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

Thomas Van Strydonck
KU Leuven

Aïna Linn Georges
Aarhus University

Armaël Guéneau
Aarhus University

Alix Trieu
Aarhus University

Amin Timany
Aarhus University

Frank Piessens
KU Leuven

Lars Birkedal
Aarhus University

Dominique Devriese
KU Leuven, Vrije Universiteit Brussel

***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.**

I. 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 [1] and running virtual machines inside virtual machines requires additional context switches with additional overhead [2].

Contrary to many other primitives, micro-policies [3] and capability machines [4], [5], [6] 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 [7], [8], [9], Skorstengaard et al. and Georges et al. have developed a methodology for *robust modular verification* of software running on capability machines [10], [11] 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 [11], [12] or equi-termination of programs [13]. 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

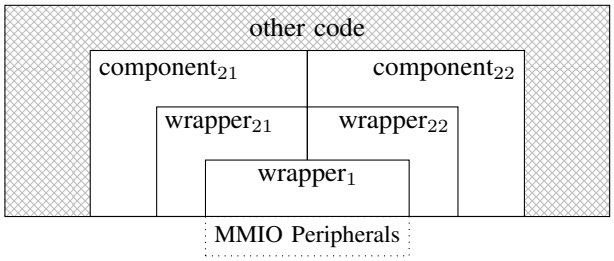


Figure 1: An example architecture of nested parapass-through wrappers around the peripherals.

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]

To make our approach concrete, we study the equivalent of BitVisor's parapass-through virtualization [14], 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 Fig. 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 [15]), if they were extended with capability machine security primitives. Fig. 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 Fig. 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 Fig. 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 Fig. 2c and Fig. 2d.

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 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. [11] with memory-mapped I/O and secure enforcement of I/O properties and reprove

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

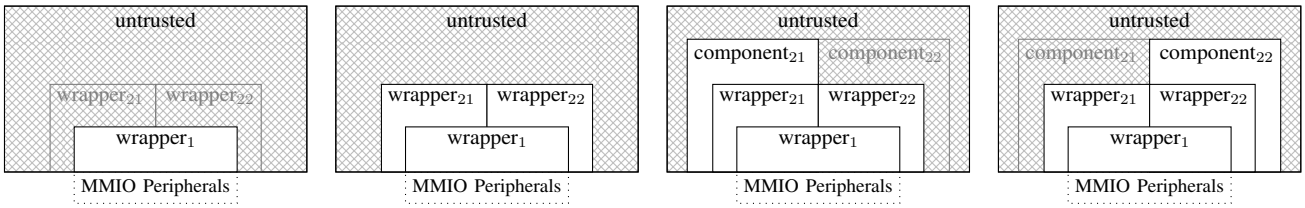


Figure 2: Four example attacker models, which could be used for analysing security of the system in Fig. 1.

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.

All our proofs have been machine-verified in Coq, using the Iris program logic, and are available online [16].

II. 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. [11] and, transitively, on that of Skorstengaard et al. [10], [12]. 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 II-A presents the operational semantics for a simple capability machine and Section II-B explains how we add memory-mapped I/O. The semantics is summarized in Fig. 3 to 6 with additions for memory-mapped I/O in blue.

A. A simple capability machine

The syntax of capability machine programs is given in Fig. 3. We consider a machine with finite memory bounded by $\mathrm{AddrMax}$. A machine word $w \in \mathrm{Word}$ 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. A permission is an element of the lattice depicted in Fig. 4. 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 [5], 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.

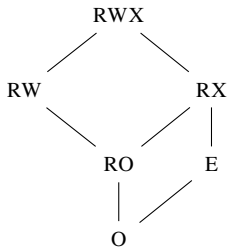


Figure 4: Permission lattice.

Fig. 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 $[\![decode(z)]\!](\varphi)$. Machine instructions i operate over registers r or either integers or registers ρ. The behavior of instruction i in configuration φ is specified by $[\![i]\!](\varphi)$ defined in Fig. 5. Most instructions use the auxiliary function updPC to increment 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.

$$
\begin{array}{llll}
a & \in \text{Addr} & \triangleq & [0, \text{AddrMax}] \\
p & \in \text{Perm} & \triangleq & \text{O} \mid \text{E} \mid \text{RO} \mid \text{RX} \mid \text{RW} \mid \text{RWX} \\
c & \in \text{Cap} & \triangleq & \{(p,b,e,a) \mid b,e,a \in \text{Addr}\} \\
w & \in \text{Word} & \triangleq & \mathbb{Z} + \text{Cap} \\
r & \in \text{RegName} & \triangleq & \mathsf{pc} \mid \mathsf{r}_0 \mid \mathsf{r}_1 \mid \ldots \mid \mathsf{r}_{31} \\
reg & \in \text{Reg} & \triangleq & \text{RegName} \to \text{Word} \\
m & \in \text{Mem} & \triangleq & \text{Addr} \to \text{Word}
\end{array}
\qquad
\begin{array}{llll}
 & \text{EventTy} & \triangleq & \text{IOWrite} \mid \text{IORead} \\
e & \in \text{Event} & \triangleq & \text{EventTy} \times \text{Addr} \times \mathbb{Z} \\
t & \in \text{Trace} & \triangleq & \text{list Event} \\
\varphi & \in \text{ExecConf} & \triangleq & \text{Reg} \times \text{Mem} \times \text{State} \times \text{Trace} \\
\delta & \in \text{DoneState} & \triangleq & \mathsf{Standby} \mid \mathsf{Halted} \mid \mathsf{Failed} \\
\mu & \in \text{ExecMode} & \triangleq & \mathsf{SingleStep} \mid \mathsf{Repeat}\ \mu \mid \mathsf{Done}\ \delta \\
\rho & \in \mathbb{Z} + \text{RegName} & &
\end{array}
$$

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

Figure 3: Machine words, machine state and instructions.

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

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

i	$[\![i]\!](\varphi)$	Conditions
`fail`	$(\mathsf{Done\ Failed}, \varphi)$	
`halt`	$(\mathsf{Done\ Halted}, \varphi)$	
`move` 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 $b \le a < e$ and $p \in \{\text{RO}, \text{RX}, \text{RW}, \text{RWX}\}$ and $a \notin$ `MMIO`
`load` r_1 r_2	$\text{updPC}(\text{updST}(\varphi[\text{reg}.r_1 \mapsto z], (\text{IORead}, a, z), s))$	$\varphi.\text{reg}(r_2) = (p,b,e,a)$ and $\text{mmioLoad}(\varphi.\text{state}, a) = (s,z)$ and $b \le a < e$ and $p \in \{\text{RO}, \text{RX}, \text{RW}, \text{RWX}\}$ and $a \in$ `MMIO`
`store` r ρ	$\text{updPC}(\varphi[\text{mem}.a \mapsto w])$	$\varphi.\text{reg}(r) = (p,b,e,a)$ and $b \le a < e$ and $p \in \{\text{RW}, \text{RWX}\}$ and $w = \text{getWord}(\varphi, \rho)$ and $a \notin$ `MMIO`
`store` r ρ	$\text{updPC}(\text{updST}(\varphi, (\text{IOWrite}, a, z), \text{mmioStore}(\varphi.\text{state}, a, z)))$	$\varphi.\text{reg}(r) = (p,b,e,a)$ and $b \le a < e$ and $p \in \{\text{RW}, \text{RWX}\}$ and $z = \text{getWord}(\varphi, \rho)$ and $z \in \mathbb{Z}$ and $a \in$ `MMIO`
`jmp` r	$(\mathsf{Done\ Standby}, \varphi[\text{reg.pc} \mapsto newPc])$	if $\varphi.\text{reg}(r) = (\text{E}, b, e, a)$, then $newPc = (\text{RX}, b, e, a)$ otherwise $newPc = \varphi.\text{reg}(r)$
`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 $b \le z_1$ and $0 \le z_2 \le e$ and $p \ne \text{E}$ 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 $p \ne \text{E}$ and $w = (p,b,e,a+z)$
`geta` r_1 r_2	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto a])$	$\varphi.\text{reg}(r_2) = (\_,\_,\_,a)$
...		
_	$(\mathsf{Done\ Failed}, \varphi)$	otherwise

Figure 5: Operational semantics: instruction semantics.

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

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

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

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

$$
(\mathsf{SingleStep}, \varphi) \to \begin{cases} [\![decode(z)]\!](\varphi) & \text{if } \varphi.\text{reg}(\mathsf{pc}) = (p,b,e,a) \wedge b \le a < e \wedge p \in \{\text{RX}, \text{RWX}\} \wedge \varphi.\text{mem}(a) = z \\ (\mathsf{Done\ Failed}, \varphi) & \text{otherwise} \end{cases}\ \textsc{ExecSingle}
$$

Figure 6: Operational semantics: reduction steps.

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 [17]. 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 Fig. 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 Fig. 5.

B. 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 Fig. 3 and 5. For simplicity, we do not yet support interrupts; the CPU and devices must poll memory for updates. We discuss adding interrupts in Section VI.

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 Fig. 3. An event e is a triple of a mode $e.\mathrm{type}$ (read or write), an address $e.\mathrm{addr}$, and an integer $e.\mathrm{value}$ that is read or written. The operational semantics of our machine in Fig. 5 is parameterized by this set State, `MMIO` and two operations $\mathrm{mmioLoad} : (\mathrm{State} \times \mathrm{Addr}) \to (\mathrm{State} \times \mathbb{Z})$ and $\mathrm{mmioStore} : (\mathrm{State} \times \mathrm{Addr} \times \mathbb{Z}) \to \mathrm{State}$, that model how the state of the devices reacts to MMIO loads and stores.

Fig. 5 shows the new behavior of `load` and `store` for MMIO addresses. `load` will use $\mathrm{mmioLoad}$ to get the value at address a from the environment and load it into the register. Similarly, `store` uses $\mathrm{mmioStore}$ to indicate that a value is written at some address. In both cases, the operations transition the environment's state and an IORead or $\mathrm{IOWrite}$ event is recorded in the trace.

III. 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 Fig. 1 can set up the capability machine to achieve their security objectives.

A. Three-layer stateful example with orthogonal wrappers

The first system we consider is setup as in Fig. 1, with an additional bottom-most wrapper $\mathrm{wrapper}_0$. 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 IV-D. The different wrappers enforce the following concrete invariants, modelling the kind of real security properties we discussed in the introduction:

- $\mathrm{Wrapper}_0$ 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 Fig. 7. We 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.
- $\mathrm{Wrapper}_1$ 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 Fig. 7.
- To ensure safety of values sent to their respective peripherals, $\mathrm{wrapper}_{21}$ solely allows positive values, whereas $\mathrm{wrapper}_{22}$ 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 Fig. 7, where $\uparrow_e^t(P_e)$ 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. $\mathrm{Wrapper}_{21}$ only allows events destined for address a_1, i.e., $\mathrm{wrapper}_{21}$ enforces predicate P_{21} on a *filtered* view of the general MMIO trace. The filtering is represented by F_{21} in Fig. 7. Thus $\mathrm{wrapper}_{21}$ enforces $\mathrm{P}_{21} \circ \mathrm{F}_{21}$ on the MMIO trace. Analogously, $\mathrm{wrapper}_{22}$ 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 Fig. 8. Note that $\mathrm{wrapper}_0$ does not have any security objective of its own, since it solely encapsulates MMIO. OBJ-1 specifies the guarantees that $\mathrm{wrapper}_1$ hopes to achieve from verification. OBJ-1 states that if the initial memory m_0 and registers r_0 constitute a valid configuration $\mathrm{init\_config\_1}(r_0, m_0)$ (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, $\mathrm{wrapper}_1$ can be sure that P_1 will hold on any trace of execution, as long as the capability machine boots into a configuration satisfying $\mathrm{init\_config\_1}$.

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 $\overline{\mathrm{F}}_2$

$$
\begin{aligned}
c\_ls_i &\triangleq (\text{RWX}, d\_ls_i, d\_e_i, d\_ls_i)\\
c\_r_i &\triangleq (\text{E}, d\_r_i, d\_e_i, d\_r_i)\\
c\_w_i &\triangleq (\text{E}, d\_e_i, d\_e_i, d\_w_i)
\end{aligned}
\qquad
\begin{aligned}
\uparrow_e^t &: (\text{Event} \to \text{Prop}) \to (\text{Trace} \to \text{Prop}) \triangleq \lambda P_{\text{e}}\ t.\ (\forall e.\ e \in t \Rightarrow P_{\text{e}}(e))\\
\text{F}_{21} &: \text{Trace} \to \text{Trace} \triangleq \lambda t.\ \text{filter}(\lambda e.\ e.\text{addr} = a_1, t)\\
\text{F}_{22} &: \text{Trace} \to \text{Trace} \triangleq \lambda t.\ \text{filter}(\lambda e.\ e.\text{addr} = a_2, t)\\
\overline{\text{F}}_2 &: \text{Trace} \to \text{Trace} \triangleq \lambda t.\ \text{filter}(\lambda e.\ e.\text{addr} \notin \{a_1, a_2\}, t)
\end{aligned}
$$

$$
\begin{aligned}
\text{P}_0 &: \text{Trace} \to \text{Prop} \triangleq \lambda\_.\ \text{True}\\
\text{P}_1 &: \text{Trace} \to \text{Prop} \triangleq \lambda t.\ \text{length}(t) < 1000\\
\text{P}_{21} &: \text{Trace} \to \text{Prop} \triangleq \lambda t.\ \uparrow_e^t (\lambda e, e.\text{value} > 0)\\
\text{P}_{22} &: \text{Trace} \to \text{Prop} \triangleq \lambda t.\ \uparrow_e^t (\lambda e, e.\text{value} < 0)
\end{aligned}
\qquad
\begin{aligned}
&\text{LS\_gen}_x : \text{list Word} \to \text{list Word} \triangleq\\
&\qquad \lambda \overline{v_{\text{cust}}}.\ [c\_ls_x, c\_r_{\text{pr}(x)}, c\_w_{\text{pr}(x)}] +\!\!+ \overline{v_{\text{cust}}}\\
&\text{LS}_0 : \text{Trace} \to \text{list Word} \triangleq \lambda\_.\ [(\text{RW}, \texttt{MMIO}_b, \texttt{MMIO}_e, \texttt{MMIO}_b)]\\
&\text{LS}_1 : \text{Trace} \to \text{list Word} \triangleq \lambda t.\ \text{LS\_gen}_1([\text{length}(t)])\\
&\text{LS}_{21} : \text{Trace} \to \text{list Word} \triangleq \lambda\_.\ \text{LS\_gen}_{21}([\,])\\
&\text{LS}_{22} : \text{Trace} \to \text{list Word} \triangleq \lambda\_.\ \text{LS\_gen}_{22}([\,])
\end{aligned}
$$

Figure 7: Definitions involved in the first example

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

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

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

Figure 8: The different security objectives that the stakeholders wish to enforce in the first example

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 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; Fig. 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 Fig. 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 Fig. 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, Fig. 9a presents the memory layout from the point of view of wrapper$_0$. As 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 $\text{LS}_0(\_)$ in Fig. 7. This capability provides the read and write closures of wrapper$_0$ access to all of MMIO. Fig. 11 demonstrates the code for the write closure. It makes use of the following macros:

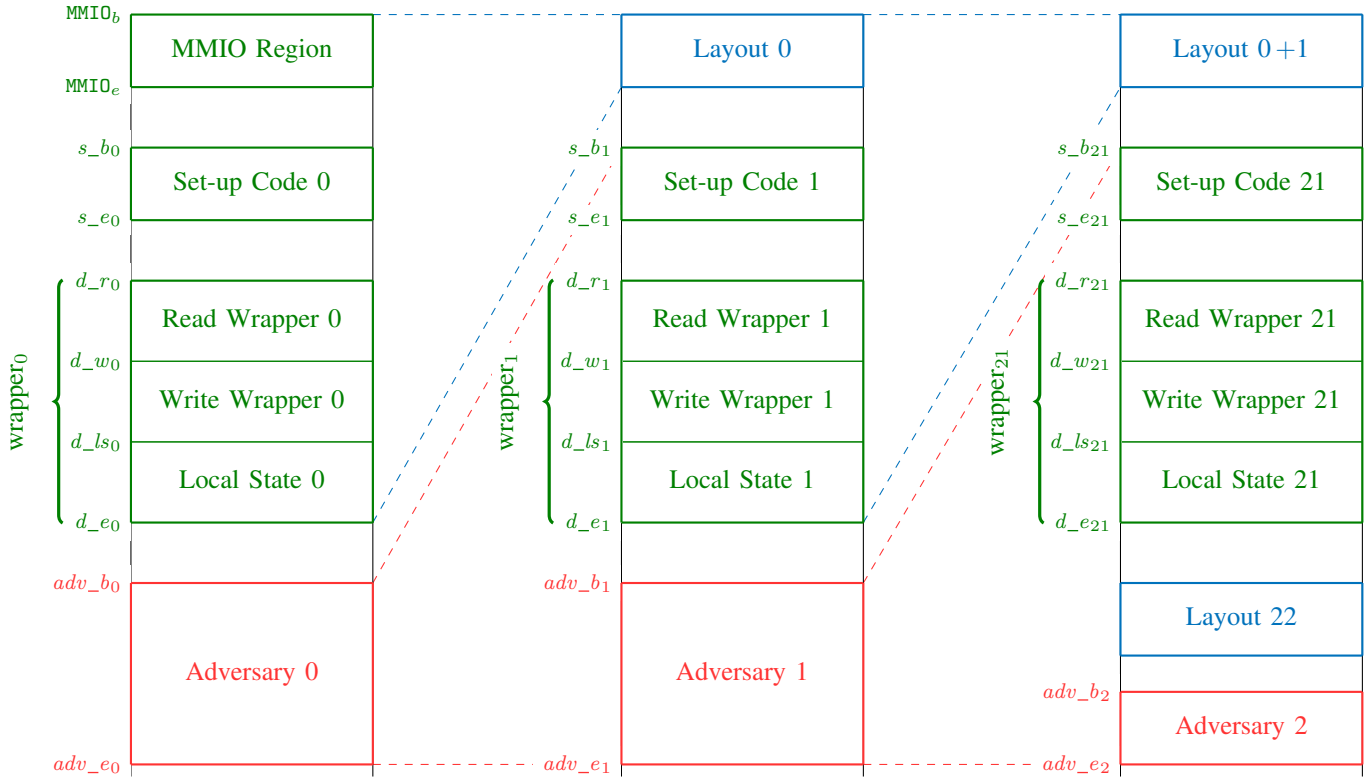
- `reqint` r r_{aux}: succeeds iff r contains an integer
- `lea_a` r ρ r_{aux}: absolute version of `lea`, that makes the capability in r point to the address corresponding to ρ.
- `rclear` $\overline{r}$: clears the set of registers $\overline{r}$ by overwriting them with the value 0.

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}

Fig. 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.

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. Fig. 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



(a) wrapper$_0$ layout (b) wrapper$_1$ layout (c) wrapper$_{21}$ layout

Figure 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.

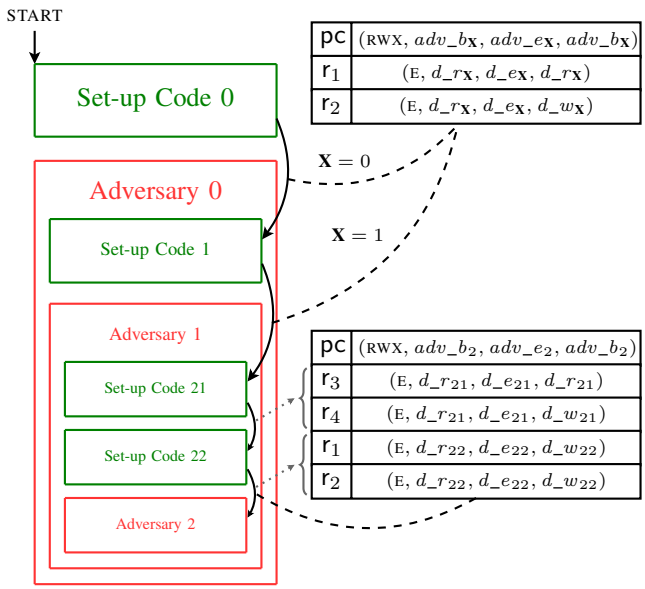


Figure 10: The flow of control and a few representative register states when jumping to the adversary in the execution of the motivating example. **Black**, dotted connectors indicate register state during the indicated transition. Gray connectors indicate that the pointed-to registers received the indicated value in the previously executed set-up code.

```
1  #1: Load MMIO capability
2  reqint r2 r25
3  move r25 pc
4  lea_a r25 d_ls0 r26
5  load r25 r25
6  lea_a r25 r2 r26
7  #2: Write MMIO value
8  store r25 r1
9  #3: Clear and return
10 rclear R_aux
11 jmp r0
```

Figure 11: The code for wrapper$_0$'s write closure (Write Wrapper 0 in Fig. 9a).

```
1  #0: Machine boots here
2  #1: MMIO capability
3  move r0 pc
4  lea_a r2 d_ls0 r1
5  move r1 pc
6  subseg r1 MMIO_b MMIO_e
7  store r2 r1
8  #2: Wrapper closures
9  move r1 pc
10 subseg r1 d_r0 d_e0
11 move r2 r1
12 lea_a r1 d_r0 r3
13 lea_a r2 d_w0 r3
14 restrict r1 E
15 restrict r2 E
16 #3: Adv capability
17 move r0 pc
18 subseg r0 adv_b0 adv_e0
19 lea_a r0 adv_b0 r3
20 #4: Clear, jump to adv
21 rclear R_clr*
22 jmp r0
```

*$R_{\text{clr}} = \text{RegName} \backslash \{\text{pc}, \text{r}_0, \text{r}_1, \text{r}_2\}$

Figure 12: The set-up code for wrapper$_0$ (Set-up Code 0 in Fig. 9a).

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. Fig. 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 Fig. 9b. The layout is similar to Fig. 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 Fig. 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 Fig. 7. LS_1 is defined in terms of a general local state $\mathrm{LS\_gen}_x$. $\mathrm{LS\_gen}_x$ 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 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, Fig. 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.

Fig. 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

```
1  #Template code
2  read_wrapper(check_read) ≜
3    is_addr r2 r25 r26
4    check_read
5    move r25 pc
6    lea_a r25 d_ls1 r26
7    load r25 r25
8    lea r25 1
9    load r26 r25
10   jmp r26
11
12 read_wrapper_1 ≜
13   read_wrapper(check_1)
14
15 #Check: < 1000 events
16 check_1 ≜
17   move r25 pc
18   lea_a r25 d_ls1 r26
19   load r25 r25
20   lea r25 3
21   load r26 r25
22   add r26 r26 1
23   lt r26 r26 1000
24   lea pc r26
25   fail
26   load r26 r25
27   add r26 r26 1
28   store r25 r26
```

Figure 13: The code for wrapper$_1$'s read closure (Read Wrapper 1 in Fig. 9b).

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 Fig. 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. Fig. 10 again shows the state of key registers at that point.

Finally, Fig. 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 Fig. 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 Fig. 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 Fig. 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.

B. 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

$$
\begin{aligned}
&\mathrm{P}_{22}\_\mathrm{bis} : \mathrm{Trace} \to \mathrm{Prop} \triangleq \\
&\quad \lambda t.\, \texttt{match}\ t\ \texttt{with} \\
&\quad \mid [\,] : \mathrm{True} \\
&\quad \mid t' +\!\!+ [e] : \mathrm{P}_{22}\_\mathrm{bis}(t') \wedge \\
&\qquad (e.\mathrm{addr} \neq a_{\mathrm{timer}} \Rightarrow \mathrm{last}(t') = \mathrm{Some}(\mathrm{IORead}, a_{\mathrm{timer}}, 1)) \\
&\mathrm{F}_{22}\_\mathrm{bis} : \mathrm{Trace} \to \mathrm{Trace} \triangleq \\
&\quad \lambda t.\, \mathrm{filter}(\lambda e.\, e.\mathrm{addr} = a_2 \vee e.\mathrm{addr} = a_{\mathrm{timer}}, t) \\
&\mathrm{LS}_{22}\_\mathrm{bis} : \mathrm{Trace} \to \mathrm{list\ Word} \triangleq \\
&\quad \lambda t.\, \mathrm{LS\_gen}_{22}([\texttt{if}\ \mathrm{last}(t) = \mathrm{Some}(\mathrm{IORead}, a_{\mathrm{timer}}, 1) \\
&\qquad \texttt{then}\ 1\ \texttt{else}\ 0)])
\end{aligned}
$$

Figure 14: Definitions involved in the second example

read from the timer address a_{timer} beforehand. In other words, wrapper$_{22}$_bis enforces the predicate P_{22}_bis in Fig. 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}):

OBJ-22-BIS

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

To correctly enforce P_{22}_bis $\circ$ F_{22}_bis, wrapper$_{22}$_bis consists of 2 different types of closures. First, a read and write closure similar to the ones demonstrated in Fig. 9, which clients can use to respectively read from and write to a_2 if the last event in 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 Fig. 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 wrapper$_1$.

Note that in our current set-up, 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 IV-A will discuss how our approach can be generalized to a setting where a closure is shared between multiple wrappers, e.g. wrapper$_{21}$ and wrapper$_{22}$_bis.

IV. PROVING THE SECURITY OBJECTIVES

This section outlines the high-level technical ideas that underlie the proofs of security objectives such as the ones in Fig. 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 IV-A discusses how each wrapper's invariants are formalized in Iris.

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 IV-B 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 IV-C discusses how we employ a semantic model to reason about the safety of unknown code in the capability machine.

In Section IV-D 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.

A. 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 Fig. 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 Fig. 16 shows.

1) 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

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

$$\textcircled{1}\ (\text{filter\_full}\ \gamma\ t * \text{filter\_val}\ \gamma\ F\ t') \twoheadrightarrow F(t) = t' \qquad \textcircled{2}\ (\text{filter\_val}\ \gamma\ F\ t * \text{filter\_val}\ \gamma\ F'\ t') \twoheadrightarrow \text{orthogonal}(F, F')$$

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

Figure 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.

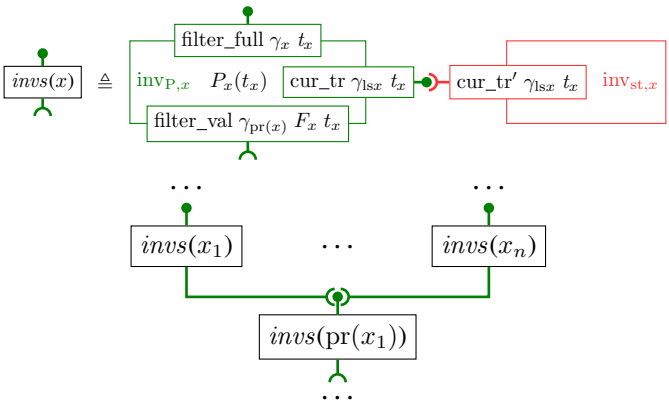


Figure 16: Figure illustrating the interaction of the different resources in the invariants $invs(x)$ (consisting of two invariants $\text{int}_{\text{P},x}$ and $\text{int}_{\text{st},x}$) of an arbitrary wrapper x. The bottom of the figure shows how the invariants of a wrapper $\text{pr}(x_1)$ are 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.

layers this way: the invariant for wrapper x_1 in Fig. 16 owns a resource filter_val $\gamma_{\text{pr}(x_1)}\ F_{x_1}\ t_{x_1}$, 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 $t_{\text{pr}(x_1)}$ that $\text{pr}(x_1)$ has. As shown in the bottom of the figure, this resource connects to the resource filter_full $\gamma_{\text{pr}(x_1)}\ t_{\text{pr}(x_1)}$ in wrapper $\text{pr}(x_1)$, which represents the full view. Here, $\gamma_{\text{pr}(x)}$ 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 $\gamma\ id\ t$ 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.

2) Reasoning modularly about sibling wrappers: In the case where multiple wrappers $x_1, \ldots, x_n$ have a common layer-below wrapper $\text{pr}(x_1)$, 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 $\text{orthogonal}(F, F')$ and defined in Fig. 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 $\gamma\ t$ and filter_val $\gamma\ F\ t$ are in fact defined in terms of this resource algebra. Fig. 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*. Fig. 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.

3) 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

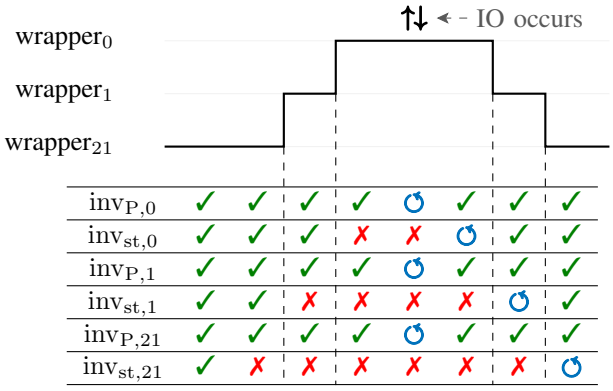


Figure 17: Overview of when different invariants need to hold when invoking wrapper$_{21}$ in our first example, where ↻ denotes an invariant being reestablished for a new view of the trace.

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, $invs(x)$ 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 [18] $\text{inv}_{\text{st},x}$, which ensures (among other things) that the local state is satisfied. The resources $\text{cur\_tr}\ \gamma_{\text{ls}x}\ t_x$ and $\text{cur\_tr}'\ \gamma_{\text{ls}x}\ t_x$ in Fig. 16 enforce that the view of the trace in both invariants is the same. Fig. 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.

4) Putting it all together: Given the representation of $\text{inv}_{\text{P},x}$ in Fig. 16, we now denote this atomic invariant with all of its parameters as $\text{inv}_{\text{P}}(F_x, P_x, tp_x, \gamma_{\text{pr}(x)}, \gamma_{\text{ls}x}, \gamma_x)$. 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 $\text{filter\_full}\ \gamma_x\ t_x$ is not present if x is a top-level wrapper. The arguments $\gamma_{\text{pr}(x)}$,$\gamma_{\text{ls}x}$,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:

$$\text{inv}_{\text{P}}(\text{id}, \_, \text{False}, \gamma_0, \gamma_{\text{ls}0}, \gamma_1) * \text{inv}_{\text{P}}(\text{id}, \_, \text{False}, \gamma_1, \gamma_{\text{ls}1}, \gamma_{21}) * \text{inv}_{\text{P}}(\text{F}_{21}, \text{P}_{21}, \text{True}, \gamma_{21}, \gamma_{\text{ls}21}, \_) * \boxed{\exists t.\ \text{filter\_val}\ \gamma_2\ \overline{\text{F}_2}\ [\,]}$$

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 $\text{filter\_val}\ \gamma_0\ \text{id}\ t$ 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 Fig. 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 wrapper$_{21}$ does not care what predicate the lower wrappers enforce when proving its own security objective.

B. 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. [11] 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]:

$$\forall \varphi.\ P \mathrel{-\!\!*} (Q \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{Repeat}\ \mathsf{SingleStep}\ \{s.\, \varphi(s)\}) \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{Repeat}\ \mathsf{SingleStep}\ \{s.\, \varphi(s)\}$$

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 details):

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

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 Fig. 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

[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.

w_{ret} in appropriate registers, then execution will jump back to w_{ret} with unmodified register contents. Additionally, a $\text{filter\_val}\ \gamma_x\ (\lambda t.\, \text{filter}(P_e, t))\ t$ resource is required to update the $\text{filter\_full}\ \gamma_x\ t$ resource in $\text{inv}_{\text{P},x}$ using property 3 in Fig. 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 $\text{filter\_val}\ \gamma_x\ (\lambda t.\, \text{filter}(P_e, t))\ t$. 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 Fig. 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*) [19], [20]. The client wrapper will not extract the $\text{filter\_val}\ \gamma_{\text{pr}(x)}\ (\lambda t.\, \text{filter}(P_e, t))\ t$ 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):

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

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 Fig. 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 $\text{filter\_val}\ \gamma_x\ (\lambda t.\, \text{filter}(P_e, t))\ t$ resource beforehand, but instead is allowed to rely on the invoked wrapper's $\text{filter\_full}\ \gamma_x\ t$ 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 $\text{filter\_full}\ \gamma_x\ t$ 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.

$$\boxed{\mathcal{E}(v)} \triangleq \forall reg.\, (\mathcal{R}(reg) * \mathsf{pc} \mapsto v * \mathop{\Large *}_{(r,w)\in reg, r\neq \mathsf{pc}} r \mapsto w) \twoheadrightarrow \mathsf{wp}\ \mathsf{Repeat\ SingleStep}\ \{\_.\ \top\}$$

$$\boxed{\mathcal{R}(reg)} \triangleq \mathop{\Large *}_{(r,w)\in reg, r\neq \mathsf{pc}} \mathcal{V}(w)$$

$$\boxed{\mathcal{V}(w)} \begin{cases} \mathcal{V}(z), \mathcal{V}(\text{O}, -) & \triangleq\ \top \\ \mathcal{V}(\text{E}, b, e, a) & \triangleq\ \square \triangleright \mathcal{E}(\text{RX}, b, e, a) \\ \mathcal{V}(p, b, e, a) & \triangleq\ \mathop{\Large *}_{a'\in[b,e)} \exists p'.\, p \preccurlyeq p' \wedge \boxed{\exists w.\, a' \mapsto_{p'} w * \mathcal{V}(w)} \end{cases}$$

Figure 18: Logical relations describing capability safety. Figure adapted from Georges et al. [11].

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 1 (View Shift Lowering):

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

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 $P_x(t +\!\!+ [e])$ hold (which is precisely what x checks before admitting the event anyway), and if x makes $\text{pr}(x)$ update the trace in the local state invariant inv_{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 inv_{P} are updated at once.

C. 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. [11], we reuse their *logical relation without revocation* and its formalization in Iris, and repeat a simplified version of the separation logic definition in Fig. 18. Additionally, we reprove their fundamental theorem (Theorem 2 below) in the presence of MMIO.

We restrict our explanation of Fig. 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 $\mathcal{V} : \text{Word} \to \text{iProp}$ (with iProp the type of propositions in Iris) defines when a word is safe.
- The expression relation $\mathcal{E} : \text{Word} \to \text{iProp}$ defines when a capability can safely be executed in the pc register.
- The register relation $\mathcal{R} : \text{Reg} \to \text{iProp}$ 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 [11], [21].

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 Fig. 10. This latter condition can be proven relatively straightforwardly from the contracts we described in Section IV-B. 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 2 (FTLR): $\forall w. \mathcal{V}(w) \twoheadrightarrow \mathcal{E}(w)$.

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 Fig. 9, $\mathcal{E}(\text{RWX}, adv\_b_{\mathbf{X}}, adv\_e_{\mathbf{X}}, adv\_b_{\mathbf{X}})$ 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, $\mathcal{V}(\text{RWX}, adv\_b_{\mathbf{X}}, adv\_e_{\mathbf{X}}, adv\_b_{\mathbf{X}})$ holds. The fundamental theorem then proves the result.

D. 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 III.
- Enforce their security objective using an atomic invariant of the form $\text{inv}_\text{P}(F_x, P_x, tp_x)$.
- 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 inv_st expresses both ownership of the code for x, and describes the closures for $\text{pr}(x)$ in the local state, and will hence be parameterized by the relevant addresses of both x and $\text{pr}(x)$. It is then denoted as follows: $\text{inv}_\text{st}(d\_r_{\text{pr}(x)}, d\_w_{\text{pr}(x)}, d\_e_{\text{pr}(x)}, d\_r_x, d\_w_x, d\_e_x)$.
- 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 Fig. 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 $\text{check\_spec\_read}(F_x, P_x, check\_read)$ and $\text{check\_spec\_write}(F_x, P_x, check\_write)$, respectively. These conditions are enforced in the local state invariant inv_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 IV-B 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 $\text{read\_spec}(d\_r_x, d\_e_x, d\_r_x, tp_x)$ and $\text{write\_spec}(d\_r_x, d\_e_x, d\_w_x, tp_x)$.

We can then state the following *lifting theorem*:

Theorem 3 (Lifting-theorem-read):

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

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 ($\mathrm{inv_{st}}$), and if an invariant enforces F and P on the trace ($\mathrm{inv_P}$), 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. $\mathrm{wrapper}_0$ in the examples, constitutes the base case in this theorem, and still has to be manually verified to satisfy $\mathrm{read\_spec}$ and $\mathrm{write\_spec}$, since its code and local state do not satisfy the template (cfr. Fig. 11 and LS_0 in Fig. 7).

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

Theorem 4 (Wrapper-safety-read):

$$\mathrm{read\_spec}(d\_r_x, d\_e_x, d\_r_x, \mathrm{True}) \twoheadrightarrow * \\ \mathcal{V}(\mathrm{E}, \mathrm{Global}, d\_r_x, d\_e_x, d\_r_x)$$

Again, a similar result holds for the write spec.

Since we have abstracted most reasoning related to wrappers into Theorems 3 and 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 4 still applied once the $\mathrm{read\_spec}$ and $\mathrm{write\_spec}$ were proven.

V. Related Work

Hardware-supported security mechanisms as well as software verification have been important ingredients of secure system development for a long time [22]. 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. [23] and Jia et al. [24] 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. [25] have used trusted execution environments (TEEs) like Intel SGX [26] or Sancus [27] 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 [28], [29], and the verification of Microsoft's Hyper-V hypervisor [30]. 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 [31] 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 [32] 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 [33], [34]. 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 [4], and have been proposed as a security foundation for both high-level languages [35], [12], [8] and assembly languages [5]. The fact that capabilities support nesting has been observed before, e.g. in Mark Miller's PhD thesis [36]. It has also been known for a long time that they provide a great foundation for nestable security architectures in high-level languages [37]. Over the past decade, capabilities at the instruction set architecture level have seen renewed interest, largely thanks to the CHERI project [6]. 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 [38], 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 [36]. It is only relatively recently that sufficiently powerful formal reasoning approaches have been developed that can prove such properties. Devriese et al. [7] proposed a reasoning approach based on logical relations, (what we now call) universal contracts and the concept of *effect parametricity*. Swasey et al. [8] 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 [10], [11]. These logics, as ours, are built in Iris [21], 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 [39] uses virtualization techniques to implement fine-grained protection on I/O through specific devices with a small TCB. Para-passthrough virtualization [14] 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. [40] 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. [41]) to deductive verification (e.g. [42]). 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.

VI. 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. [43]. Additionally, with the advent of modern capability hardware in the form of Arm's Morello prototype [44], 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.

References

[1] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and security isolation of library OSes for multi-process applications," in *European Conference on Computer Systems*. Association for Computing Machinery, Apr. 2014, pp. 1–14.

[2] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *OSDI*. USENIX Association, Oct. 2010.

[3] A. A. de Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-policies: Formally verified, tag-based security monitors," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 813–830.

[4] H. M. Levy, *Capability-Based Computer Systems*. Digital Press, 1984. [Online]. Available: https://homes.cs.washington.edu/~levy/capabook/

[5] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware Support for Fast Capability-based Addressing," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1994, pp. 319–327.

[6] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.

[7] D. Devriese, L. Birkedal, and F. Piessens, "Reasoning about object capabilities with logical relations and effect parametricity," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016*. IEEE, 2016, pp. 147–162.

[8] D. Swasey, D. Garg, and D. Dreyer, "Robust and Compositional Verification of Object Capability Patterns," in *OOPSLA*. ACM, 2017.

[9] T. Van Strydonck, F. Piessens, and D. Devriese, "Linear capabilities for fully abstract compilation of separation-logic-verified code," *Proc. ACM Program. Lang.*, vol. ICFP, 2019.

[10] L. Skorstengaard, D. Devriese, and L. 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*, 2018, pp. 475–501.

[11] A. L. Georges, A. Guéneau, T. Van Strydonck, A. Timany, A. Trieu, S. Huyghebaert, D. Devriese, and L. Birkedal, "Efficient and provable local capability revocation using uninitialized capabilities," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 6:1–6:30, Jan. 2021.

[12] L. Skorstengaard, D. Devriese, and L. Birkedal, "Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management," *ACM Transactions on Programming Languages and Systems*, vol. 42, no. 1, pp. 5:1–5:53, Dec. 2019.

[13] ——, "Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities," vol. 3, no. POPL, 2019.

[14] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "BitVisor: A thin hypervisor for enforcing {I/O} device security," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '09. ACM, Mar. 2009.

[15] Microchip Technology Inc., "SAM D5x/E5x Family Data Sheet," 2019. [Online]. Available: https://www.mouser.com/datasheet/2/268/60001507A-1130176.pdf

[16] T. Van Strydonck, A. L. Georges, A. Guéneau, A. Trieu, A. Timany, F. Piessens, L. Birkedal, and D. Devriese, "Proving full-system security properties under multiple attacker models on capability machines: Coq mechanization," 9 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5514350

[17] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." New York, NY, USA: Association for Computing Machinery, 2007.

[18] Iris Team, "The Iris documentation and Coq development." 2021. [Online]. Available: https://iris-project.org

[19] B. Jacobs and F. Piessens, "Expressive modular fine-grained concurrency specification." New York, NY, USA: Association for Computing Machinery, 2011.

[20] K. Svendsen, L. Birkedal, and M. Parkinson, "Higher-order concurrent abstract predicates," *Modular specification and verification for higher-order languages with state*, p. 108, 2012.

[21] R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *J. Funct. Program.*, vol. 28, p. e20, 2018.

[22] D. MacKenzie and G. Pottinger, "Mathematics, technology, and trust: Formal verification, computer security, and the U.S. military," *IEEE Ann. Hist. Comput.*, vol. 19, no. 3, pp. 41–59, 1997.

[23] M. Sammler, D. Garg, D. Dreyer, and T. Litak, "The high-level benefits of low-level sandboxing," vol. 4, no. POPL, 2019.

[24] L. Jia, S. Sen, D. Garg, and A. Datta, "A logic of programs with interface-confined code," in *2015 IEEE 28th Computer Security Foundations Symposium*, 2015, pp. 512–525.

[25] P. Agten, B. Jacobs, and F. Piessens, "Sound modular verification of C code executing in an unverified context," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, S. K. Rajamani and D. Walker, Eds. ACM, 2015, pp. 581–594.

[26] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[27] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Trans. Priv. Secur.*, vol. 20, no. 3, pp. 7:1–7:33, Jul. 2017.

[28] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an OS kernel," in *ACM Symposium on Operating Systems Principles 2009*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 207–220.

[29] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, no. 6, pp. 107–115, 2010.

[30] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science. Springer, pp. 23–42.

[31] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu, "Toward compositional verification of interruptible OS kernels and device drivers," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. Berger, Eds. ACM, 2016, pp. 431–447.

[32] ——, "Toward compositional verification of interruptible OS kernels and device drivers," *J. Autom. Reason.*, vol. 61, no. 1-4, pp. 141–189, 2018.

[33] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson, "Microkernels meet recursive virtual machines," in *USENIX Symposium on Operating Systems Design and Implementation*, K. Petersen and W. Zwaenepoel, Eds. ACM, 1996, pp. 137–151.

[34] B. Kauer, P. Veríssimo, and A. N. Bessani, "Recursive virtual machines for advanced security mechanisms," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*. IEEE Computer Society, 2011, pp. 117–122.

[35] S. Maffeis, J. C. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 125–140.

[36] M. S. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, Johns Hopkins University, 2006.

[37] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten, "Extensible security architecture for java," in *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, M. Banâtre, H. M. Levy, and W. M. Waite, Eds. ACM, 1997, pp. 116–128.

[38] H. Xia, J. Woodruff, H. Barral, L. Esswood, A. Joannou, R. Kovacsics, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Richardson, S. W. Moore, and R. N. M. Watson, "CheriRTOS: A Capability Model for Embedded Devices," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct. 2018, pp. 92–99.

[39] Y. Cheng, X. Ding, and R. H. Deng, "Driverguard: Virtualization-based fine-grained protection on I/O flows," *ACM Trans. Inf. Syst. Secur.*, vol. 16, no. 2, pp. 6:1–6:30, 2013.

[40] Y. Juglaret, C. Hritcu, A. A. D. Amorim, B. Eng, and B. C. Pierce, "Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 45–60.

[41] T. Ball, E. Bounimova, R. Kumar, and V. Levin, "SLAM2: static driver verification with under 4% false alarms," in *International Conference on Formal Methods in Computer-Aided Design*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 35–42.

[42] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, and F. Piessens, "Sound formal verification of linux's USB BP keyboard driver," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, A. Goodloe and S. Person, Eds., vol. 7226. Springer, 2012, pp. 210–215.

[43] A. T. Markettos, J. Baldwin, R. Bukin, P. G. Neumann, S. W. Moore, and R. N. Watson, "Position paper: Defending direct memory access with CHERI capabilities," 2020.

[44] Arm Limited, "Arm architecture reference manual supplement morello for a-profile architecture," 2020.