, pages 105–123. Springer International Publishing, 2017.

[157] Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. Secure resource sharing for embedded protected module architectures. eng. In volume 9311, pages 71–87. Akram, RN, Springer, 2015.

[158] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, 1741–1758. Association for Computing Machinery, 2019.

[159] Thomas Van Strydonck, Dominique Devriese, and Frank Piessens. Linear capabilities for modular fully-abstract compilation of verified code. *Principles of Secure Compilation (PriSC)*, 2018.

[160] Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. Proving full-system security properties under multiple attacker models on capability machines. Accepted for publication at CSF22, 2022.

[161] Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. Proving full-system security properties under multiple attacker models on capability machines: coq mechanization, Sept. 2021.

[162] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, 3(ICFP), 2019.

[163] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Journal Of Functional Programming*, 31(PII S0956796821000022):1–55, Mar. 2021.

[164] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code - technical appendix including proofs and details. 2020. URL: https://soft.vub.ac.be/~dodevrie/seplogic-lincaps-tr20201130.pdf.

[165] Thijs Vercammen, Thomas Van Strydonck, and Dominique Devriese. Borrowed capabilities: flexibly enforcing revocation on a capability architecture. *Workshop on the Security of Software/Hardware Interfaces (SILM)*, 2021.

[166] Verilator, the fastest Verilog/SystemVerilog simulator. https://www.veripool.org/verilator/, 1994. [Online; accessed 10-02-2022].

[167] Simon Friis Vindum and Lars Birkedal. Contextual refinement of the michael-scott queue (proof pearl). In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 76–90. ACM, 2021.

[168] Frédéric Vogels, Bart Jacobs, and Frank Piessens. Featherweight VeriFast. *Logical Methods in Computer Science*, 11(3), 2015.

[169] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.

[170] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architecture for java. In Michel Banâtre, Henry M. Levy, and William M. Waite, editors, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 116–128. ACM, 1997.

[171] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: Base user-level ISA. Technical report, 2011.

[172] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: practical capabilities for UNIX. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[173] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter Neumann. Capability Hardware Enhanced RISC Instructions (CHERI). https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/, 2021. [Online; accessed 20-07-2021].

[174] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, Sept. 2019.

[175] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, Oct. 2020.

[176] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[177] Nicholas Wei Sheng Sim. *Strengthening memory safety in Rust: exploring CHERI capabilities for a safe language.* Master's thesis, University of Cambridge, 2020.

[178] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: tag-isolated memory bringing fine-grained enclaves to RISC-V. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019.* The Internet Society, 2019.

[179] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony C. J. Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. CHERI concentrate: practical compressed capabilities. *IEEE Trans. Computers*, 68(10):1455–1469, 2019.

[180] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[181] Jonathan D. Woodruff. CHERI: A RISC capability machine for practical memory safety. Technical report, University of Cambridge., 2014.

[182] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety. In *IEEE/ACM International Symposium on Microarchitecture.* ACM, Oct. 2019.

[183] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alexander Richardson, Simon W. Moore, and Robert N. M. Watson. CheriRTOS: A capability model for embedded devices. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 92–99, Oct. 2018.

[184] Xilinx. Vivado. https://www.xilinx.com/products/design-tools/vivado.html, 2012. [Online; accessed 13-07-2021].

[185] Xilinx. Zynq UltraScale+ MPSoC data sheet. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf, 2021. [Online; accessed 20-07-2021].

[186] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. Elasticlave: an efficient memory model for enclaves. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Aug. 2022.

List of Publications

Journal Papers

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Journal Of Functional Programming*, 31(PII S0956796821000022):1–55, Mar. 2021

Conference Papers

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, 3(ICFP), 2019

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proceedings of the ACM on Programming Languages*, 5(POPL):6:1–6:30, Jan. 2021

Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. Proving full-system security properties under multiple attacker models on capability machines. Accepted for publication at CSF22, 2022

Technical Reports

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code - technical appendix including proofs and details. 2020. URL: https://soft.vub.ac.be/~dodevrie/seplogic-lincaps-tr20201130.pdf

Sander Huyghebaert, Thomas Van Strydonck, Steven Keuchel, and Dominique Devriese. Uninitialized capabilities. arXiv: 2006.01608 [cs]

Workshop Papers

Thomas Van Strydonck, Dominique Devriese, and Frank Piessens. Linear capabilities for modular fully-abstract compilation of verified code. *Principles of Secure Compilation (PriSC)*, 2018

Thijs Vercammen, Thomas Van Strydonck, and Dominique Devriese. Borrowed capabilities: flexibly enforcing revocation on a capability architecture. *Workshop on the Security of Software/Hardware Interfaces (SILM)*, 2021

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu. French. In *Journées Francophones des Langages Applicatifs 2021*. Institut de Recherche en Informatique Fondamentale, Apr. 2021

In Submission

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Cerise: program verification on a capability machine in the presence of untrusted code. *In Submission*

Thomas Van Strydonck, Job Noorman, Leonardo Alves Dias, Jennifer Jackson, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. CHERI-TrEE: flexible enclaves on capability machines. *In Submission*

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
https://www.distrinet.cs.kuleuven.be