KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Security Primitives for Protected-Module Architectures

Based on Program-Counter-Based Memory Access Control

Raoul Strackx

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

December 2014

Security Primitives for Protected-Module Architectures

Based on Program-Counter-Based Memory Access Control

Raoul STRACKX

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

December 2014

ISBN 978-94-6018-940-1

D/2014/7515/163

Preface

This thesis is the result of 5 years of work, during which time I got a lot of help from a lot of people. I would like to take this opportunity to thank them here.

Foremost, I would like to thank my supervisor, Frank Piessens who constantly exceeded any expectations a PhD student may have of his supervisor. Frank, thank you for allowing me to discover where my interests lay, for every time you took up the role of "devil's advocate" (and apologized for having to do so) and for your endless support when I hardheadedly wanted to resubmit my paper to the next big conference. Also thank you for all the stimulating discussions over the years, even at times when you lacked the time.

I would also like to thank the members of my PhD jury: Frank Piessens, Wouter Joosen, Dave Clarke, Bart Preneel, Ingrid Verbauwhede, David Grawrock, Andrew Martin, and Jean Berlamont.

Also a thank you to Pieter Agten, Niels Avonds, Dave Clarke, Wilfried Daniels, Ruan de Clercq, Christophe Huygens, Niels Lambrigts, Job Noorman, Marco Patrignani, Frank Piessens Bart Preneel, Dries Schellekens, Anthony Van Herrewege, Gijs Vanspauwen, Ruben Vandevelde, and Ingrid Verbauwhede for sharing my interest in protected-module architectures and our interesting discussions. I hope that we can continue our collaboration in the near and distant future.

A special thanks to my 05.18 version 1.0 colleagues, Pieter Philippaerts and Frédéric Vogels. Sharing an office often felt like appearing in a live Monthy Python sketch (especially "The Argument Clinic" comes to mind) but I have great memories of those times.

To all the members of 05.18 version 1.1 and the "lunch buddies," a special thanks for all their jokes, sass and support of the last few years. Milica Milutinović, you seem to possess a remarkable amount of energy. I hope that some of it will rub off on me, eventually. Rula Sayaf, your lighthearted nature at all times

is remarkable and an example to us all. Pieter Agten, we shared some of my greatest travels and hope that many more will follow in the next few years. Frédéric Vogels, thanks for all the moelleux that you brought over the years. Because of all your (in your own words) “extravagances” I had to “endure” over the years, I know that one of the first things you will do when you read this text, is to look for spelling and grammar mistakes. Even with all your vices you’re a good friend, so I will even make it *eazie* for you. Mathy Vanhoef, since you started working here, you have been corrupted by Frédéric. He trained you well. Thanks for all the sass that we have exchanged. Jesper Cockx, you are everything a colleague and friend could wish for. Marco, it is great to have you joining us for lunch, even though your real Italian lunches often makes us jealous. Our sincere apologies if our jealousy ever turns violent. Jan Tobias, thanks for all your weird, but interesting stories of the last few months. They make for entertaining discussions over lunch.

I would also like to specifically thank some people for their support. Dominique Devriese, thank you for all the unexpected coffee breaks, and for all the interesting discussions that usually followed. It has been my pleasure.

Nick Nikiforakis, it was a lot of fun being able to share the excitement of imminent arrivals of paper notifications over the years. I hope we can continue to do so in the future, regardless of the continent where we live.

Gowri Suryanarayana, thanks for all the Indian sweets over the years, they were very tasty.

Marleen Somers, thank you for all your help over the years. You always made sure that conference registrations and hotel bookings were correct and handled quickly, even when you had a lot of other work and already were working overtime. Also your compassion and help when my luggage was permanently lost, was greatly appreciated.

Also to the other people of the secretariat, administration, and business office, thank you for all the fun conversations at the coffee machine and to try to keep all the paper work to a bare minimum. Unfortunately some of you have left the last few months, you’re truly missed. To the new colleagues, I hope that you’ll stay at least as long at the department as your predecessors because you’re great to be around with.

Yolande Berbers, thank you for your trust in my didactic abilities. Also the large amount of freedom has been greatly appreciated.

Also a special thanks to all the students that were forced to attend my exercise sessions. I have tried to make most lessons interesting and challenging, I hope I succeeded.

Work at the department for the last years has been great and I would also like to thank the many people who made that happen. With the grave risk of leaving somebody out: Yolande Berbers, Pieter Agten, Jasper Bogaerts, Ronald Cools, Dave Clarke, Jesper Cockx, Bart De Decker, Willem De Groef, Philippe De Ryck, Lieven Desmet, Dominique Devriese, Francesco Gadaleta, Tom Holvoet, Danny Hughes, Christophe Huygens, Bart Jacobs, Katrien Janssens, Fred Jonker, Wouter Joosen, Bert Lagaisse, Jef Maerien, Milica Milutinović, Jan Tobias Mühlberg, Syeda Nayyab Zia Naqvi, Nick Nikiforakis, Job Noorman, Steven Op de beeck, Marco Patrignani, Willem Penninckx, Pieter Philippaerts, Frank Piessens, Davy Preuveneers, José Proença, Andreas Put, Anne-Sophie Putseys, Esther Renson, Ghita Saevels, Rula Sayaf, Ilya Sergey, Jan Smans, Marleen Somers, Karen Spruyt, Eric Steegmans, Gowri Suryanarayana, Klaas Thoelen, Steven Van Acker, Rinde van Lon, Inge Vandenborne, Annick Vandijck, Mathy Vanhoef, Dries Vanoverberghe, Gijs Vanspauwen, Karen Verresen, Frédéric Vogels, Kim Wuyts and Yves Younan.

Last but definitely not least, I would also like to thank my family for their never-ending support. A special thanks to my dad for the long hours of making me do those language and mathematical exercises. They paid off in the end. To my mother, thank you for all the unexpected desserts when I was studying for exams and the encouraging words. They really helped to fight through tough times. To my sister, thank you for showing to me that an academic diploma may not always be easy, perseverance will always get you where you want to end up. To Gert, your funny stories and bad jokes always made me laugh, even when I was working against a deadline. Thank you for that. Also to my grandparents, thank you all for your never-ending support. It is very sad that some of you are not here to celebrate with us. And a very special thanks to Louis Swinnen, for having the foresight and courage to buy a computer and letting his grandchildren play with it.

I would also like to thank the following projects and scholarships for their financial support:

- The Agency for Innovation by Science and Technology in Flanders (IWT)
- The Intel Lab's University Research Office
- The Research Council KU Leuven: GOA TENSE (GOA/11/007)

Abstract

Our society increasingly depends on computing devices. Customers rely on laptops and mobile devices to access security sensitive applications such as online banking. Companies have to protect their trade secrets. And governments have to guard their country's critical infrastructure against espionage and sabotage.

Security of computing devices in such use cases is paramount and various security measures have been developed that raise the bar significantly for attackers. However, vulnerabilities in such systems still exist and are frequently exploited successfully. A common pitfall is that software security takes a layered approach where privileged layers keep getting extended with new components over the system's lifetime. This results in a snowball effect on the size of these privileged levels and this in turn increases the likelihood of software vulnerabilities. As all applications running on the device rely on the integrity of these layers, increasing the size of their code base has a negative impact on security of the overall device.

Many security measures have been proposed to automatically harden the most privileged software layer, but they all fail to provide strong security guarantees. In this thesis, we considered a different approach. We developed three security primitives that can be applied at most privilege levels. Unlike related approaches, this set of primitives is fixed. Software components can be added easily at any privilege level, but no additional primitives will be required. Moreover, these security primitives can provide provable security guarantees.

The most important security primitive is a program-counter-based access control mechanism. By enforcing different access rights on physical memory depending in which code module the processor is executing, sensitive parts of an application can be strongly isolated. For instance, a cryptographic key can be isolated to a protected module implementing a certificate-signing service. Since the key cannot be accessed by any other code than the module's, it cannot be stolen even if malware was already present on the platform. The access control mechanism also guarantees that the module is only accessible through the interface that it exposes explicitly.

Second, we enable protected modules to limit which protected modules can access their services. Unrestricted access to modules may still allow a large range of attacks. While an attacker may no longer be able to access a well-isolated cryptographic key, for example, the exposed interface may still allow her to sign forged certificates. We prevent such attacks using a capability-based access control mechanism; a module can only be accessed if the caller ever received the capability to do so.

Third, we propose a fast and practical state-continuity system. Using cryptographic properties, protected modules can use many services of the unprotected, legacy system while providing strong security guarantees. Unfortunately, these security guarantees only are ensured while the system is up and running. In practice systems crash, loose power or need to reboot. Integrity and confidentiality protecting the module's state before it is stored on disk is insufficient in such cases. Care must be taken that an attacker cannot present a stale state of a protected module as being fresh. Practical implementations are hindered by the substantial economical costs to add non-volatile storage to the processor. Many computing devices are already shipped with access-controlled non-volatile storage off-chip, but this memory is slow, very limited in size and wears out quickly. We propose an algorithm based on a simple hardware component to avoid all these shortcomings.

During this dissertation we also focused on how these newly-added security primitives can be applied to provide strong security guarantees. A fully-abstract compilation scheme was developed that ensures that software-level abstractions provided to programmers cannot be broken; for every low-level attack, there also exists an attack at source-code level. This significantly simplifies reasoning about the security guarantees that protected modules provide and increases security of the overall system.

Beknopte samenvatting

Onze samenleving hangt steeds meer af van haar computersystemen. Consumenten vertrouwen op laptops en mobiele toestellen om gevoelige toepassingen te raadplegen zoals internetbankieren. Bedrijven dienen hun (digitale) handelsgeheimen te beschermen. En overheden moeten de kritische infrastructuur van hun land waarborgen tegen spionage en sabotage.

Beveiliging van computersystemen in dergelijke toepassingen is van cruciaal belang. Talloze beschermingsmaatregelen zijn dan ook reeds ontwikkeld die succesvolle aanvallen aanzienlijk moeilijker maken. Kwetsbaarheden in software komen echter nog steeds voor en worden nog altijd frequent uitgebuit. Een veelvoorkomende valkuil is dat softwareontwikkeling een gelaagde structuur volgt waarbij geprivilegieerde lagen steeds groter worden wanneer nieuwe componenten worden toegevoegd tijdens de levensduur van het systeem. Dit resulteert in een sneeuwbaleffect van de grootte van deze geprivilegieerde lagen, wat op zijn beurt de kans op softwarekwetsbaarheden sterk vergroot. Aangezien alle applicaties die op het systeem uitvoeren op de integriteit van deze lagen steunen, heeft een toename in de grootte ervan een negatieve invloed op de veiligheid van het geheel van het systeem.

Talloze beschermingsmaatregelen zijn voorgesteld om de meest geprivilegieerde laag automatisch te beschermen, maar geen enkele biedt sterke veiligheidsgaranties. In deze thesis hebben we voor een andere aanpak geopteerd. We hebben drie beveiligingsprimitieven ontwikkeld die op bijna elke geprivilegieerde laag kunnen worden toegepast. In tegenstelling tot andere aanpakken, zijn deze primitieven niet veranderlijk. Nieuwe softwarecomponenten kunnen worden toegevoegd aan het systeem, zonder dat nieuwe primitieven moeten worden ontwikkeld. Bovendien bieden deze primitieven bewijsbare beveiligingsgaranties.

Het belangrijkste beveiligingsprimitief is het toegangscontrolemechanisme op basis van de *program counter*. Door verschillende toegangsrechten op fysiek geheugen af te dwingen op basis van de codemodule die de processor aan

het uitvoeren is, kunnen gevoelige delen van het programma sterk geïsoleerd worden. Neem als voorbeeld een beschermde module die certificaten digitaal ondertekent. De cryptografische sleutel van die module kan sterk geïsoleerd worden. Aangezien het controlemechanisme toegang tot deze sleutel van buiten de module verhindert, kan deze niet gestolen worden door kwaadaardige software. Hetzelfde mechanisme zorgt er ook voor dat de module enkel aangesproken kan worden via de interface die het expliciet beschikbaar stelt.

Een tweede beveiligingsprimitief laat beschermde modules toe om te beperken vanaf waar ze aangesproken kunnen worden. Ongelimiteerde toegang tot modules zou een brede waaier aan aanvallen niet kunnen verhinderen. Zo zou een aanvaller nog steeds zelf gefabriceerde certificaten kunnen laten ondertekenen, ook al heeft zij geen toegang tot de cryptografische sleutel. We verhinderen dergelijke aanvallen door het gebruik van een *capability*gebaseerde toegangscontrole; modules kunnen enkel aangesproken worden indien de aanroeper ooit de mogelijkheid hiervoor heeft gekregen.

Tot slot hebben we een snel en praktisch *state-continuity* systeem ontwikkeld. Door cryptografische primitieven toe te passen, kunnen beschermde modules gebruikmaken van onbetrouwbare toepassingen zonder dat dit een sterke invloed heeft op de geboden veiligheidsgaranties. Spijtig genoeg kunnen deze garanties enkel gegarandeerd worden zolang het systeem aan het uitvoeren is. Maar in de praktijk crashen systemen, verliezen ze plots hun stroomtoevoer of moeten deze opnieuw opstarten. Het beschermen van de integriteit en confidentialiteit van de status van modules is in deze gevallen onvoldoende. Maatregelen moeten genomen worden om ervoor te zorgen dat een aanvaller geen oude status als nieuw kan presenteren. Praktische implementaties worden hierbij gehinderd doordat niet-volatiel geheugen om economische redenen niet aan de processor kan worden toegevoegd. Veel computersystemen beschikken reeds over niet-volatiel geheugen met een toegangscontrolemechanisme naast de processor maar dit is echter traag, klein en het slijt snel. We hebben een alternatief ontwikkeld op basis van een eenvoudige hardwarecomponent en een algoritme dat deze tekortkomingen ontwijkt.

Tijdens deze dissertatie hebben we ook aandacht besteed aan hoe deze ontwikkelde beveiligingsprimitieven praktisch gebruikt kunnen worden om sterke veiligheidsgaranties te bieden. Er werd een volledig abstract compilatieproces ontwikkeld dat ervoor zorgt dat abstracties die aan programmeurs aangeboden worden op broncodeniveau, niet doorbroken kunnen worden. Zo bestaat voor elke aanval tegen modules die gebruik maken van de beveiligingsprimitieven, een overeenkomstige aanval op het niveau van de gebruikte programmeertaal. Dit vereenvoudigt de redenering over de aangeboden beveiligingsgaranties van modules enorm en zorgt voor een sterkere beveiliging van het systeem in het algemeen.

Contents

List of Figures

List of Tables

List of Acronyms

ABI	Application Binary Interface
AES	Advanced Encryption Standard
AES-NI	Advanced Encryption Standard New Instructions
AIK	Attestation Identity Key
AMT	Active Management Technology
API	Application Programming Interface
BIOS	Basic Input/Output System
CA	Certificate Authority
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DMA	Direct Memory Access
DRM	Digital Rights Management
DRTM	Dynamic Root of Trust for Measurement
EPT	Extended Page Table
HSM	Hardware Security Module
I/O	Input/Output
IOMMU	Input/Output Memory Management Unit
IPC	Inter-Process Communication
LOC	Lines of Code
LPC	Low Pin Count
MAC	Message Authentication Code

MMU	Memory Management Unit
MTM	Mobile Trusted Module
NVRAM	Non-Volatile Random Access Memory
PAL	Piece of Application Logic
PCR	Platform Configuration Register
PMA	Protected-Module Architecture
PUF	Physical Unclonable Function
RAM	Random Access Memory
rPUF	Reconfigurable Physical Unclonable Function
RSA	Rivest Shamir Adleman
SHA	Secure Hash Algorithm
SLB	Secure Load Block
SMI	System Management Interrupt
SMM	System Management Mode
SPI	Serial Peripheral Interface
SPM	Self-Protecting Module
SRNG	Secure Random Number Generator
SRTM	Static Root of Trust for Measurement
TCB	Trusted Computing Base
TOCTOU	Time-of-Check Time-of-Use
TPM	Trusted Platform Module
TXT	Trusted Execution Technology (a.k.a. LaGrande Technology)
UPS	Uninterruptible Power Supply
VM	Virtual Machine
VMM	Virtual Machine Monitor

Chapter 1

Introduction

As of January 1, 2014, the International Tennis Federation allows tennis players to use Player Analysis Technology (PAT) [58]. This will make the sport significantly more technological, far exceeding Hawk-Eye, the ball-tracking software that is already in use at some big tournaments. By incorporating motion sensors in tennis rackets [59], for example, coaches can analyze the force the tennis ball is hit with, its average position on the racket, etc. and use this information to better instruct their players. However, if this data falls in the hands of the player's opponent, signs of fatigue of certain muscle groups may also be deduced from it and give an unfair advantage.

Player Analysis Technology is only one of many examples of how our society increasingly relies on (the security[1] of) computing devices, even in case of seemingly low-tech undertakings such as playing tennis. Other examples include smartcards that support over-the-air updates, programmable sensor-networks, setup boxes and internet-connected TVs. Also infrastructure critical to our society depend on software. SCADA (Supervisory Control And Data Acquisition) systems are used to detect line voltage and take parts of our electrical grid online and offline. They also control water flow and pipe pressure of our water system and have many other critical applications. Of course also more obvious examples can be added to this list, such as online banking, e-government and cloud computing where multiple, mutual distrusting virtual servers execute on the same physical server.

As these devices are increasingly becoming interconnected and support

[1]In this dissertation we will following the definitions of terms (e.g., security, safety, system, etc.) of Avizienis et al. [9] as much as possible but make an exception for standard terminology (e.g, type safety, memory-safe language, etc.).

software extensibility, the chances of them being successfully exploited increase significantly. This raises important research questions. How can we protect sensitive information from falling into the wrong hands? What architectural support is required to provide strong security guarantees while supporting interconnected, extensible devices? How can we deal with software extensions of stakeholders that do not necessarily trust one another? How do we deal with legacy software in such a system that may be vulnerable to attacks? Can we reduce the impact that such vulnerabilities can have?

To situate work accomplished during this PhD, we start this chapter with a brief overview how software can be attacked and which kind of security measures already exist. In Section 1.3 we discuss why, in spite of a large body of security measures, software vulnerabilities are still exploited daily. Next, we discuss the core idea of "Protected-Module Architectures" (PMAs), a promising novel research direction, and give a short overview of our own contributions to this area of research (Sections 1.4 and 1.5). Finally, we give a short overview of independent research results by industry and give an overview of the remainder of this dissertation text (Sections 1.6 and 1.7).

Readers already familiar with low-level software security may want to skip to Section 1.3.

1.1 Attacking Software

Recent advantages in artificial intelligence show very interesting results. Many will remember IBM's Deep Blue computer defeating reigning chess champion Garry Kasparov in 1997 [55]. Or, more recently, IBM's Watson computer winning a game of "Jeopardy!" in 2011 [45]. While the hardware capacity of Deep Blue and Watson were impressive (at least at their time), they architecturally don't differ that much from an average laptop or mobile phone; the hardware only accepts simple, basic instructions such as fetch the memory contents at address x, add the contents of register $r1$ and $r2$, etc.

Artificial intelligence emerges from how these basic instructions are combined. Writing these hardware instructions directly by hand would be a very tedious and error prone task. Instead, applications are written in a programming language that provides the programmer with a much more abstract representation of the platform. She can, for example, specify that the code requires a square board of 8 by 8 without having to specify exactly where it needs to be stored in memory. Moving a pawn can then be represented as swapping the contents of two memory cells. Deep Blue's software uses such more abstract views of the game and computes all possible steps that can be taken (to a certain limit) starting from the current setup of the chess board. Next it analyses and grades every resulting setup and finally outputs the step leading to the best outcome.

Software written in a programming language cannot be understood directly by a processor. First, it needs to be passed to another application, the compiler, that will translate the source code to machine instructions.

Vulnerabilities can originate at any level of this development process. A programmer can make a mistake when, for example, implementing the allowed steps a pawn may take. Or the compiler may have been written incorrectly, causing the application to sometimes behave incorrectly. Or even the hardware itself may be faulty. Not every programming mistake may be exploitable for attack. Certain bugs may only lead to undesirable side-effects such as a chess game that allows players to make illegal moves. Others are more severe and enable attackers to gain complete control over the application; they may be able to inject their own code inside the application and force its execution. This may lead to a large set of capabilities ranging from overwriting high scores of a chess game, to inspecting and modifying files on disk and even to operating the webcam without the user's knowledge.

Buffer Overflows. One of the most notorious software vulnerabilities are buffer overflows. According to the NIST National Vulnerability Database [86], the percentage of buffer overflow vulnerabilities of all vulnerabilities in publicly available software reported, increased dramatically from a mere 2.13% in 2006 to 15.95% in 2011. Since then the percentage of reported vulnerabilities dropped slightly to 12.10% for the first quarter of 2014.

Buffer overflows occur when a program tries to write past the boundaries of a memory buffer. Just as an overflowing bucket of water may cause a nearby electrical circuit to shortcut, the application may exhibit unexpected behavior. In general there are two possible attack targets. Most commonly, an attacker will attempt to overwrite control flow data. By, for example, overwriting the memory location containing the return address of a function – the address where execution should be resumed after the function completed – an attacker is able to redirect execution to her own injected code.

A second attack target is non-control data. Such attacks do not attempt to force the execution of injected code, but overwrite data stored by the application. Obvious targets of this kind of attack are variables keeping track of which user is logged in. Once data has been overwritten, the application executes normally, but operates on incorrect data, possibly giving users elevated privileges.

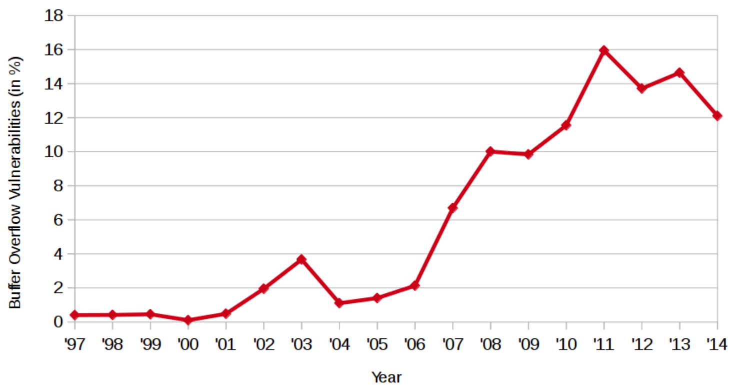


Figure 1.1: The NIST National Vulnerability Database [86] reports a significant increase in the portion of buffer overflow vulnerabilities in publicly available software since 2006. In 2013 (the last full year figures are available of) they accounted for 14.6% of all software vulnerabilities reported.

Buffer Overreads. Data does not always have to be overwritten to cause security issues. Early 2009, we demonstrated [122] that buffer overreads are a realistic threat. While similar to a buffer overflow, during a buffer overread attack data is only read passed the bounds of a memory region, it is *not* overwritten. Passing unintended data to an attacker may be a security issue for two reasons: (1) valuable information may be deducible from the data leaked and used to bypass security measures relying on the secrecy of memory [13, 32, 33], or (2) the outputted information itself can be sensitive such as passwords or cryptographic keys belonging to different users of the application.

Probably the most well-known example of the latter is Heartbleed. This buffer overread vulnerability was discovered in the OpenSSL library [28, 83] and disclosed on April 7, 2014. The OpenSSL library is intended to guarantee integrity and confidentiality of network connections, but a missing bounds check could result in sensitive memory contents being leaked to an attacker. As private user information on a webserver, including login credentials, could be extracted and given the widespread use of the vulnerable library, the security incident received world-wide attention. Later it also became apparent that even private SSL keys may have leaked [48].

1.2 Existing Security Measures

Given the ever increasing dependence of modern society on software and the impact software vulnerabilities may have, a large body of security measures has been developed. To situate the research contributions of this PhD, we discuss here very briefly different areas of research. For a more elaborate overview, we refer the reader to other work [42, 146].

1.2.1 More Secure Libraries

Programming languages such as C or C++ do not automatically store the size of allocated memory regions in memory. It is up to the programmer to keep track of this information. This is often cumbersome and simply omitted. Especially older library functions such as `strcpy` that copies a string (an array of characters) from one location to another, do not require a programmer to specify the maximum number of characters that can be copied. After buffer overflow vulnerabilities received more attention, such functions have since been updated. Older functions are still available for backward-compatibility reasons.

Interestingly enough, the “more secure” version is often still susceptible to a buffer overread. Consider, for example, `strncpy` that, contrary to its older and less secure variant, takes an additional argument specifying the length of the output buffer. While it prevents the destination buffer from overflowing, it does *not* guarantee that the newly written string is terminated by a ‘`\0`’ character. Most string functions, however, will rely upon this character to mark the end of the string and data located after the string in memory may be outputted to an attacker. It is unclear how often such vulnerabilities occur in practice.

1.2.2 Hardening Legacy Software Automatically

Writing software is a very time-consuming undertaking. Hence, one important line of research focuses on minor compiler or operating system updates to harden possibly vulnerable software. Such research proposals do not remove the vulnerabilities at hand, but they prevent that vulnerabilities can be exploited or, at least, significantly raise the bar for attackers.

One class of security measures tries to detect that a buffer overflow has taken place. StackGuard [32, 33] and ProPolice [44] take this approach and are two of the most widely applied security measures against buffer overflows. StackGuard enables the detection of a stack-based buffer overflow by placing a random value, called a *canary*, before sensitive control data on the stack. A buffer overflow trying to overflow the return address of the current function, will also overwrite the canary. By checking the integrity of the canary before returning from the current function, a buffer overflow attack may be detected and the application can be stopped before any harmful event can occur.

ProPolice [44] operates similar to StackGuard but also re-orders local variables, avoiding that a buffer overflow can overwrite valuable data such as pointers before the canary is overwritten.

Other security measures defending legacy software do not try to detect buffer overflows, but make it significantly more difficult to execute attacks successfully. Address Space Layout Randomization (ASLR) [13], one of the most applied security measures of this class, randomizes the memory layout of applications at launch-time. Every time an application is started, different memory locations are assigned to valuable data. This does not prevent buffer overflows, but makes it much harder for an attacker to overflow a buffer with sensible data. Redirecting control flow, for instance, is much harder as the location of injected code at runtime is unknown to the attacker.

StackGuard, ProPolice and ASLR rely on the memory secrecy assumption: an attacker cannot read unintended memory contents prior to an attack. But in the event of a buffer overread this assumption does not always hold [122]. Data or code pointers may leak to the attacker and enable her to derandomize the memory layout.

Multistack [148] takes another approach without relying on the memory secrecy assumption. It partitions the program stack into multiple stacks, separating likely attack vectors (e.g., buffers of characters) from likely attack targets (e.g., return addresses or function pointers). Using hardware memory protection, it avoids that a buffer overflow on one stack can overflow onto another. Benchmarks show that multistack can be implemented with almost no overhead on x86 processors. In 2010 we evaluated Multistack on ARM processors and proposed some optimizations to reduce overhead on this platform as well [121].

1.2.3 Memory-Safe Languages

In the previous paragraphs we focused mainly on buffer overflow vulnerabilities because they are the most exploited type of low-level software vulnerabilities. But also other memory errors exists such as double freeing an allocated memory region or dereferencing a pointer after it has been freed. Such vulnerabilities can occur in older languages (e.g., C and C++) because they rely on the programmer to avoid such errors. Failure to comply may result in security vulnerabilities.

Detailed knowledge of the application, compiler, linker and operating system is required to determine whether these vulnerabilities are practically exploitable.

Most modern languages such as Java, C#, Haskell and OCaml, avoid such vulnerabilities altogether by relying on a sound type system. Before a Java application is compiled, for example, it is checked that it complies with the type system at hand. Errors such as writing to a memory region before it is allocated, will be detected before the application can be executed. Other errors such as writing past the bounds of an array, are more difficult to detect at compile time and dynamic checks are inserted.

While memory-safe languages are clearly superior in terms of security, non-memory-safe languages are still widely used. For instance, the C programming language is still the most widely used language according to the TIOBE Programming Community [130]. The reason for this is threefold: (1) Many legacy applications and libraries are written in C and form the base for newer versions. Rewriting them in a memory-safe programming language usually is not economically viable. (2) While many languages have clear advantages from an academic point of view, they require a different thought process and highly-skilled programmers in these languages are hard to find. (3) Operating systems or other applications require close interaction with the hardware. This is much easier to accomplish in low-level languages such as C. Moreover, some low-end devices are significantly resource constrained and the increased performance, memory and energy costs of memory-safe languages may not be acceptable.

1.2.4 Software Verification

The use of a memory-safe programming language can avoid exploitable memory-error vulnerabilities in an application. But also other implementation errors may enable successful attacks. Insufficient input validation in a web application may lead to SQL injection or Cross Side Scripting (XSS) attacks. Use of unsanitized user-provided paths may enable an attacker to traverse the entire file system and access sensitive files. Or unexpectedly large input data may cause the application to stop responding.

Such vulnerabilities (as well as memory errors) can be detected using formal software verification. Tools aiding in formal verification, enable programmers to specify key properties of the application and guarantee with mathematical certainty that they always hold. Unfortunately formal verification of even simple applications can only be done by highly-skilled users and is very time consuming. Verification of large applications is almost infeasible in practice. Instead, significant assumptions are often made such as that a library always behaves correctly or that the underlying operating system cannot be attacked. Nonetheless, verification tools such as VeriFast [60] and VCC [29] are promising.

1.2.5 OS Protection Features

In practice an attacker may be able to gain access to a computing system in two ways. Either a vulnerable application could be exploited, or the user can be tricked into executing the attacker's provided code (e.g., by sending her an obfuscated executable via e-mail). Security measures need to be taken to ensure that once an attacker gained execution privileges on the platform, she does not have full control over the system. Two important features of modern operating systems play a key role, with limited success.

First, the introduction of protection rings significantly increased security. Protection rings enable a strict separation between the kernel – the more privileged piece of software that can directly access hardware such as the hard disk and decides which application can run – and applications that take advantage of hardware abstractions provided by the kernel. If a user-level application attempts to issue an instruction that requires kernel privileges, this will cause a trap to the kernel where it will be handled accordingly (e.g., by terminating the application). Similarly, the hardware will prevent a user-level application from accessing kernel memory directly. The kernel can only be accessed using a specific interface of system calls that it provides. Using this separation of protection rings, applications can also be isolated from one another; an attacker who successfully attacked an application is not able to directly read or write memory locations belonging to another process.

The arrival of protection ring hardware sparked the question of what should be included in the kernel and what should execute at user-level.[2] A number of different operating system architectures have been proposed, but most fall in one of two groups. One group, the monolithical kernels, implement process isolation, memory management, device drivers, etc. in the kernel. The other group, the microkernels, only implement a minimal amount of features at the highest privilege level. Components such as memory management and device drivers are offloaded to services executing at user-level. While this has clear advantages from a security perspective, most modern operating systems are monolithical systems. We elaborate further on microkernels in Appendix B.

Second, the operating system is able to define an access control policy on resources. While most computing devices today are truly *personal* computers,

[2]Some researchers even proposed the use of more than two privilege level. The influential MULTICS operating system [52], for example, called for 64 privilege rings.

most devices used to be shared. Hence, there was a need to prevent files belonging to one user from being accessed by another. This sparked the development of access control policies which are widely implemented. Unfortunately most applications today execute with full user privileges and an attacker who exploited such an application can access all the user's files, even if access to those files is not required for the correct working of the application.

1.2.6 Hardware Security Modules

Some applications require extremely strong security guarantees. For instance, private cryptographic keys used to sign government-issued electronic identity cards or website certificates must not, under any circumstances, leak to an attacker. To protect such high-value data, they are often stored in a hardware security module (HSM). These secure co-processors protect sensitive data against two types of attacks. First, to protect against software attacks, they only provide limited functionality and usually do not allow third-party code to run on its processor. Untrusted software running on the main processor can only access the HSM via a secure interface that prevents unintended behavior. An HSM, for example, may provide an interface to create and use cryptographic keys (e.g., to sign certificates), but should prevent that cryptographic keys can be exported from the HSM in plaintext.

Second, security measures are taken to prevent hardware attacks against the HSM. While software attacks are often the easiest attacks to execute – and thus usually the preferable option for attackers – there may not be any exploitable vulnerability in the software of the HSM. In such situations a determined attacker may target the HSM's hardware directly. Hardware attacks can be divided in two categories. In invasive attacks the hardware chip is directly physically accessed. An attacker may, for example, ground signals preventing correct execution of instructions. Invasive attacks do not always actively modify the behavior of the chip. In passive attacks of this category data buses may be probed to intercept sensitive information that pass over them. A variety of countermeasures exist to defend against invasive attacks, ranging from covering hardware components in epoxy making them harder to access to continuous monitoring the integrity of a protective mesh over the chip.

Non-invasive attacks on the other hand, preserve the chip's physical integrity. Passive attacks in this category only monitor information that can be observed externally such as timing differences, power consumption or electromagnetic radiation. When this information can be correlated with the internal state of the chip, sensitive information may leak to an attacker. An attacker may however also actively modify the chip's physical environment. Providing spikes in the power supply, for example, may cause unintended behavior of the chip and (indirectly) leak sensitive information. Sensors onboard the chip may mitigate such attacks.

To boost security of commodity computing devices, most modern desktop and laptops are also equipped with an HSM, called the Trusted Platform Module (TPM). While this chip is very resource constrained and operates at very low speeds, it has a number of interesting security primitives. We briefly discuss some of these primitives we will use throughout the text but refer to other work for a more elaborate discussion [49, 76, 95].

Platform Configuration Registers. One of the key components of the TPM chip are its Platform Configuration Registers (PCRs). PCRs are a set of registers that can be used to securely collect the state of the platform. To ensure that malicious software cannot tamper with the contents of PCR registers, they cannot be set to a specific value. A PCR_i can only be extended with measurement m by hashing its current content appended with m:

$$\text{PCR}_i^{\text{new}} \leftarrow \text{SHA-1}(\text{PCR}_i^{\text{old}}||m)$$

There are two types of PCR registers. Static PCRs are initialized to 0 at boot time and cannot be reset afterwards. Dynamic PCRs, in contrast, are set to -1 at boot time and can be reset to 0 after the system booted. The different initial values enable a third party to distinguish between reboots and PCR resets.

To collect the state of the platform, every component in the boot process first computes the hash of the next boot component, extends a PCR register with the result and only then hands over control. Modification of one of these components, for example to provide a persistent backdoor to an attacker, will result in a different PCR value. Hence, when PCR content remains unchanged, the system has booted to a pristine state.

Attestation. In a number of use cases it is desirable that a user can prove certain properties to a third party. Employees connecting to a company network, for example, may need to prove that their operating system was not compromised and runs the latest software updates. Or users of a public network may need to prove that they run an up-to-date anti-virus scanner. The TPM chip can be used to provide such attestations; the TPM chip can sign the contents of one or more PCR registers that can be presented to the third party. As the used private key cannot be extracted from the TPM chip and PCR contents cannot be set to a specific value by an attacker, this attests the state of the platform.

Sealed Storage. Sensitive data can also be tied to a specific state of the platform. Software can provide sensitive data to the TPM chip and request that it is sealed to the contents of one or more PCR registers. The resulting ciphertext can be stored at an untrusted location. Only when the systems enters a state where PCR content matches the specified value at the time of sealing, will the TPM decrypt the ciphertext.

NVRAM. Sensitive data can also be stored in TPM NVRAM based on PCR contents. Unfortunately TPM NVRAM has significant drawbacks. Most TPM implementations only provide 1,280 bytes of non-volatile memory and support only 100,000 write operations. Accessing TPM NVRAM every second would wear it out in less than 28 hours.

Monotonic Counters. Finally, TPM chips are also shipped with at least 4 monotonic counters; these are counters that can be incremented but never decremented. Unfortunately, the TPMv1.2 specification [132] only requires these counters to be incrementable at least once every 5 seconds. TPM vendors may provide increments at higher speeds but counters must not wear out within 7 years of operation.

1.3 Why Software Security Is Still a Mess

The security measures presented in Section 1.2 all seem promising. But users are still presented with daily software updates to patch vulnerable systems and have the impression that "everything can be attacked successfully." Why is the state of software security in practice so bad?

One could argue that software security is still a very young field, especially when compared to medicine or physics, and that software security will increase automatically over time when programmers are becoming better educated. While this may be the case for some type of vulnerabilities, it is unlikely that it will stop all attacks. Some vulnerabilities are just too subtle and too easily overlooked, especially when under time pressure.

Development of security measures, combined with better education, may provide better chances of success. In order to get widely adopted, security measures should meet some key properties:

- *Easily Applicable:* Software security measures need to be easy to apply by users and developers. Security measures that hinder users in their daily operations, will simply be disabled. Similarly, developers won't apply a security measure if it is too cumbersome. Software verification is a prime example of such a security measure; formally proving that software meets some security property is non-trivial and too time costly.

- *Limited Overhead:* Security measures that significantly impact the performance, memory usage or energy requirements, may not be suitable for widespread deployment. For instance, security measures exists that remove all buffer overflow vulnerabilities of an application by adding bounds checks. However, due to their huge performance overhead (PAriCheck [147] and Baggy Bounce [5] pose a 49% and 60% performance overhead respectively on the SPEC2000int benchmark), they are not widely deployed.

- *Evolutionary:* Writing software is a labor-intensive undertaking. Only in extraordinary cases will existing software be discarded and re-implemented to provide stronger security guarantees. Successful security measures such as StackGuard, ProPolice and ASLR can be applied to fortify most legacy software or simply be disabled for non-compatible applications.

- *Effectiveness:* The cost users and software vendors are willing to pay for increased security, naturally also depends on the effectiveness of the applied security measures. Unfortunately, most security measures rely on a huge Trusted Computing Base (TCB) – the hardware and software that needs to be trusted upon – especially when applied on commodity computing devices equipped with a huge monolithical kernel such as MS Windows or Linux. The security guarantees that such security measures can provide is thus also limited.

- *Gain a Critical Mass:* Software vendors are usually hesitant to apply newly developed security measures. Their impact on legacy software may not yet be clear and long-term support may be uncertain. Especially switching to another, more secure language is a risky undertaking; without widespread adoption, the language compiler and runtime environment may not be maintained in the future and skilled developers may be hard to find. The decision is also hard to revert in the future.

To minimize overhead, security measures are often specifically tailored to a very specific type of vulnerability (e.g., stack-based buffer overflows). Other types need to be addressed by applying additional countermeasures. The hope is that by stacking security measures, the bar is raised so high for attackers that successful exploits are infeasible. Unfortunately the sheer size of the codebase that needs to be protected makes this approach unlikely to succeed. The number of vulnerabilities that are exploited on a daily basis, are a direct result.

1.4 A New Deal:[3] Protected Module Architectures

Software vendors and system administrators could continue to add newly developed security measures to computing devices, raising the bar for attackers ever higher. While this would make attacks more challenging to execute, it is unlikely that this approach will ever be able to provide provably strong security guarantees. Protected-Module Architectures (PMAs) provide a viable alternative.

Concept. In 2008 McCune et al. proposed [78] a radical new research direction based on two observations. First, modern monolithical operating systems provide process isolation primitives, but the kernel is too large to be implemented free from vulnerabilities. Moreover, its continuous extension to support new file systems, process scheduling algorithms and new peripherals generates a snowball effect of an ever growing TCB. Both properties make formal verification not only infeasible, it would pose a huge barrier for third-party vendors to develop new kernel services as they too would need to be formally verified. In order to provide strong security guarantees to applications running on commodity computing devices, a novel isolation approach is required that avoids the pitfalls of common monolithical kernels.[4]

Second, not all components of an application are security sensitive and a vulnerability in one compartment should not affect security of another. For instance, a vulnerability in a component that parses incoming messages or that collects and sends memory dumps when the application crashes, should not affect the secrecy of the cryptographic key used by a component providing an encryption/decryption service. Isolation of sensitive parts of an application can lead to limited-sized components that can provide strong security guarantees as they are mostly self-contained; they only need to place limited trust in services provided by other parts of the application or the operating system. They are still dependent on the kernel to eventually receive allocated CPU time or get access to files, but failure to provide these services does not necessarily break required security properties. For instance, files could be integrity and version protected

[3]The term "New Deal" refers to a series of measures taken by US Congress and President Franklin D. Roosevelt in response to the Great Depression of the 1930s [14]. They focused on Relief, Recovery, and Reform. Given the poor state of security of high-end computing devices and even more so of low-end, embedded devices, society is also in need of a "new deal" regarding system security.

[4]Microkernels largely avoid these pitfalls already by minimizing the amount of services running at kernel-level. The seL4 microkernel even has been formally verified completely [64]. However, virtually all commodity applications run on top of monolithical kernels and attempts to rebuild them into microkernels have largely failed. In Appendix B we discuss the similarities and differences between Protected-Module Architectures and microkernels in more detail.

before they are passed to the operating system for storage. Kernel-level malware may then corrupt stored files, delete them or replay old versions, but sensitive information cannot leak. Similarly, an attacker who gained kernel-level access, may simply stop security sensitive components from being scheduled or prevent the system from ever resuming normal operation after it rebooted.

Based on these observations, McCune et al. proposed a novel protection mechanism. It enables module vendors to provide strong security guarantees while relying only on a limited and stable set of primitives – additional primitives will not need to be added in the future. In contrast to many existing security measures, the protection mechanism does not assume that the system started in a pristine state. An attacker may already have gained access to the highest privilege levels. Only the provided primitives must not be subvertible. Security architectures that follow a similar reasoning have since become known as *"Protected-Module Architectures"* (PMAs).

Which security primitives are presented to modules depends on the implementation of the PMA, but all ensure two vital properties. Most importantly, protected modules are in complete control of their own security. They are completely isolated from the rest of the system, including from other modules. Malware executing at any privilege level cannot directly access modules' memory content.

Given that an attacker may have infiltrated to the highest privilege levels of the system, modules may have been compromised before they could be protected. In such cases PMAs must ensure that an attacker does not gain any additional power. Their security primitives guarantee that (1) the compromised module cannot access previously stored secrets and (2) the module will fail to attest its correct execution to a (remote) verifier.

Flicker. In the same work [78] McCune et al. also presented Flicker, a prototype of such a protected-module architecture. Flicker relied heavily on the security features provided by the TPM chip and the late launch feature of the then recent AMD processors.[5] A late launch was designed to securely start a hypervisor or a security kernel. When issued, a late launch provides a Secure Load Block (SLB) as parameter. This code and data chunk will receive control after the late launch. To prevent software attacks against the SLB, a late launch sequence disables interrupts, prevents direct memory access (DMA) to the SLB and disables debug access. Virtual memory is disabled as well. To tie TPM features to the provided SLB, the TPM's dynamic PCR registers are reset and PCR 17 is extended with the measurement of the SLB.

[5]Intel processors provide with Trusted eXecution Technology (TXT) similar capabilities.

Flicker uses a late launch sequence to completely isolate sensitive parts of an application, called Pieces of Application Logic (PALs) by its authors, from the rest of the system. As a result, PALs are short-lived. Every time a PAL is called, it needs to marshall all required inputs and initiate a late launch to start the PAL's execution. Afterwards normal, untrusted execution is resumed and the PAL's output can be unmarshalled. When required, PALs can seal sensitive data for the next invocation of the same or another PAL.

Flicker's merit is that it is able to provide provable security properties by applying existing security-hardware primitives in a completely new way. Unfortunately, since these primitives were never designed for such use cases, they incur some significant disadvantages. First, development of PALs is a challenging undertaking, especially when state needs to be maintained between invocations or shared between multiple PALs. Special care may also have to be taken in such cases to ensure that an attacker cannot replay messages. The fact that PALs cannot access the legacy application's memory region, complicates their development as well. While some inputs may be marshalled/unmarshalled easily, this is definitely not always the case, for example, when a PAL needs to access a graph stored in unprotected memory.

Second, Flicker incurs a significant performance overhead. Issuing a late launch sequence for every PAL invocation and accessing the TPM to retrieve previously stored data, makes Flicker unsuitable to protect sensitive parts of performance-critical applications. However, it must be noted that this overhead only occurs when crossing PAL/legacy-code boundaries. Execution of legacy applications is not affected.

Third, Flicker takes advantage of complex hardware. Their power and energy requirements makes its approach infeasible for low-end applications such as sensor networks.

Research Questions. Despite Flicker's drawbacks, it showed to be a very promising research direction. It finally demonstrated how strong security guarantees could be offered to applications running on huge monolithical operating systems while only relying on an extremely limited software TCB of 250 lines of code. This lead to a number of research questions. Some of them have been (partially) answered during this doctoral work, others are still open for further research.

- *What is the minimal set of security primitives required?* We have proposed a novel program-counter-based access control mechanism [120] that provides strong isolation and enables lightweight inter-module communication (see also Chapter 2). Later we added a new primitive:

state-continuous execution [116, 117] that guarantees that even in the event of unexpected crashes or power failure, an attacker cannot roll back a module's state (see Chapter 4). We expect that this set of primitives may be reduced further. State-continuity support for example may only require access to an attestable, monotonic counter.

- *Can PMAs suitable for low-end devices be developed?* Noorman et al. [87] showed that the proposed program-counter-based access control mechanism can be implemented efficiently in hardware and used in low-end applications such as sensor networks.
- *Which security guarantees can PMAs offer? How can these be verified?* Agten et al. [4] and Patrignani et al. [96,98] showed how protected modules can be compiled to protected-module architectures without introducing potential software vulnerabilities. As a result protected modules can only be attacked at assembly-level when a related attack exists at source-code level. Later Agten et al. [3] also presented work on how protected modules can be verified at source-code level.
- *How can PMAs communicate securely with peripheral devices, including screen and keyboard?* This is still an open research question. See Section 5.2.1 for more details.
- *Can availability properties be guaranteed to PMA architectures?* This is also still an open research question. Section 5.2.1 also provides more details on this topic.

1.5 Building Secure Subsystems: Contributions to the Research Area

McCune et al.'s approach finally provides verifiable security guarantees to isolated PALs executing op top of an untrusted legacy operating system. This dissertation focused on the problem of developing secure subsystems using the same core principle. Such subsystems are much more complex. Consider as an example a secure payment framework. Application vendors must be able to connect to a single payment module that in turn connects to the payment system of the user's bank or selected payment method. Multiple stakeholders are involved in such a system that do not necessarily trust one another. An application vendor using the payment subsystem requires strong security guarantees that the user transferred the correct amount. Financial institutions may want to provide their own payment module without placing trust in modules of other vendors. Building such subsystems is very challenging as modules cannot be merged.

We present here an overview of our contributions in this research area. We refer the reader to one of the thesis chapters when work has been incorporated in full, or to the published papers.

Self-Protecting Modules: A Noval Isolation Approach. At the beginning of 2010 we came up with a novel isolation mechanism [120]. Unlike Flicker, it does not rely on short-lived modules. Instead, protected modules are treated as "black boxes" by untrusted components; modules execute in the same address space as the rest of the untrusted application, but cannot be accessed from outside the module. They can only be called through an explicit interface and maintain their state between invocations. Obviously, the services themselves do have full access to their own memory locations. This includes the exclusive right to disable their own protection, after the required security measures have been taken to prevent sensitive information from leaking to unprotected memory. Hence the name *"Self-Protecting Modules"* (SPMs).

To guarantee these isolation properties, a program-counter-based access control mechanism was developed. Depending on the value of the program counter (i.e., the location of the currently executing instruction) different access rights are enforced. Obviously, when the program counter points inside an SPM, instructions have more privileges to the SPM's memory region than when we are executing untrusted code outside the module.

Next to the separation between unprotected memory and the SPM, the SPM itself is also divided into two[6] sections. Code implementing the module's services, is placed in the Public section. This section is read-accessible from unprotected memory, but can only be entered via specific entry points. The other section – the Secret section – contains security-sensitive information (e.g., cryptographic keys and the call stack). It is only accessible when executing within the SPM's boundaries.

Secure Inter-SPM Communication. Next to enabling strong isolation of protected modules, the program-counter-based access control mechanism has a second strong property: It enables the easy creation of secure channels between

[6]The original paper [120] presented a solution using three sections: SEntry, SPublic and SSecret. Follow-up work simplified this setup and replaced the SEntry section with entry points to the SPM. To avoid confusion, we present here the simplified setup which will also be used throughout this text.

modules. Even when an attacker gains kernel-level access to the system, integrity and confidentiality of messages exchanged is guaranteed.

SPMs can be authenticated in two steps. First, the world-readable Public section of the SPM is compared to a stored hash value. As the Public section cannot be modified after the module's protection is enabled and it is the only executable section, it prevents SPMs from masquerading their behavior. The access right restrictions guarantee to the caller which code will be executed. In the second step, the layout of the SPM is checked. Containing the correct executable code is not sufficient. Modules with an incorrect layout (e.g., a too short Secret section) may use unprotected memory to store sensitive information. When both checks succeed, the SPM is authenticated and can be called securely by jumping to one of its entry points. Since messages cannot be intercepted and are always delivered, this provides another important building block for secure subsystems.

Early work [87, 119, 120] exposed a public interface to untrusted code. While SPMs are in full control over the sensitive information they store internally, exposing insecure interfaces may still present a major security risk. An SPM implementing a signing service, for example, may guarantee that its private key will never leak to the outside world. When this SPM can be called from unprotected code however, an attacker can still request arbitrary data to be signed.

In follow-up work [115] (see Chapter 3) we showed that a capability-mechanism can be easily built easily on top of the provided secure communication primitive. Only when an SPM received an unforgeable token, is it able to request service. This significantly simplifies building secure subsystem of inter-connected SPMs.

Widely Applicable: High-end and Low-end Prototypes. In contrast to Flicker, the program-counter-based access control mechanism was designed as a suitable solution for low-end systems. The initial paper [120] only described its concept. A working prototype was later developed by Noorman et al. [87]. It showed that the access control mechanism can be implemented in hardware with almost no performance overhead when module boundaries are crossed and at very limited overhead in terms of power consumption and die size. Moreover, it also proved that modules can execute with a zero software TCB. The correct execution of modules can be attested, without relying on any other software component on the platform.

While the program-counter-based access control mechanism requires an access control decision for every memory access, we also showed that it can be implemented efficiently on top of existing, modern x86 processors. Fides [119] (see Chapter 2) takes this approach to execute provably secure SPMs by implementing a small hypervisor. Salus [10] (see Chapter 3) applies a similar mechanism but is implemented in the Linux kernel. It shows that the same isolation principle can also be used to compartmentalize applications where each compartment executes with least privileges.

State-Continuous Execution. Protected-module architectures are able to avoid a large TCB by using untrusted code to execute non-security sensitive tasks. The untrusted TCP/IP stack, for example, can be used to establish a communication channel over a network. By only placing TLS functionality in a protected module, strong security guarantees of the communication channel can be provided with only a limited TCB [119].

Unfortunately, this only holds partially for modules that rely on the untrusted operating system to provide non-volatile storage. Cryptographic primitives can be used to integrity and confidentiality protect data before it is passed to the OS. By deriving the used cryptographic keys directly from the content of the Public section and layout of the module [87], only the module that initially requested storage of sensitive data, can ever retrieve it. However, when freshness of data needs to be guaranteed as well, additional security primitives need to be provided. This is much more challenging than protection against replay attacks, as modules also need to store their own state securely. An attacker with full access to the untrusted store may be able to replay the state of the module itself, not only of received messages.

Isolated, stateful modules need to be able to execute state-continuously. Once input is provided to a module, it should either (eventually) enter a state based on the provided input, or it should never advance at all. This is a non-trivial problem when the platform itself may be powered down during execution or when the execution of modules may be interrupted. Technical limitations present additional challenges. Non-volatile memory, for example, can only be added when a different fabrication process of processors is used. This is infeasible in practice due to huge economic costs. The performance gap between processing speed and time to write-access non-volatile memory, poses another challenge. Especially when the module is used on a time-critical execution path and execution cannot resume until its state is committed. We have proposed a solution that avoids practical limitations, with only very limited performance overhead [116, 117] (see chapter 4).

Fully-Abstract Compilation. Protected module architectures provide strong isolation guarantees of software vulnerabilities external to the module. None of the proposed platforms however, can defend against vulnerabilities in protected modules themselves. Software modules that contain buffer overflows or expose vulnerable interfaces, can therefore still be exploited and sensitive information may still leak to an attacker. To significantly increase security guarantees that PMAs can provide, we need to be able to prove correctness of protected modules.

Developing modules secure from low-level attacks is challenging because reasoning about (possible) attacks requires detailed information about the module's source code, its compilation and memory protection mechanism. Developing modules in a memory-safe language, as proposed in Section 1.2, is only a partial solution as an in-application attacker is able to interact with the module at machine-code level, not only at source-code level. Security measures such as type safety is only enforced at source-code level. Once the source code is compiled to machine code, its imposed restrictions can no longer be enforced – the abstraction introduced by type-safe languages can be broken.

Early 2012 we developed a *fully-abstract compilation* scheme [4]. This entails that for every attack that exists at machine-code level, there also exists a related attack at source-code level. This is an important property because it allows us to reason about module security at source-code level *without* losing generality.

1.6 Independent Research Results by Industry

One simple approach to guarantee isolated execution of software handling sensitive data is to execute this code on a separate chip. Hardware security modules such as the TPM chip take this approach, but this obviously leads to significant hardware costs. ARM developed with TrustZone [140] a cheaper alternative. TrustZone provides a "dual-virtual CPU" design were trusted and untrusted code is executed on the same physical CPU, but each in its own isolated "world". Hardware resources are either duplicated between both worlds or are tagged to indicate to which world they belong to. While this provides strong isolation guarantees between code running in the Secure and Normal World, it does not increase security when malicious code is executed inside the secure world. To mitigate this threat, applications cannot request execution of their own sensitive code inside the secure world. This technology however is used to guarantee security of some very specific applications. Apple's Touch ID technology, for example, stores the user's biometric data within the secure world.

Intel processors equipped with Active Management Technology (AMT) can provide similar security guarantees [88]. In contrast to ARM TrustZone trusted code is not executed on the main processor, but on a physically separate chip called the Management Engine. This enables versatile management functionality such as remote access to the platform even the main processor is not powered on and/or when the system's kernel fails to boot correctly. Similar to ARM TrustZone, execution privileges in AMT mode are heavily restricted and applications cannot request the execution of arbitrary code on the Management Engine.

In June 2013 Intel made its work on an alternative approach called Software Guard eXtensions (SGX) public. Intel SGX has some significant similarities with protected-module architectures described in this dissertation and related work [77, 89]: applications can place security sensitive parts of an application in protected "enclaves" that execute in complete isolation from the rest of the system. Given the strong security guarantees that protected-module architectures (including Intel SGX) provide and the support of a large and influential chip manufacturer such as Intel, we expect that this approach will become widely applied in practice in the near future. This also implies that many of the research results discussed in this dissertation (e.g., fully-abstract compilation, state-continuous execution, etc.) may be applied on a wide scale in the future. Intel SGX is discussed in more detail in Appendix A.

1.7 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2: Fides: Selectively Hardening Software Application Components. In the next chapter we present a two-level architecture. At the lowest level, we discuss the program-counter-based access control mechanism and show by using hardware-based virtualization support that it can be implemented efficiently on commodity processors. At the second level, we briefly discuss how a fully-abstract compiler can be used to guarantee strong security properties of compiled modules.

Chapter 3: Salus: Non-Hierarchical Memory Access Rights to Enforce the Principle of Least Privilege. In Chapter 3 we show that the previously presented memory access control mechanism can also be used to isolate potential attack vectors from likely attack targets. To reduce the potential impact of insecure module interfaces, we also present a capability mechanism: only when a valid access token can be provided, will the caller's request be serviced.

Chapter 4: ICE: A Passive, High-Speed, State-Continuity Scheme. State-continuous execution of modules has largely been neglected by related protected-module architectures. In Chapter 4 we present a solution that does not require accesses to slow TPM NVRAM for every state update.

Chapter 5: Conclusion. In Chapter 5 we re-iterate the contributions of this dissertation and discuss new research challenges for the near and distant future.

Appendix A: Intel Software Guard eXtensions. In June 2013 Intel publicly announced its work on a protected-module architecture called Software Guard eXtensions (SGX). We discuss this technology in Appendix A and highlight similarities and differences with academic research prototypes.

Appendix B: Protected-Module Architectures vs Microkernels. Protected-module architectures have some similarities with microkernels. In Appendix B we discuss key characteristics of microkernels, provide insights why they never got widely adopted and discuss why we expect that protected-module architecture won't share the same fate.

Chapter 2

Fides: Selectively Hardening Software Application Components

Publications & Acknowledgments

Strackx, R., and Piessens, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM conference on Computer and Communications Security* (New York, NY, USA, October 2012), CCS'12, ACM, pp. 2–13.

The main research effort of this work was lead by Raoul Strackx with the guidance of Frank Piessens.

Abstract

Protecting commodity operating systems against software exploits is known to be challenging, because of their sheer size. The same goes for key software applications such as web browsers or mail clients. As a consequence, a significant fraction of internet-connected computers is infected with malware.

To mitigate this threat, we propose a combined approach of (1) a run-time security architecture that can efficiently protect fine-grained software modules

executing on a standard operating system, and (2) a compiler that compiles standard C source code modules to such protected binary modules.

The offered security guarantees are significant: relying on a TCB of only a few thousand lines of code, we show that the power of arbitrary kernel-level or process-level malware is reduced to interacting with the module through the module's public API. With a proper API design and implementation, modules are fully protected.

The run-time architecture can be loaded on demand and only incurs performance overhead when it is loaded. Benchmarks show that, once loaded, it incurs a 3.22% system-wide performance cost. For applications that make intensive use of protected modules, and hence benefit most of the security guarantees provided, the performance cost is up to 14%.

2.1 Introduction

A significant fraction of Internet-connected computers is infected with malware, usually with kernel-level access. Yet, many of these computers are used for security-sensitive tasks, and handle sensitive information such as passwords, corporate data, etc. While efforts to increase the security of commodity operating systems [109] and applications [26,41,125] are important and ongoing, their sheer size makes it very unlikely that they can be made secure enough to avoid infection [6,42] with kernel-level malware. Due to the layered design of commodity systems, kernel-level malware can break the confidentiality and integrity of all data and code on the system.

This unfortunate situation has triggered researchers to design systems that can execute security sensitive code in an isolated area of the system, thus improving the security guarantees that can be offered. Of course, an important design goal (and design challenge) is to realize this while remaining compatible with current operating systems and hardware. Most of these proposed systems leverage recent hardware extensions for trusted computing or virtualization to execute code, and differ in the granularity of protection they offer (protection of full applications [26,41,47,75,125] versus protection of small pieces of application logic [11,77,78,105,112]), and in their root of trust (a correctly booted system [47, 112] or a hardware security module such as a TPM chip [11,62,77,78,105]). We provide a more complete overview of existing work in section 2.6.

State-of-the-art systems for protection of software modules focus on attesting the correct and secure execution of a single module to a third party [11,77,78,105]. We focus on the complementary case of increasing the security guarantees of applications for the owner of the system. We propose an approach to selectively harden security-critical parts of an application. An SSL-enabled webserver, for example, could be built in a modular way where sensitive information is passed between trusted modules until it is finally encrypted and passed to the TCP/IP stack. This would reduce the power of a kernel-level attacker to one with only access to the network. Current systems are ill-equipped for this task: writing co-operating protected modules is too hard, messages passed between modules may never be delivered and a lack of support for multiple instances of the same module prevents a modular application design.

In this chapter, we propose a system consisting of two parts: a run-time security architecture and a compiler. The security architecture implements a program-counter based access control model. A protected module is divided into a public and a secret section. The secret section stores the sensitive data and is only accessible from within the module. The public section contains the module's code and can be read from outside of the module. This enables authentication and secure communication between modules in a cheap and secure way: an attacker is not able to intercept, modify or masquerade any messages between protected modules.

The compiler provides an easy way to compile standard C-code into protected modules. Since the program-counter dependent access control model allows modules and unprotected code to share the same virtual address space, their interaction is straightforward. This significantly simplifies the hardening of applications.

Modules compiled with our compiler effectively reduce the power of kernel-level malware and in-process attackers to only being able to interact with the modules through a public API. In earlier work [4] we have proven for a simplified model of our access control mechanism and compiler that with a proper API design and implementation the module is fully protected: an attacker that is able to inject arbitrary assembly code at kernel-level is only as strong as an attacker interacting through the module's API.

More specifically, we make the following contributions:

- We propose *Fides*, a security architecture for fine-grained protection of software modules, based on a memory access control model that makes access privileges dependent on the value of the program counter (instruction pointer). The access control model is strong enough to support fully abstract compilation [1] of modules; low-level attacks against a compiled module exist if and only if the source-level module can also be exploited.
- We show how this access control model supports novel features, such as

1. the ability to support function pointers to trusted modules. Secrecy and integrity of any data passed as arguments is ensured by the authentication of the pointer's destination.
2. the ability to update modules after they are deployed, thereby allowing legacy software to be ported easily and incrementally with minimal modification.

- We report on a fully functioning prototype implementation, demonstrating that Fides can be implemented on commodity hardware while remaining fully compatible with legacy systems.
- We present a compiler that compiles standard C source code modules into protected binary modules.
- We show that Fides has an average performance overhead of around 3% on the overall system, which is reduced to 0% when no modules are loaded. Macrobenchmarks show an overhead of up to 14% for applications that intensively use Fides' services.

We do not consider trusted I/O and leave it as future work. However, a trusted path between an I/O module and I/O devices can be established as in related work [80, 150].

The remainder of this chapter is structured as follows. First, we clarify our objectives by defining the attacker model and desired security properties in Section 2.2. Section 2.3 gives an overview of the security architecture and its key concepts. In Sections 2.4 and 2.5, we discuss how the run-time system and compiler were implemented and evaluate performance. We finish with a discussion of related work and a conclusion.

2.2 Objectives

High-level programming languages offer protection facilities such as abstract data types, private field modifiers, or module systems. While these constructs were mainly designed to enforce software engineering principles, they can also be used as building blocks to provide security properties. Declaring a variable holding a cryptographic key as private, for example, prevents direct access from other classes. This protection however does not usually remain after the source code is compiled. An attacker with in-process or kernel level access is not bound by the type system of the higher language. We will show that Fides *is* able to provide such strong security guarantees. We first discuss the abilities of an attacker and then discuss how Fides provides these guarantees.

2.2.1 Attacker Model

We consider an attacker with two powerful capabilities. First, an attacker can execute arbitrary code – user-level or kernel-level – in the legacy operating system. This kind of root-level access is a realistic threat: legacy operating systems consist of millions of lines of code and this unavoidably leads to the presence of programming bugs, such as buffer overflows [6], that can be exploited by an attacker to inject code [42, 146].

With kernel-level privileges, the attacker can try to corrupt or read the state of protected modules, modify the virtual memory layout of applications containing protected modules or intercept their loading process to tamper with security-sensitive code or data. The attacker can also try to intervene in the communication between modules, or to attack data that protected modules wish to store persistently.

Second, the attacker can build and deploy her own protected modules. Our security architecture does *not* assume that software modules that request protection can be trusted. In other words, it is our goal to ensure the security of a protected module by one stakeholder, even if modules of malicious stakeholders are also loaded in the system.

With respect to the cryptographic capabilities of the attacker, we assume the standard Dolev-Yao model [37]: cryptographic messages can be manipulated, for instance by duplicating, re-ordering or replaying them, but the underlying cryptographic primitives cannot be broken.

We assume the attacker has *no physical access to the hardware.* An attacker with control over the physical system may disconnect memory, place probes on the memory bus, or perform a hard reset. Since remote exploitation of commodity systems is far more common than exploitation through physical access, this is a reasonable assumption.

2.2.2 Security Properties

To provide strong security guarantees, we use a combination of a run-time system and a compiler.

The Fides Run-Time System

The Fides run-time system enforces a program-counter based access control mechanism. It guarantees the following security properties:

- *Restriction of entry points.* Protected modules can only be invoked through specific entry points, preventing an attacker from jumping to an incorrect location in the module and executing on unintended execution paths [111].
- *Confidentiality and integrity of module data.* A protected module can store sensitive data in a way so it can only be read or modified by the module itself.
- *Authentication of modules.* Modules can authenticate each other securely. This also implies that code of modules is integrity protected.
- *Secure communication between modules.* Fides guarantees integrity, confidentiality and delivery of data exchanged between modules.
- *Minimal TCB.* The correct and secure execution of a module only depends on (1) the hardware, (2) the Fides architecture and (3) the module itself and any other module that it calls. In particular, the operating system is excluded from the TCB.

Secure Compilation of Modules

Using the protection mechanisms offered by the run-time system as building blocks, the compiler allows the compilation of standard C source code into protected modules. It provides the following security guarantees:

- *Integrity of execution.* An attacker is not able to influence the correct execution of the module
- *Secure communication between modules.* The compiler ensures that sensitive information is passed only between modules using a secure channel.
- *Secrecy of sensitive information.* Only information that is passed explicitly to unprotected memory or to another module exits the module. Leakage of possibly sensitive information, for example information lingering in save-by-caller[1] registers, is prevented.

[1]Storing and restoring the contents of all registers for every function call would result in a significant performance impact. Instead, the application binary interface (ABI) defines which CPU registers need to be stored/restored by the caller of the function (i.e., save-by-caller registers), which need to be stored and restored by the called function (i.e., save-by-callee registers) or which do not need to keep their contents (i.e., scratch registers).

Note that it is *not* our objective to protect against vulnerabilities in protected modules: the security of a protected module can be compromised if there are exploitable vulnerabilities in its implementation. Examples include logical faults (i.e. a faulty API design [73]), or memory errors [42, 122, 146]. Instead, our goal is to protect the module from malware that exploits vulnerabilities in the surrounding applications or underlying operating system. A vulnerable module however, can only affect the security of other modules if they explicitly place trust in the former and, for example, exchange sensitive information. An attacker introducing malicious modules in the system does *not* gain any more power as they are not trusted by any other module. In section 2.5 we will show that a low-level attack against modules exist iff also a high-level attack exists.

2.3 Overview of the approach

In Fides, an application and the protected modules it uses, share the same virtual address space. Protection of the modules is provided by enforcing a memory access control model: access rights to memory locations depend on the value of the program counter. Roughly speaking, while the processor is executing within the boundaries of a specific protected module (i.e. the program counter points to an instruction that is part of the module), it can access memory allocated to that module. If the processor is executing outside the boundaries of the module, it has only limited rights to access the module's memory. In particular, destruction of the module is also only possible from within the module: this is why we use the term *self-protecting module (SPM)* for protected modules in Fides.

This section gives an overview of how this basic mechanism is used to support communicating protected modules. First, we discuss in more detail the layout of an SPM and the enforced memory access control (Section 2.3.1). Next, we describe the four primitive operations offered by Fides (Section 2.3.2). Then, in Sections 2.3.3 and 2.3.4, we discuss the typical life cycle of an SPM, and how SPMs can authenticate each other and collaborate securely. Sections 2.3.5, and 2.3.6 respectively show how SPMs can be updated and how SPMs can persistently store their state over reboots. Finally, attestation of the correct execution of an SPM to a remote verifier is addressed in Section 2.3.7.

2.3.1 Layout of a Self-Protecting Module

A self-protecting module is a chunk of memory divided in two sections (see Figure 2.1). The *Secret* section contains the module's sensitive data. This not

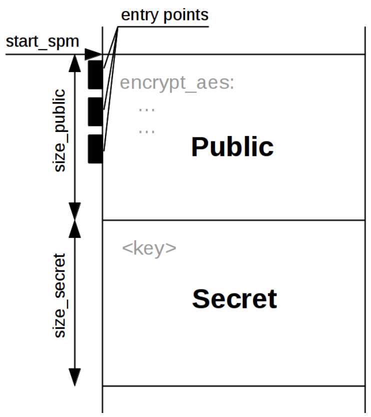


Figure 2.1: The layout of an SPM.

Table 2.1: The enforced memory access control model.

from\to	Entry pnt.	Public	Secret	unprot.
Entry pnt.	`r-x`	`r-x`	`rw-`	`rwx`
Public	`r-x`	`r-x`	`rw-`	`rwx`
Secret	`---`	`---`	`---`	`---`
Unprot./ other SPM	`r-x`	`r--`	`---`	`rwx`

only includes sensitive data processed by the module, such as cryptographic keys, but also data used for the correct execution of the module, such as its call stack. Read and write access to the Secret section is only allowed from within the module. Access from outside the module, including from another instance of the same SPM, is prevented. These access restrictions give modules total control over any data stored in their Secret section.

The *Public* section contains information that is non-confidential and should only be integrity-protected. This includes all[2] the module's code and constant values such as strings. Once the module is protected, the contents of this section can no longer be modified, it can only be read and/or executed. Read access is allowed from unprotected code as well as from other SPMs, allowing authentication of modules.

[2]Self-modifying code and interpreted code could be supported easily by making the Secret section executable, but we consider such support out of scope for this chapter.

SPMs are able to access unprotected memory in the same address space. While Fides' design does not impose any access limitations on these locations, access restrictions set by the legacy kernel are enforced to prevent malicious modules, for example, from overwriting kernel space.

Each SPM comes with a list of memory locations in the Public section that are valid entry points into the SPM. Fides will guarantee that an SPM can only be executed by jumping to a valid entry point. This prevents attacks that attempt to extract information by selectively executing code snippets [111].

Table 2.1 summarizes the enforced access control rules. It shows for instance that code in unprotected memory or other SPMs can read the Public section of an SPM, or can execute an address that is an entry point of the SPM (from this point on, the program counter is within the Public section and the access rights are elevated).

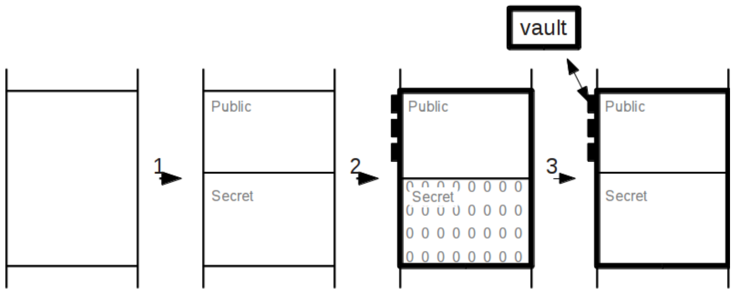
2.3.2 Primitive Operations

Fides implements four primitive operations to create, destroy and query the location and layout of SPMs.

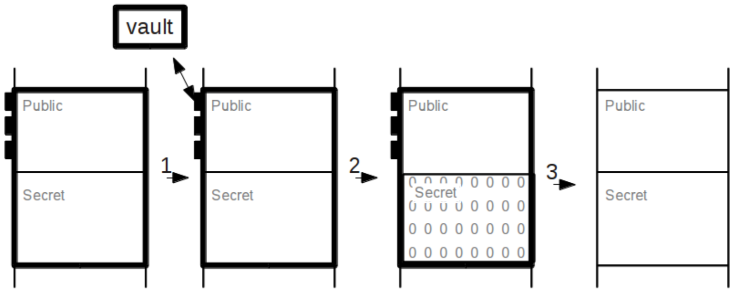
The `crtSPM` primitive is used to create an SPM. It takes the location and size of the Public and Secret sections and a list of entry points. First, Fides verifies whether all referenced logical pages are mapped to physical pages, that they do not overlap with any existing SPMs, and that all entry points point into the Public section. Then, Fides creates an identifier `spm_id` for the SPM. Fides guarantees that until it is rebooted, no other SPM will receive the same identifier. Therefore the identifier can be used to differentiate instances of the same module. Fides also clears the Secret section (set to all zeroes) to remove the initial contents of the Secret section from the attack surface. Finally, memory access protection of the SPM is enabled.

The `killSPM` primitive will destruct the SPM that called it (or generate a fault if called from unprotected code). Enforcing that only SPMs can destruct themselves is important for security: it prevents attacks where an attacker destroys an SPM unexpectedly e.g. during a callback to unprotected memory. It also allows SPMs to pause their destruction until its sensitive data is stored securely to disk and overwritten in memory. Note that this does not prevent Fides from interrupting non-responsive, possibly malicious, modules (see section 2.4.1).

Fides supports two primitives to allow authentication of SPMs. The `lytSPM` primitive is given any virtual address and returns the base address, layout and



(a) Initialization



(b) Destruction

Figure 2.2: The life cycle of an SPM from (a) initialization to (b) destruction.

`spm_id` of the module that is mapped at the specified location. The `tstSPM` primitive is more efficient and returns whether the SPM with a given `spm_id` is loaded starting at the specified location. Both primitives check whether the referenced SPM is loaded correctly: as SPMs are loaded in processes' virtual address space, pages may not be mapped, mapped to incorrect physical pages, or mapped out of order.

2.3.3 Life Cycle of a Self-Protecting Module

Figure 2.2a and 2.2b describe the life cycle of an SPM. We explain the steps from creation to destruction in detail.

Setting up an SPM. First (Figure 2.2a, step 1), the legacy operating system provides a user process with a chunk of (possibly physically non-contiguous) memory and the SPM is placed in unprotected memory.

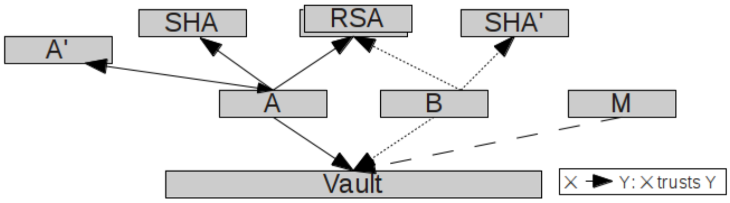


Figure 2.3: Fides' ability to establish secure channels, enables the easy creation of complex trust networks. Modules A and B are able to explicitly place trust on (possibly another instance of) an SPM implementing RSA operations without trusting each other. Similarly, a malicious module M will not be able to cause any harm as it is not trusted by any module.

In the second step, the `crtSPM` primitive is called. An attacker that compromised the previous step(s), will be detected later on and mitigated. At this point, the SPM can be authenticated and service other modules. However, most SPMs will need to restore their secret state from persistent storage after creation (step 3). In Fides, this is handled by a special SPM, called *the Vault* that will only return previously stored data over a secure channel. Details of authentication and secure communication will be discussed in Section 2.3.4. In Section 2.3.6 we elaborate on the workings and security of the Vault.

Destroying an SPM. When the SPM is no longer needed, it should be destroyed properly (Figure 2.2b). First, the module accesses the Vault to store any secret data that must be available for later executions. In step 2, the secret data of the SPM is overwritten to prevent it from being disclosed in unprotected memory. Finally, the module calls the `killSPM` primitive to lift the imposed access control of the module's memory.

2.3.4 Secure Local Communication

One of Fides' objectives is to support a system of collaborating modules (see Figure 2.3), each with its own secrets and services that it offers to other modules, adhering to the principle of least privilege [106]. Hence, SPMs must be able to authenticate each other, and establish secure communication channels. We explain both aspects in detail.

The identity of an SPM is captured in what we call a *security report*. It contains four parts:

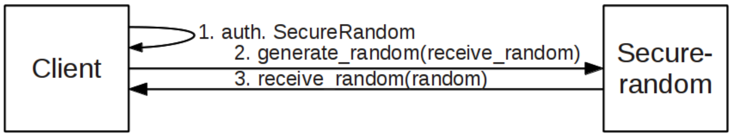
- *A cryptographic hash of the Public section* allowing verification that the Public section was not compromised before protection was enabled. It is essentially the SPM's identity as the access control model only allows code execution from this section.
- *The layout of the SPM, including the sizes of the Public and Secret sections, and the list of entry points (relative to the Public section)* to verify that the protection request was not compromised. Modification of the size of the Secret section in the SPM's initialization phase for example, may cause the use of unprotected memory to store sensitive information.
- *A serial and version number.* The authentication mechanism is flexible enough to support SPMs to be updated easily. As the cryptographic hash of the Public section will differ between versions, a serial number is required to link different versions of the same SPM together. A version number prevents the re-use of old (e.g. security vulnerable) SPMs.
- *Cryptographic signature.* The security report is signed by its issuer. SPMs have a list of trusted certificate authorities (CAs) to verify the signature of SPMs they authenticate.

Since a security report is signed by its issuer, it can be stored in unprotected memory. Any working mechanism to retrieve an SPM's security report will suffice. For simplicity, we will assume in the rest of the chapter that it is stored in front of the SPM.

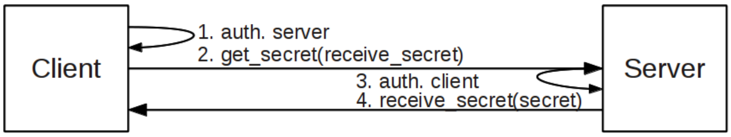
One-way authenticated service call. Consider a *SecureRandom* module that provides cryptographic random numbers, and a *Client* module that authenticates and requests its service (Figure 2.4a).

First, the Client calls the `lytSPM` primitive, locates the security report, and verifies (1) its signature on the security report, (2) the hash of the Public section and the layout of the SPM against the information in the security report, and (3) whether serial and version number are as expected.

Second, the `SecureRandom` module is called. This is similar to calling a function: parameters are loaded in registers and a jump to the appropriate entry point is performed. An important difference with regular function calls on the x86 platform is that the return address must also be passed in a register. Under normal operation return addresses are pushed on the call stack of the caller. However, to protect the integrity of their execution, modules are not allowed to access each others call stack and the return address cannot be retrieved. Hence, a continuation entry point, in this case `receive_random`, is provided as a parameter (similar to continuation-passing-style programming [8, 101]).



(a) One-way authentication



(b) Two-way authentication

Figure 2.4: Communication protocols between two SPMs.

In the final step, `SecureRandom` generates a random number and returns it by performing a jump operation to the `receive_random` entry point.

The bandwidth of the secure channel can be increased significantly by storing large messages in memory shared between sender and receiver. We will further discuss this mechanism in Section 2.4.1.

In case the Client module requires any additional random numbers, the `SecureRandom` module can be re-authenticated using the `tstSPM` primitive. Based on the `spm_id` returned by `lytSPM` when the module was first authenticated, it ensures that the *same instance* of the module will be accessed and rechecking the security report is not required. We will show in Section 2.5 that repeated authentication is significantly more efficient than initial authentication.

Two-way authenticated service call. Two-way authentication is very similar. Assume that a module Client wishes to communicate with another module, Server, and that mutual authentication is required (Figure 2.4b).

First, Client locates the Server's security report and authenticates the module as in one-way authentication. In step two, a message is sent to the Server containing the entry point, `receive_secret`, where the response should be returned to. In step three, the Server locates the Client's security report using the provided return point and the `lytSPM` primitive. Only after successful authentication of the client, sensitive data is returned.

In case the origin of service requests must be proven, a secret session token can be passed to the authenticated endpoint during the initial authentication. The session-token should be passed in all future service requests.

2.3.5 Updating SPMs

Fides' authentication scheme allows a module to be updated easily without requiring any modification of modules or unprotected code that are clients of the updated module.

Updating works as follows. A client authenticates a module starting from just a function pointer: using the `lytSPM` primitive, the SPM queries Fides for the base address of the referenced module and locates its security report. After authenticating the issuer and verifying its serial number[3], the module is authenticated as described in Section 2.3.4. In addition the client should check the version number. Updated versions might contain API inconsistencies to previous versions e.g. services may be serviced on new entry points. Similarly, the version number should be high enough to prevent attacks where a module is downgraded to an older, vulnerable version. To transfer secret state from an old version to a newer, special support is required. An update protocol could be implemented by the modules to pass the information, or a support SPM could be built to pass persistent sensitive information to updated SPMs. This approach allows the Vault to remain simple and easy verifiable: it will only return sensitive data to the same SPM that previously requested storage.

Security of updating modules depends on the ability to create crytographically signed security reports. Since it can safely be assumed that only the creator of the initial report has access to the private key, an attacker is not able to fabricate his own new versions.

2.3.6 The Vault

The Vault is an SPM that stores sensitive information on behalf of other SPMs. It offers two services. First, an SPM can ask the Vault to store persistent secret data. The Vault will append the identity of the requesting SPM (its layout and cryptographic hash of the public section), encrypt and sign the data and store it using the (untrusted) services of the legacy operating system.

Second, *only* an SPM that previously stored secret data can retrieve it again. After mutual authentication, the Vault retrieves the encrypted data from the legacy operating system, checks its integrity, decrypts it and sends it over a secure channel to the requesting SPM.

[3]The issuer/serial number combination is assumed to uniquely determine functionality, and this should be stable over updates of the module.

The Vault is treated specially by Fides: it is created when Fides is booted, and it receives its own secret data directly from the secure storage space on the TPM.

Besides offering secure storage, the Vault also ensures state continuity [93]. In particular, protection against two possible attacks is provided. First, in a rollback attack, an attacker passes a stale version of an SPM's stored data from disk to the Vault. Depending on the module's functionality, this may result in a security vulnerability, such as the reuse of cryptographic keys.

Second, the Vault should also provide crash resilience. As a compromised legacy kernel may allow an attacker to cause the system to crash, persistent storage of fresh data could be prevented based on subtle timing differences. This essentially enables an attacker to prevent the system from making progress.

In chapter 4 we discuss in detail how state-continuous execution and storage can be provided efficiently.

2.3.7 Remote Attestation

Fides' access control model and local communication mechanism can also be leveraged to attest correct execution of modules with two key characteristics. First, meaningful attestation can be provided to the remote party, called the verifier. Only a small TCB consisting of the Fides architecture, an attestation module, the Vault and the attested module(s), are included in the measurement. Second, the attestation is transparent: the correct execution of any module can be attested without *any* modification. This improves reusability of modules.

Attestation in Fides is based on a two-layered approach where each layer attests its correct execution. Due to page constraints, only a sketch of the mechanism is presented. It relies however on μTPMs presented by McCune et al. [77]. Interested readers are referred there.

At the lowest level, the TPM chip ensures the correct loading of Fides and boots trust on the next layer. To achieve this, PCR registers 17, 18 and 19 are extended with a measurement of Fides, the security report of an attestation module and the Attestation Identity Key (AIK_{SPM}) respectively.

At the second level, *attestation modules* provide an attestation service and implement PCR extend and quote functionality similar to a hardware TPM chip. This prevents hardware PCR registers from being cluttered. As several identical attestation modules can also be loaded in the system, the number of SPMs that can be attested at the same time is virtually unlimited.

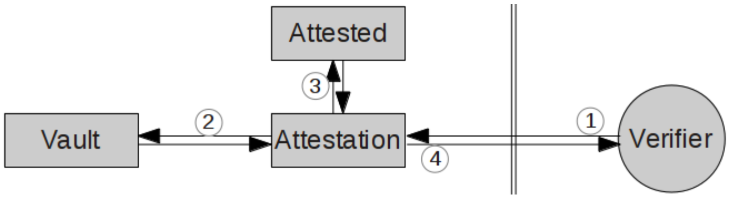


Figure 2.5: Using Fides' access control model and fast local communication, attestation can be supported easily and transparently to *any* attested module.

Figure 2.5 displays how the correct execution of an *Attested* SPM can be proven. First, the verifier provides two nonces n_1 and n_2 and an attestation module is created. Next, the attestation module extends its measurement with the Vault and requests its AIK$_{SPM}$ key. Similarly the attested module is measured and contacted in step 3. Finally, the attestation module extends its measurement with the received result and signs it together with n_2. A similar request is sent to the lower level with n_1, but is only granted when the request is made from a module compliant with the measured security report in PCR 18. In step 4, both quotes are sent to the verifier.

In case the attested SPM calls other SPMs, the verifier is able to rely on the authenticated communication mechanism to ensure that no untrusted SPMs are used in the computation of the result. Alternatively, attestation-aware SPMs could notify the attestation module which SPMs are used.

2.4 A Prototype Implementation

A key element of Fides is the program-counter dependent memory access control model. Since access rights have to be checked on each memory access, implementing this completely in software would have a huge performance cost. Alternatively, modifying hardware, an approach taken by related research [40, 70, 139], has serious drawbacks. In this section, we describe an efficient implementation of the run-time system and the compiler on readily available hardware.

2.4.1 The Fides architecture

The key observation is that the memory access control rules only change when entering and exiting SPMs. In our implementation, we use a small dynamic hypervisor to isolate SPMs from the rest of the system, and we ensure that the

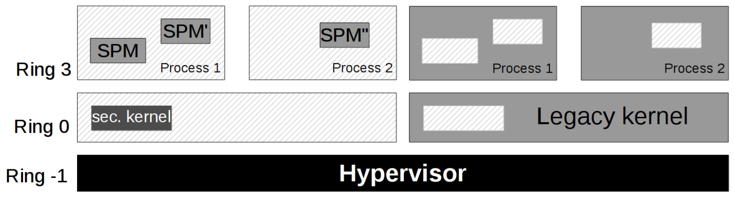


Figure 2.6: Layout of the Fides architecture. Hatched areas represent partially accessible memory regions.

correct memory permissions are set on entering/exiting SPMs. Hence, there is only an overhead on entering and exiting SPMs, leading to a reasonable performance overhead.

We introduce a minimal hypervisor that runs two virtual machines, the *Secure VM* and the *Legacy VM* (see Figure 2.6). Both VMs have the same guest physical view of host physical memory, but they have a different configuration of memory access control. Note that there is no duplication of memory, only two virtual views of the same physical memory.

Our prototype implementation can be loaded and unloaded when required, avoiding any overhead when no SPMs are in use. Fides is bootstrapped by loading a device driver in the legacy kernel to gain supervisor privileges. Then, physical contiguous memory is allocated to store a hypervisor and the Secure VM. Next, a dynamic root of trust is started and the hypervisor and Secure VM are launched. Finally, the running legacy kernel is pulled in the Legacy VM, and memory access control of both VMs is configured [63, 103].

The Legacy VM. The legacy kernel and user applications continue their operation without any interruption: the only difference after the start of Fides is that access to certain parts of memory is now prohibited in the Legacy VM. More specifically, the memory where SPMs and parts of the TCB are stored, is protected. If the legacy VM accesses this memory (for instance tries to read or write the Secret section of an SPM), this will trap to the hypervisor and the access attempt is prevented.

The Fides device driver that was used to bootstrap Fides also provides an interface to the security architecture, for instance to create and query SPMs. This interface can not be exploited: no sensitive information is ever returned. However, given our attacker model, an attacker may change the returned results before code in the Legacy VM can process it. Hence, these primitives can only be used securely from within an SPM thereby avoiding the results to leave the Secure VM. This problem does not occur when SPMs are created from unprotected memory: when an attacker interfered with the creation of an SPM, this will be detected by the authentication protocol and no sensitive information will be passed to it.

This design ensures excellent compatibility with legacy operating systems: the only change from the OS's viewpoint is that certain memory regions (that are unused during normal operation) are rendered inaccessible.

Hypervisor. The hypervisor is executing at the most privileged level and serves three simple purposes. First, it provides coarse-grained memory isolation of the legacy VM, secure VM and itself. It also prevents any access to SPMs that is not allowed from unprotected memory. However, it does not implement the fine-grained program-counter dependent memory access control. This is implemented in a separate security kernel in the secure VM and will be discussed later.

Second, the hypervisor schedules both virtual machines for execution based on a simple request passing mechanism. The secure VM is scheduled only when a request is passed to it (i.e. when an SPM is called), or when it did not yet finish executing the previous request. Hence, the Secure VM consumes no CPU cycles when no SPMs are being executed.

Third, the hypervisor creates a new dynamic root of trust (DRTM a.k.a. late launch) when it is loaded. This allows the attestation of the correct launch of the security architecture. It also allows the TPM chip to store sensitive information based on this measurement, such as the cryptographic keys used by the Vault. If Fides was compromised before it was protected in memory and launched or a hypervisor was already present, the result of this measurement differs and sensitive data is inaccessible.

Secure VM. To allow easy access to the unprotected memory, the Secure VM has the same view of physical memory as the Legacy VM but with different access control settings: SPMs can be accessed but are protected by a security kernel

Security kernel. To reduce the size of the TCB, only a minimal amount of features are used: memory paging, a separation of user and kernel mode, page fault handling and a few system calls. We now discuss how these features are used to enforce the fine-grained access control model and how SPMs can use Fides' primitive operations.

To ensure isolation, SPMs are executed in user mode. When a module is invoked, the security kernel receives a request specifying the virtual address of the entry point called. This address is translated to a physical address by directly traversing the (untrusted) page tables in the Legacy VM. Next the containing module is located. When no module is found an error is returned to the Legacy VM, else a new address space is created mapping the entire module. As modules are always mapped at the same virtual addresses as in the Legacy VM, it is easy to access unprotected memory locations. When these are not yet mapped, a page fault will be generated. At that time the referenced physical page is located, checked whether it is part of an SPM and checked against the access rights of the SPM. To prevent an SPM from receiving unauthorized access to memory locations, the address space is rebuilt each time an SPM is invoked. Note that the page tables of the Legacy VM are *not* trusted: they are only used to check which physical page was referenced.

The security kernel also ensures that modules are properly loaded: since the untrusted page tables of the legacy kernel are used, an attacker may try to only load modules partially in memory or rearrange the order of its pages. To mitigate this threat, the security kernel records the order of the *physical pages* when a module is created and ensures that the same order is used when the module is called or its presence tested using the `lytSPM` and `tstSPM` primitives.

TOCTOU attacks are mitigated by preventing concurrent execution of modules. As modules can only destroy themselves, an authenticated module must still be mapped in memory when it is called. This is however overly restrictive as only the presence of the called module must be ensured.

Besides passing information between SPMs in registers, the security kernel also provides support to pass bulk data using a special shared memory segment that is accessible only to the currently executing SPM. Hence, the receiver automatically gains access when it is called. To prevent information leakage, the called SPM must overwrite the passed data before execution returns to unprotected code. Access to this memory segment from the legacy VM is prevented using Extended Page Tables (EPT), the same hardware mechanism used to isolate different VMs.

Limitations of the prototype implementation. To prevent time-of-check-to-time-of-use attacks, SPMs must not be destroyed after they were authenticated but before they are called. This would cause sensitive information stored in registers to leak to untrusted code. Our prototype currently handles this by preventing SPMs to be interrupted. However, this is largely a prototype limitation, and *not* fundamental. Fides could, for example, support interrupts by suspending and resuming the executing SPM after the interrupt is handled.

Entries to other SPMs are denied to prevent destruction of already authenticated communication endpoints by the interrupted module. Non-responsive modules on the other hand, may be destroyed by the security kernel without further consideration. Alternatively, support for multicore processors could also be added and SPMs can be run on one specific core. In that case, unprotected code is able to execute uninterrupted. This may be acceptable, as it can be expected that SPMs are only responsible for a small fraction of all computation.

In production systems the use of DMA should be prevented from overwriting an SPM. Just as the prototype currently prevents the Legacy VM to access SPM locations, the IOMMU should be configured to prevent DMA accesses to modules.

2.4.2 Automated Compilation of Modules

We modified the LLVM[4] compiler to compile standard C source code modules into protected modules. More specifically, the compiler ensures the following:

- Each module implements its own stack.
- When returning to unprotected memory, registers and condition flags are cleared.
- Function pointers point to unprotected memory or to a function in the SPM with a correct signature.
- Function call annotations specify that the referenced module must be authenticated before the function is called and possibly sensitive information is passed.
- The entry point handling returns from callback functions, cannot be exploited. The entry point is only serviced when a callback actually took place.

We now discuss two notable implementation details: supporting function calls to SPMs and the use of function pointers by modules.

Supporting function calls to SPMs. For each SPM, a wrapper is created to allow easy invocation. The wrapper serves two purposes. First, it loads and unloads the SPM when appropriate. Second, it creates a stub function for each

[4]http://llvm.org/

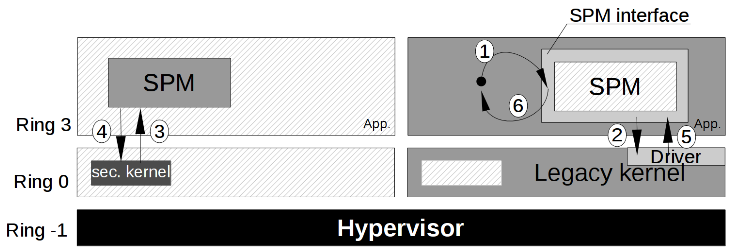


Figure 2.7: Implementation of the Fides architecture.

available entry point. Figure 2.7 displays a schematic overview of an invocation. In step 1 untrusted code accesses a stub in the SPM's interface as a normal function. Arguments are passed together with the entry point to the security kernel via the Fides driver and hypervisor (step 2). After all consistency checks pass, the SPM is invoked (step 3). The SPM's execution stops when it tries to execute unprotected memory, either because the SPM's service returns or because an external function is called. In both cases the security kernel returns the contents of all registers to the stub (steps 4-6). There appropriate action is taken: returning to its caller or invocation of the function pointer before re-entering the SPM.

Supporting function pointers. Support for function pointers dereferenced within a module is added in two steps. In the first step, an LLVM function pass replaces every function pointer dereference with a call to one of two support functions, depending on whether the call should be made to a trusted module or unprotected code. The developer of the module should specify the type of the target of the function pointer by annotating the source code. These support functions will be compiled as part of the module and allows easy handling of function pointers without having to use the LLVM intermediate representation. In step two, the module of the target of the function pointer is authenticated, if required. The `spm_id` of the destination is stored within the boundaries of the module to limit the cost of subsequent calls. Finally, possible side channels that leak information about the inner state of the module, such as the condition flags and save-by-caller registers are cleared and the value of the stack pointer is stored before a jump to the function pointer is made. A special return entry point is added to the module (one per module) to facilitate returns from the function. As with any entry point, the return location after the module has serviced the request, is passed through register `%rbp`.

Table 2.2: The TCB consists of only 7K lines of code.

VMM	Secure kernel	Shared	Total TCB
1,045	1,947	4,167	7,159

Compiler limitations. While modules can be written in standard C-code, some source code annotations are required. Allocation requests using the standard `malloc` function, for example, must specify where the memory should be allocated. A support library has been created to statically link functions of the libc library with the module.

2.5 Evaluation

2.5.1 Security Evaluation

Fides offers strong security guarantees to compiled modules by relying on a run-time system of only limited size and a compiler. We now discuss both components.

Run-time system. The run-time system implements a program-counter based access control mechanism in three layers. At the lowest level we rely on the TPM chip and assume that an attacker is not able to launch physical attacks against the system. When the Fides architecture is loaded, a dynamic root of trust measurement (DRTM) is started, measuring the memory state of the system. Based on this measurement the cryptographic keys used by the Vault are sealed in the TPM. An attacker that compromised the correct loading of Fides, for example by modifying the binary on disk, will cause an incorrect measurement and access to the sealed keys is prevented. As the Vault is the only SPM that stores persistent data, access to sensitive data is prevented.

In the second layer, the hypervisor protects all security sensitive memory locations against faulty legacy applications and a compromised kernel. This includes secrecy of confidential data as well as integrity of code.

The third layer, the security kernel, protects modules from potentially malicious SPMs by realizing the program-counter dependent access control model.

An important enabler for formal verification of the TCB is to make sure that the size of the TCB is small. Table 2.2 displays the TCB of Fides for its different parts, as measured by the SLOCCount[5] application. Only the hypervisor (VMM) and the secure kernel are trusted. They contain 1,045 and 1,947 lines of C and assembly code respectively. This does not include the 4,167 lines of code that is shared between the parts. The driver (690 LOC) used to support communication with Fides is not security sensitive and thus is excluded from the TCB. This totals the size of the TCB to only 7,159 LOC.

Compilation of SPMs. Facilities offered by high-level languages such as a private field modifier, allow easy reasoning about an application's security guarantees and its verification. A low-level attacker, not bound by these restrictions, may however still exploit a vulnerability anywhere in the application and break these security guarantees. It has been proven [4] that a simplified version of Fides' fine-grained access control mechanism, is able to support fully-abstract compilation of a high-level language with private fields: when no source-level attack against a module exists, it also can't be exploited after compilation.

To achieve such high security guarantees, modules must be compiled securely. Each module's stack, for example, must be placed in its Secret section. Our modified compiler is able to compile standard C source code to modules meeting these requirements.

2.5.2 Performance Evaluation

We performed three types of performance benchmarks on our prototype. First, we measure the system-wide performance impact of Fides. Next, we measure the cost of local communication (Section 2.5.2) and finally we benchmark an TLS-enabled web server (Section 2.5.2).

All our experiments were performed on a Dell Latitude E6510, a mid-end consumer laptop equipped with an Intel Core i5 560M processor running at 2.67 GHz and 4 GiB of RAM. Due to limitations of our prototype, we disabled all but one core in the BIOS. An unmodified version of KUbuntu 10.10 running the 2.6.35-22-generic x86_64 kernel was used as the operating system.

System-wide performance cost

To measure the performance impact of Fides on the overall system, we ran the SPECint 2006 and lmbench benchmarks. Figure 2.8a displays the results of the

[5]http://www.dwheeler.com/sloccount/

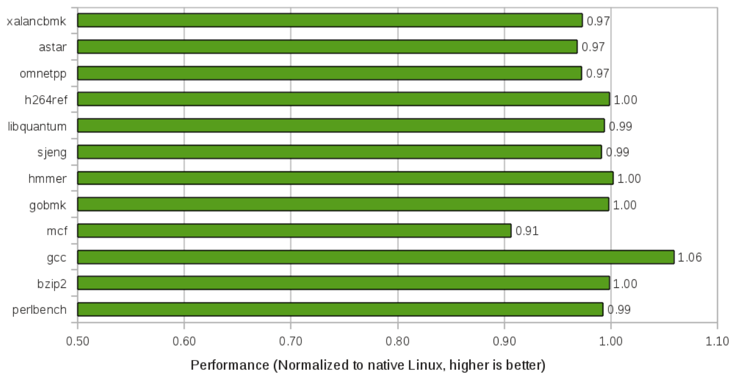
former. With the exception of the `mcf` application (10.36%), all applications have an overhead of less than 3.28%. We contribute the performance increase of gcc to cache effects.

Figure 2.8b displays the results of 9 important operations of the lmbench suite: null (null system call), fork, exec, ctxsw (context switch among 16 processes, each 64 KiB in size), mmap, page fault, bcopy (block memory copy), mmap read (read from a file mapped into a process), and socket (local communication by socket). All tests show a performance overhead of less than 10% and on average even as low as 3.22%. As our implementation does not require any additional computation when no SPMs are executed, this performance overhead can be contributed completely to the hardware virtualization support. We expect that as this support matures, overhead will be reduced further. Also note that Fides can be unloaded when it is no longer required, reducing the overhead to 0%.

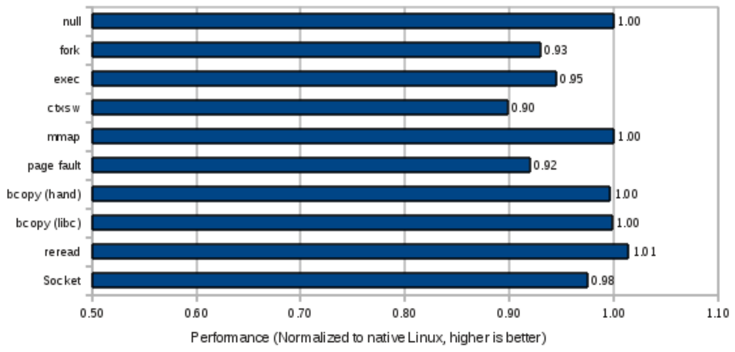
The hypervisor used by the Fides architecture minimizes the number of VM exits. Using hardware support of the processor, modifications of any control register, including `CR3` that references the active paging tables, do not pass control to the hypervisor. The extended page tables (EPT) prevent guest access to physical pages containing security sensitive data or code. In practice a VM exit is only generated when the `cpuid` or `vmcall` instruction is issued. The former requests information about the processor on the platform while the latter is used to implement an interface to the hypervisor. Both are rarely used.

Local communication between SPMs

To measure the impact of communication, two microbenchmarks were used. The first one measures the cost of a call to an SPM compared to a call to a similar driver in the legacy operating system. A simple SPM of two 4 KiB pages was used for the test. When its entry point is executed, it immediately passes control back to the calling application. The driver used for comparison is similar. When it is accessed using the ioctl interface, it immediately returns. Table 2.3 displays two results. The Entry row shows the measurement of the time between the point of call in the user application and the point of delivery in the SPM or driver. The Round trip row measures the time between call and return. Each test was executed 100,000 times. Results show a performance overhead of 8,167% and 8,781% respectively. This significant overhead is caused by the fact that for each SPM invocation two VM entries and exits are required to pass execution control from the legacy VM to the secure VM and back, as well as four context switches from supervisor to user mode are required (two in each VM, see Figure 2.7). However, given the substantial security guarantees, these costs seem very acceptable.



(a) SPECint 2006



(b) lmbench

Figure 2.8: The performance impact of Fides on the overall system.

Table 2.3: SPM vs. driver access overhead (in μs).

	SPM	Driver	Overhead
Entry	4.35	0.05	8,167.14%
Round trip	6.58	0.07	8,781.73%

Table 2.4: Microbenchmarks measuring the cost (in μs) of calling an (authenticated) SPM.

	one-way auth.		two-way auth.	
	tstSPM	sha512	tstSPM	sha512
timing	7.82	95.72	8.28	110.63
overhead	6.03%	1,198%	12.22%	1,400%

The second microbenchmark measures the cost of different authentication techniques, over an average of 100,000 runs. Two SPMs were created, called Ping and Pong. Ping invokes a service in Pong that simply returns a static response. Four different setups were used, shown in table 2.4. Performance results without any authentication measured 7.37μs and is used as the baseline. Columns one and two display results for one-way authentication. Column one measures repeated authentication where Fides is requested to check the `spm_id` of the called module and column two measures an initial authentication using SHA-512 and the `lytSPM` system call, which is less flexible than authentication using a security report, but used, for example, by the Vault. Initial authentication has a performance penalty of 1,197.97%. Repeated authentication is much less expensive at 6.03%. Similar tests were conducted for two-way authentication. Performance penalties increased to 1,400.06% and 12.22% respectively.

TLS-Enabled Web Server

As a macrobenchmark, we protected an TLS-enabled web server. Our goal was not only to protect the web server's long term secret, but the entire TLS-connection, including session information. This prevents a kernel-level attacker from hijacking the connection and renders him only as powerful as an attacker with complete control over the network.

We used the PolarSSL cryptographic library[6] and some functions of the diet libc library that are security sensitive in our use case (i.e. `sscanf`) to implement the SPM. We also implemented our own simple `malloc` memory management. The NSPR[7] library was used to create a multithreaded server. Each connection is handled in a separate thread with its own SPM.

We used the Apache Benchmark to benchmark a web server returning a static 74-byte page to the client over an TLS-connection protected by a 1024-bit RSA encryption key. Table 2.5 displays the server's performance with a varying number of concurrent tranctions, each setup receiving 10,000 requests. Repeated

[6]http://polarssl.org/

[7]https://www.mozilla.org/projects/nspr/

Table 2.5: HTTPS-server performance (in #req/s).

Concurrency	Unprotected	Protected
1	50.27	50.09
5	83.72	83.26
10	97.34	83.44
50	102.73	89.27
100	103.10	88.74

context switches during the TLS negotiation phase lead to a performance cost of up to 13.93%.

2.6 Related Work

There exists a vast body of research on software security. For system-level software, memory safety related vulnerabilities are an important threat. We refer the reader to Younan et al. [146] for a comprehensive survey and to Erlingsson et al. [43] for a gentle introduction. Practical countermeasures however, cannot defend against all possible attacks and countermeasures with strong guarantees typically come with a significant cost.

An alternative approach is to turn to formal verification of systems and applications to provide very strong assurance of security properties. Impressive achievements include the verification of the HyperV hypervisor [29], and the complete seL4 microkernel [64]. While seL4 is also able to provide strong security guarantees, a key design objective for Fides is compatibility with legacy operating systems: what should *minimally* change to a commodity OS to support protection of critical software components against kernel-level malware. For that design objective Fides outperforms seL4 easily.

Other research results proposes hardware modifications to increase security guarantees [70, 139]. However, one of our objectives is to remain compatible with existing systems.

Yet another line of research takes advantage of virtualization techniques to increase the protection of sensitive data by increasing protection of the kernel [109] or applications [26, 41, 125] in the presence of malware. While these research results present interesting solutions, we are not convinced that they can ever be made provable secure due to a possibly very large TCB. The line of research most related to the work in this chapter sets out to bootstrap trust in commodity computers by securely capturing a computer's state. An excellent survey of this research field is given by Parno et al. [94]. We only discuss the most relevant work.

Existing research can be categorized based on the root of trust. Some works assume a trusted boot sequence to start a hypervisor before the commodity operating system is loaded. Terra [47] takes this approach to isolate closed boxes of software. Possible attack vectors are minimized by preventing additional code to be loaded in the box. Nizza [112] takes a more integrated approach, executing small pieces of code in isolation on the Nizza microkernel. While this architecture is similar to Fides, its TCB of 100,000 lines of code is an order of magnitude larger.

The root of trust can also be started dynamically, after the system has booted. Pioneer [110] and Conqueror [74] take this approach completely in software. However, many assumptions are hard to guarantee in practice and confidentiality of data cannot be provided.

Stronger guarantees can be provided when the TPM chip is used [34, 62]. Seminal work in this field has been conducted by McCune et al. Their Flicker architecture [78] can execute pieces of code, called PALs, in complete isolation while secrecy of sensitive information is guaranteed. The TPM chip is used intensively by Flicker, leading to a significant performance cost. The TrustVisor architecture [77] mitigates many of these disadvantages by using a hypervisor and a software delegate of the TPM chip. P-MAPS [105] operates similar to TrustVisor but does allow protected code to access unprotected pages. More recently, Azab et al. showed [11] that the System Management Mode (SMM) can be used to implement a hypervisor-like security measure, ensuring integrity and security of module code and data. While these systems also offer strong isolation of modules, their focus is on remote attestation. They are ill-equipped for practical implementation of applications with a large number of (interconnected) modules: (1) writing co-operating protected modules is hard since modules do not share the same address space. (2) Messages sent between modules may never be delivered. (3) A lack of support for multiple instances of the same module makes it extremely challenging to build modular systems. Our approach mitigates these disadvantages by combining a run-time system and a compiler to allow programmers to easily develop protected modules that are able to seamlessly interact with unprotected code and other modules.

Finally, our own previous work on trusted subsystems in embedded systems [120] proposed a program-counter dependent memory access control model. An implementation technique was sketched for embedded systems with a flat address space and with special-purpose hardware support. El Defrawy et al. [40] implemented attestation of code in embedded devices based on this access control model, but limited themselves to only a single module. This significantly reduces the complexity of a hardware implementation as no primitives for module creation, destruction or authentication is needed. A full hardware implementation for high-end CPUs may not be feasible as the access control mechanism would require interaction with existing memory translation mechanisms. A key contribution of Fides is that it shows that similar ideas can be implemented on commodity hardware while remaining fully compatible with legacy software.

2.7 Conclusions

Commodity operating systems have been proven hard to protect against kernel-level malware. This chapter presented a combined run-time system and compiler approach to selectively harden modules. Using a program-counter based access control model, programmers are able to develop modules in standard C-code that co-operate seamlessly with unprotected code and other modules. It has been proven that such modules are fully protected while system-wide performance impact is limited.

2.8 Post-Publication Remarks

Hardware virtualization support enables hypervisors to almost [63, 103] completely capture and manipulate a virtual machine's state. This enables hypervisor-based protected-module architectures such as Fides to provide versatile security primitives to unmodified guest operating systems and applications. As security measures can be implemented with a limited amount of code and can easily be extended to provide additional security primitives, such architectures provide interesting research platforms. Vasudevan et al., for example, took this approach and presented XMHF [134], a formally verified, modular hypervisor framework that was later extended to support secure I/O [151].

Unfortunately, VM-boundary crosses incur a significant performance overhead when compared to system calls. In order to avoid these overheads, most research systems proposed after publication of Fides, have looked at hardware modifications. Subsequent work by Noorman et al., for example, implemented Sancus, a protected-module architecture based on the same program-counter-based access control mechanism on a low-end embedded device [87]. Sancus shows that SPM-boundary crosses can be implemented with almost no performance overhead and with limited hardware costs. De Clercq et al. later presented a secure interrupt mechanism compatible with Sancus [35].

Koeberl et al. presented, TrustLite [65], an alternative approach to implement a protected-module-architecture on embedded devices. By adding an "execution-aware memory protection unit" (EA-MPU) to the platform that enforces a memory access policy per memory region, platform constraints on developers are reduced. For instance, protected modules, called trustlets, can span over multiple memory regions and inter-trustlet communication overhead can be reduced by placing messages in a shared memory region. In contrast to Sancus, TrustLite does not support the destruction of trustlets without a platform reset. This significantly simplifies inter-trustlet communication as callers are no longer required to ensure that the called module is still present in memory.

Owusu et al. also proposed a hardware-based protected-module architecture but targeted the x86 platform. Their OASIS [89] architecture enables the creation of highly-isolated protected modules. These modules are not only protected from malware running in unprotected memory, but also from hardware attacks by keeping them in CPU cache. Data can be bound to protected modules based on a cryptographic key that is only available to the module; similar to Sancus, the cryptographic key is derived from the initial state of the module and a platform key. Remote attestation primitives enable remote code attestation. Inter-module communication is not considered, but can be implemented similar to remote attestation techniques. OASIS shares significant similarities with Intel SGX [7, 51, 57, 81], which will be discussed in more detail in Appendix A.

All these approaches only provide a limited amount of protected modules or only provide coarse, page-based protection. This may be sufficient when application components such as libraries need to be fortified. Source-code languages such as Java or C# however, consider much smaller objects as primary isolation components. When compiling such languages to coarse-grained protected-module architectures by placing multiple source-level objects in a single protected module, references to objects need to be carefully handled when passing protection boundaries [98]. Support for much finer-grained memory protection may enable a much easier fully-abstract compilation scheme.

Mondriaan Memory Protection [142, 143] supports such a fine-grained protection granularity and was designed to meet a large range of applications. For instance, the authors point out that fine-grained protection can facilitate buffer overflow countermeasures, optimize garbage collection and can avoid copying data across protection domains leading to speed-ups of many applications (e.g., the network stack). The versatile nature of Mondriaan's protection scheme enables it to support a large range of protection mechanisms. As such, program-pointer-based access control could also be supported easily using a similar approach taken by Fides. This comes at the unfortunate cost of requiring multiple hardware caches to minimize memory access pressure. Impact on die size remains unclear, but may be significant.

Capability systems provide yet another interesting alternative. Having deep roots in academic research, including hardware support for such systems [102], they are recently being reconsidered. Woodruff et al. propose CHERI, a hardware-based capability system [144]. CHERI supports the creation of segments at byte-granularity and can enforce that segments are only accessible using a capability to do so. Capabilities are designed as pointers that are specially tagged in memory. A light-weight typed assembly language ensures that capabilities cannot be fabricated. One of CHERI's disadvantages is that capabilities are only dependent on the location of the segment they reference. When segments are destroyed, the entire memory contents needs to be scanned and stale capabilities purged. Failure to do so may allow capabilities to linger and enable an attacker to access a new segment without ever receiving a capability to do so.

Vilanova et al. [136] also proposed a capability system called CODOMS, but in contrast to CHERI, the "domains" they reference can consist of multiple memory ranges each with a specific set of access rights. To simplify capability revocation, capabilities may be tagged with a counter and the address of a reference counter. Capabilities are only valid when both counter values match. Similar to Mondriaan Memory Protection, a large variety of memory protection schemes can be supported, including Fides-like program-counter-based memory access control, but at a non-negligible hardware cost [136].

Chapter 3

Salus: Non-Hierarchical Memory Access Rights to Enforce the Principle of Least Privilege

Publications & Acknowledgments

Avonds, N., Strackx, R., Agten, P., and Piessens, F. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *Security and Privacy in Communication Networks (SecureComm'13)* (Sept. 2013), vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer International Publishing, pp. 252–269.

Strackx, R., Agten, P., Avonds, N., and Piessens, F. Salus: Kernel support for secure process compartments. *Accepted for publication in Endorsed Transactions on Security and Safety*, EAI.

This chapter is partly based on the master thesis of Niels Avonds. His efforts to implement a protected-module architecture under the supervision of Pieter Agten, Raoul Strackx, and Frank Piessens, lead to a publication in the proceedings of the SecureComm'13 conference. Later this work was extended by Raoul Strackx to support the creation of unforgeable references to compartments.

This work is accepted for publication in the journal for Endorsed Transactions on Security and Safety and is fully incorporated in this chapter.

Abstract

Consumer devices are increasingly being used to perform security and privacy critical tasks. The software used to perform these tasks is often vulnerable to attacks, due to bugs in the application itself or in included software libraries. Recent work proposes the isolation of security-sensitive parts of applications into protected modules, each of which can be accessed only through a predefined public interface. But most parts of an application can be considered security-sensitive at some level, and an attacker who is able to gain in-application level access may be able to abuse services from protected modules.

We propose Salus, a Linux kernel modification that provides a novel approach for partitioning processes into isolated compartments sharing the *same* address space. Salus significantly reduces the impact of insecure interfaces and vulnerable compartments by enabling compartments (1) to restrict the system calls they are allowed to perform, (2) to authenticate their callers and callees and (3) to enforce that they can only be accessed via unforgeable references. We describe the design of Salus, report on a prototype implementation and evaluate it in terms of security and performance. We show that Salus provides a significant security improvement with a low performance overhead, without relying on any non-standard hardware support.

3.1 Introduction

Both desktop and mobile devices are increasingly being used to perform security and privacy critical tasks, such as online banking, online tax declarations and purchasing goods from online stores. The software to perform these tasks either runs inside a web browser, or is written as a standalone application. In both cases, the software is often vulnerable to attack, either due to bugs in the application itself or due to bugs in included software libraries or in the runtime environment used to execute the application (e.g. the browser).

Because of their widespread use and potentially high-impact nature, such applications form an interesting target for cybercriminals. Past research has focused on defending against specific attack vectors such as buffer overflows [6, 42, 122, 146], format string vulnerabilities [31] and non-control-data attacks [25].

Even though many of these defense mechanisms are applied in practice, successful attacks against high-value applications are still common.

To provide stronger security guarantees, recent research efforts have shifted from trying to defend entire applications against every possible attack to providing strong isolation of sensitive parts of an application with a minimal trusted computing base (TCB). Cryptographic keys of an application, for example, can be isolated in a protected module that has complete control over its own secrets; the module can only be accessed via its public interface. Accessing the cryptographic keys directly at assembly level is prevented by the security architecture. Thus, an attacker that has successfully exploited a vulnerability in the non-security sensitive part of the application still cannot access the cryptographic keys.

A large number of security architectures providing such protection mechanism have been proposed in this field, including software implementations using hardware virtual machine support [77, 119], trusted computing primitives [78], implementations based on system management mode [11] and even completely hardware-based solutions [87, 89, 120]. Recent research papers by Intel indicate that hardware support for these security architectures will also become available on mainstream x86 platforms in the near future [7, 51, 81].

In practice, isolating security-sensitive parts of an application is difficult since most program logic can be considered security-sensitive at some level [112]. A too coarse-grained approach will result in bloated modules that may contain vulnerabilities and that are too big to be formally verified [47]. Minimum-sized modules on the other hand, can provide strong and easily verifiable guarantees, but may need to expose insecure interfaces to interact with other modules. This is a common problem of module-isolating security platforms, both in software as in hardware. Application developers are trapped in a catch-22 with possibly severe security consequences. In the recent DigiNotar attack [53], for example, the root CA's private cryptographic key was securely stored in a hardware security module (HSM), but its insecure interface enabled attackers to sign arbitrary certificates.

In order to improve upon these shortcomings, we acknowledge that almost every part of an application performs security-sensitive operations. To reduce chances of a successful attack, we propose to partition the *entire* application into compartments and implement a non-hierarchical access control mechanism between compartments. Compartments not only provide provable secure isolation of stored private data (as modules in related work do), but are also able to confine software vulnerabilities to the compartments they occur in by (1) restricting the types of system calls that they are allowed to issue, (2) enabling authentication of calling and called compartments and (3) enabling

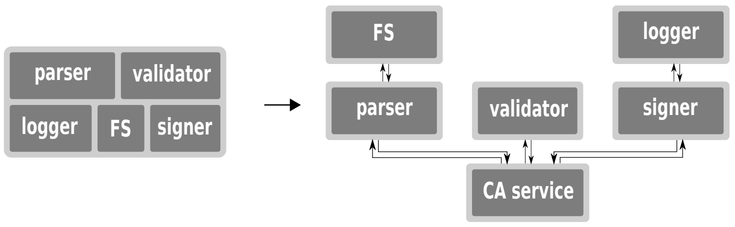


Figure 3.1: Salus' compartmentalization enables strong isolation of security-sensitive data *and* contains possibly vulnerable code. Multiple vulnerable compartments need to be exploited to attack the system successfully.

compartments to only service requests made through unforgeable references, reducing the impact of insecure interfaces. By separating likely attack vectors from attack targets and placing them into different compartments, an attacker would need to exploit vulnerabilities in multiple compartments to reach her goal.

Each compartment resides in its own chunk of memory, consisting of a *public section* containing the code of the compartment and a *private section* storing sensitive data (e.g. cryptographic keys or passwords). Only when executing the public section of a compartment can the private section of that compartment be accessed. To force other compartments to use a compartment's public interface, execution can only enter the public section via well-defined code entry points and, if required by the compartment, unforgeable references. As an additional protection measure and to support the principle of least privilege [106], compartments have the ability to restrict the types of system calls they are allowed to perform. Once a compartement drops a system call privilege, it cannot be re-acquired. This further reduces the impact of compromised compartments. The compartments of a single process all run in the same address space, providing a lightweight programming model that enables legacy applications to be ported easily and incrementally.

Consider, as an example, an X.509 certificate signing application consisting of a parser, a validator, a signer and a logging component (Figure 3.1). When run as a single monolithic application, a vulnerability in any one of these components can lead to the compromise of the entire application. When placing each of these components in a separate compartment under Salus, components can only call each other through their well-defined interfaces using unforgeable references and each component can authenticate both its callers and its callees. This restricts the flow of data and control between compartments to predefined patterns and raises the bar for a successful attack significantly. Consider as an example an attacker who exploited a vulnerability in the parser. In order for her to sign arbitrary certificates, she would either need to provide specially crafted credentials for the submitted certificate that would not cause the “Validator” to raise flags, or she would need to gain direct access to the “Signer” compartment by exploiting another vulnerability in the “CA Service” compartment to leak the unforgeable reference.

Furthermore, by combining unforgeable references and restricting the system calls that can be issued by a compartment, we can provide fine-grained access control to the kernel. Consider as an example the parser and assume that it reads its signing requests directly from the file system. At development time, there are two options. Option 1 is to grant the compartment access to the `open`/`close` and `read`/`write` system calls. In that case an attacker who exploited a vulnerability in the parser can inspect the entire file system with the application's privileges. The second option provides stronger security guarantees by revoking the parser compartment all system call privileges and only providing it with an unforgeable reference to a file system compartment (FS in Figure 3.1). This newly added compartment tightly restrict access to a single folder or file type and only provides the parser access to the files it approves. Having almost unrestricted access to the file system itself, a vulnerability in the FS compartment would enable an attacker to launch similar attacks as in option 1. However, given that this compartment is likely to be several orders of magnitude smaller than the parser compartment, the probability that such an exploitable vulnerability can be found is limited. Such constructs are a well-known advantage of capability systems [19, 36, 144].

Concretely, we make the following contributions in this chapter:

- We present a novel approach for partitioning processes into compartments with support for strong isolation of sensitive data *and* containment of vulnerabilities. To the best of our knowledge, Salus is the first solution that simultaneously (1) reduces the impact of insecure compartment interfaces, (2) enables compartments to restrict the types of system calls they are allowed to perform and (3) executes compartments in the same address space allowing legacy applications to be ported easily without having to marshall in- and output messages.
- We report on a prototype implementation of Salus in the Linux kernel.
- We evaluate the security of our approach and the performance of our prototype.

The remainder of this chapter is structured as follows: in Section 3.2 we define our attacker model and describe our desired security properties. In Section 3.3

we provide a high-level overview of Salus, before presenting our prototype implementation in Section 3.4. Finally we evaluate our approach in Section 3.5, discuss related work in Section 3.6 and conclude in Section 3.7.

3.2 Attacker Model & Security Properties

We consider an attacker able to inject and execute malicious shellcode in vulnerable compartments, for example, by exploiting a buffer overflow vulnerability. Our system must defend against such attacks in the following way:

- The exploitation of a compartment must not affect the security of compartments other than those that explicitly trust the compromised compartment.
- Once a compartment is exploited, an attacker is only able to call other compartments via their proper interfaces *iff* it received a reference to those compartments. Simply guessing the compartment's virtual address is not sufficient.
- An exploited compartment may still interact with other compartments and pass compartment references. Called compartments however, will check the types of received arguments and will refuse to call other compartments with an incorrect type.
- Attackers are explicitly allowed to create new compartments. There is thus no guarantee that compartments requesting protection can be trusted. Hence, Salus must isolate compartments from one stakeholder from those of another, possibly malicious, stakeholder.
- An attacker should not be able to execute system calls that have been revoked.

Kernel-level and physical attacks are considered out of scope. Regarding the cryptographic primitives used, we assume the standard Dolev-Yao model [37]: An attacker can observe, intercept and adapt any message. Moreover, an attacker can create messages, for example by duplicating observed data. However, the cryptographic primitives used cannot be broken.

Table 3.1: The memory access control model enforces, for example, that a compartment's private section (4^{th} column) can only be read-write accessed from the public section of the same compartment (3^{rd} row).

from\to	Entry pnt.	Public sec.	Private sec.	Unprot. mem.
Entry pnt.	`---`	`--x`	`---`	`---`
Public sect.	`r-x`	`r-x`	`rw-`	`rwx`
Private sect.	`---`	`---`	`---`	`---`
Unprot. mem/ other comp.	`r-x`	`r--`	`---`	`rwx`

3.3 Overview of the Approach

This section presents a high-level overview of Salus. Section 3.3.1 describes the memory access control mechanisms on which Salus is based. Section 3.3.2 presents the services Salus provides to protected applications and section 3.3.3 shows how these services are used in a typical life cycle of a compartmentalized application. Authenticated communication between compartments and unforgeable references to compartments are discussed in sections 3.3.4 and 3.3.5 respectively. Finally we discuss how new and legacy applications can be compartimentalized in section 3.3.6.

3.3.1 Compartments of Least Privilege

Structure of a Compartment

The basic layout of a compartment, shown in Figure 3.2, is a virtual memory region divided into two sections: a public section and a private section. The *public* section contains the compartment's code and any data that should be read accessible by other compartments of the same application. This section can never be modified after initialization, which enables other compartments to authenticate the compartment based on a cryptographic hash of the public section (see Section 3.3.4). The start of the functions that make up the compartment's public interface are marked as entry points. Execution of the compartment can only be entered through these memory locations (see Table 3.1).

The *private* section contains the compartment's private data, which consists of application-specific security-sensitive data (e.g. cryptographic keys) as well as data relevant to the correct execution of the compartment, such as the runtime

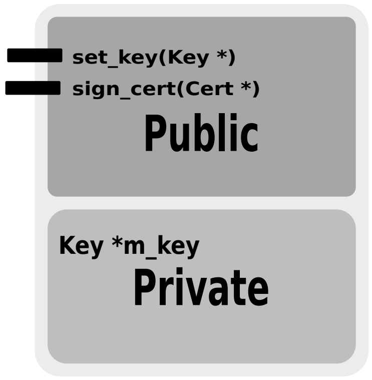


Figure 3.2: Salus' memory access control model enables the creation of compartments that provide strong isolation guarantees to sensitive data. Secure communication primitives reduce the impact of an insecure interface.

call stack. The data in the private section is read and write accessible[1] from within the compartment, but completely inaccessible for code executing outside of the compartment. Note that since each compartment has its own private call stack, intercompartmental function call arguments and return addresses must be passed via CPU registers (as opposed to passing them using the runtime stack).

Applications can still have a memory region that is not part of any compartment. This region is termed *unprotected memory* and is read/write accessible from any compartment. All compartments of the same application run in the same address space, which facilitates the compartmentalization of legacy applications. Nonetheless, fine-grained compartmentalization of a large code base can still require significant developer effort. Therefore, Salus enables applications to be compartmentalized incrementally by storing code and/or data in unprotected memory. While unprotected memory does not provide any of the security guarantees of compartments, it does provide an incremental upgrade path for legacy applications.

As an example of a compartment, consider a single compartment providing a certificate signing service. The compartment provides two functions as part of

[1]By preventing code execution in the private section, the chances that an attacker is able to successfully exploit a vulnerability in a compartment, is reduced significantly. We acknowledge that this restriction may hinder applications that rely on generated code (e.g., JITed applications). Support for such applications could be easily added; at creation-time the creator should specify whether the new compartment's private section should be executable. As we believe this is a special case, we will not consider it for the remainder of the chapter.

its public interface (see Figure 3.2). The first function, `set_key`, allows setting the cryptographic key used to sign certificates. This key is stored as the `m_key` variable in the private section. The second function, `sign_cert`, handles the actual signing requests. Salus' memory access control model ensures that only these two functions are executable; any attempt to jump to another memory location in the compartment will fail. Similarly, any attempt to directly read or write the cryptographic key in the private section from unprotected code or from another compartment will be prevented. Only after calling a valid entry point will read and write access to the private section be enabled, making the cryptographic key only accessible while the compartment is being executed. When the function is terminated, execution returns to the caller and read/write access to the compartment's private section will again be disabled.

Special care is required when execution returns to a compartment after a call to another compartment. Execution must resume at the return location, which is the instruction right after the call instruction in the caller compartment. This location however does not typically correspond to an entry point and hence would cause a memory access violation according to Salus' memory access control model (see Table 3.1). Compartments can implement a *return entry point* to avoid this access violation. Right before calling another compartment, the return location is placed on the top of the calling compartment's private stack while the location of the return entry point is passed to the callee in a register. When the intercompartmental call has finished, execution flow jumps to the return entry point where the return location is retrieved from the compartment's stack and jumped to. Note that a return entry point is a software implementation and follows the same access rights as any other entry point.

Restriction of Privileges

Salus provides two important primitives to limit the impact of a compromised compartment. The first primitive is caller and callee authentication. By authenticating callers and callees, a compartment can limit its interaction to trusted compartments only. Although this does not protect against trusted compartments that have been compromised, it does significantly limit the capabilities of an attacker after a successful exploit. For instance, the "signer" compartment of the CA signing service displayed in Figure 3.1, may only accept calls from the "CA service" compartment. As such, an attacker who successfully exploited a vulnerability in the parser may attempt to call the signing compartment, but the latter will refuse to service the attacker's service request.

The second primitive allows compartments to disable specific system calls for any code executed from within their public section. Once a system call is disabled, it cannot be re-enabled. By carefully partitioning an application into compartments, each of which should disable any system call it doesn't need, the impact of the exploitation of a vulnerable compartment is minimized. Note that much more fine-grained solutions exist than restricting complete system calls [138]. However, we focus on providing strong compartmentalization primitives that can be used as a building blocks for finer-grained privilege restriction mechanisms.

3.3.2 Provided Services

To enable compartmentalization of applications, Salus provides runtime support of the following services:

Create After code is loaded into memory, this service can be used to create a new compartment. Given a memory location and size for the compartment to create, Salus will enable memory protection for this region and will return a system-wide unique ID for the new compartment. Note that our attacker model explicitly allows the creation of new compartments by an attacker.

Destroy A compartment can only be destroyed by the compartment itself. After destruction, the memory access protection is disabled. Hence, a compartment should overwrite any private data before destruction.

Request compartment ID and layout To support secure communication, Salus provides a service to request the ID and layout (i.e. the size and locations of the public and private sections and the available entry points) of a compartment covering a given memory location. If there is no compartment at the specified location, the service returns an error code. This service is used as a primitive in compartment authentication.

Request caller ID To support caller authentication, Salus provides a service to request the ID and layout of the compartment that called an entry point of the current compartment.

Disable system call To limit the impact of the exploitation of a compartment, unused system calls can be disabled. To prevent an attacker from gaining system call privileges by creating a new compartment, compartments inherit system call privileges from their parent.

3.3.3 Life Cycle of a Compartmentalized Application

Compartmentalized applications can be started as any other application. After the (trusted) operating system or loader loads the application into memory and starts its execution, the application can create the required compartments. Finally, execution can jump to the compartment containing the application's main function. Compartments can be created at any point during the application's execution, for example, at the time a new (compartmentalized) plugin is loaded.

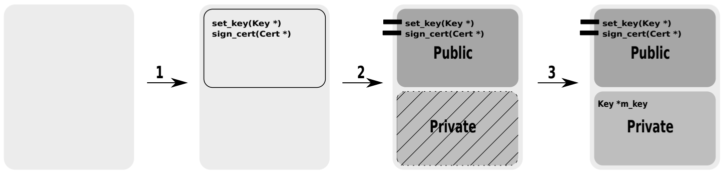
Creation of Compartments

Figure 3.3a shows the process of setting up a compartment. As the first step of setting up a new compartment, the application allocates (unprotected) memory and loads the compartment's code. Next, the application enables protection of this memory region, by calling Salus' creation service. Note that there is no guarantee that the new compartment's code has been loaded correctly into memory, since the creator might have been compromised already. However, any tampering with the code will be detected when the compartment tries to communicate with another compartment, as will be explained in Section 3.3.4.

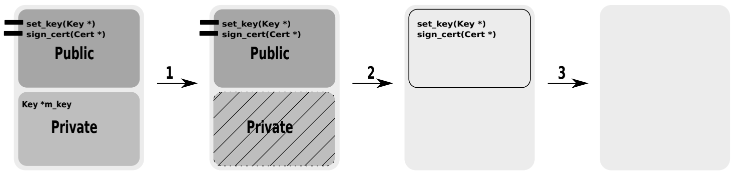
When a new compartment is created, Salus clears the first byte of the private section. This serves as a flag to indicate to the compartment that it should initialize itself when its service is first requested. As part of its initialization, a compartment should clear the private memory locations it will use. This prevents an attacker from crafting a private section by setting it up in unprotected memory locations where a new compartment will later be created. Initialization code should typically also disable the system calls that will not be used during further execution of the compartment.

Destruction of Compartments

The destruction of a compartment, shown in Figure 3.3b, can only be initiated by the compartment itself. This ensures that compartments can clear their private section (which may contain sensitive data), before the memory protection is lifted. In addition, trusted communication endpoints could be notified of the compartment's imminent destruction. After destruction, the unprotected memory area of the destructed compartment can be freed.



(a) Creation



(b) Destruction

Figure 3.3: The life cycle of a protected compartment from creation (3.3a) to destruction (3.3b).

3.3.4 Secure Communication

Salus' memory isolation mechanism provides strong guarantees that sensitive data in the private section can only be accessed by code in the public section [4,96–98]. Reconsidering our certificate signing service as an example (see Figure 3.1), we can prove that the signing key will never leave its compartment. But an attacker with access to the compartment's interface is still able to sign arbitrary certificates. Salus can limit the feasibility of such attacks in two ways: (1) by enforcing both caller and callee authentication, and (2) by requiring that callers have an unforgeable reference to the compartment at hand, which means that guessing the location of a compartment is insufficient to access it. In this section we will focus on authentication of compartments. While we will only discuss authentication of calling and called compartments, a similar approach can be applied when locations of other compartments are passed as arguments. In Section 3.3.5 we will show how compartments can enforce that they can only be called through unforgeable references.

Security Report

Authenticating a compartment consists of verifying whether that compartment adheres to a trusted *security report* of that compartment. A security report of a compartment consists of:

The cryptographic hash of its public section This allows any code to verify that the public section of the compartment has not been tampered with: the cryptographic hash should be recalculated at runtime and be compared to the known-good value stored in the security report. This protects against an attacker who is able to modify the public section of a compartment during its creation, before memory protection is enabled (see Section 3.3.3).

The layout of the compartment When a creation request originates from unprotected memory, the request itself may have been tampered with. An attacker could, for instance, specify an incorrect private section size for the compartment to create. This may result in the use of unprotected memory that should be under Salus' protection. By storing the known-good layout of the compartment in the security report, any code can verify that the layout was not tampered with during creation of the compartment.

A cryptographic signature In order to have integrity protection and authentication of the security report, it is digitally signed by its issuer. Each compartment can decide independently whether or not to trust a certain issuer, which opens up the opportunity to integrate compartments from different parties into a single application. Since the cryptographic signature provides integrity protection, security reports can simply be stored in unprotected memory.

Authentication of Called Compartments

When exchanging sensitive information between compartments, caller and callee must authenticate each other *before* sensitive data is exchanged.

To authenticate a compartment to be called, its ID must first be obtained using Salus' 'request compartment ID' service. Next, the callee's security report must be acquired. For this a central service where each compartment registers to on initialization, can be used. Given the callee's ID, the service should return the (location of the) corresponding security report. Note that this service need not be trusted, as any tampering with the information returned will be detected during the next steps. Once the security report has been obtained, it should be validated by checking the cryptographic signature and by checking that the issuer is trusted. Each compartment should contain a list of trusted security report issuers. Next, the callee compartment's layout should be requested from Salus and a hash of the public section should be calculated. The layout and the hash must be compared to the values listed in the security report. This completes the authentication and allows the caller to securely call one of the callee's public functions.

When calling a compartment that has already been authenticated in the past, a re-validation must occur because the callee may have been destroyed since the last interaction. A full authentication using the security report on every call would be very time consuming, so to reduce the performance impact, Salus allows compartments to be re-authenticated quickly based on their ID. Salus ensures each compartment has an ID that is unique on the system until the next reboot. Hence, a re-authentication can simply consist of requesting the ID of the compartment to be called (using the 'request compartment ID' service) and checking that it is the same as during the initial authentication. Using unique identifiers has the added benefit that code can distinguish between different instances of the same compartment.

Authentication of Calling Compartments

To enable compartments to limit use of their (possibly insecure) interface to trusted caller compartments, Salus provides primitives for caller authentication. For a compartment to authenticate its caller, it can first request the caller's ID and memory location (using the 'request caller ID' service) and proceed to authenticate the caller using the same steps as described above.

3.3.5 Unforgeable references

Salus' access control mechanism and supporting services enable authentication of both callers and callees. Unfortunately, in some situations this does not suffice. Let's reconsider the CA service from Section 3.1 as an example but now assume that it receives signing requests over a network. Figure 3.4 displays how the application can be partitioned into different compartments. A compartment `Listener` listens for incoming network connections and spawns a new `CAConnection` compartment for every connecting client. This compartment is in charge for all future communication with the client. This is similar to a `Socket` object in an object-oriented language. When a connection is established, clients must provide login credentials and a certificate request. In order to isolate vulnerabilities, `CAConnection` hands off incoming messages to a compartmentalized parser. If messages parsed correctly, the parser returns `Credentials` and `Request` compartments to the `CAConnection` compartment, or an error code if parsing failed. Once all data is collected, the `CAService` is called. Based on the provided `Credentials` and `Request` compartments, it will authenticate the client credentials, verify that the client is allowed to request a certificate for the specified domain and finally instruct the `Signer` (not displayed) to sign the certificate request.

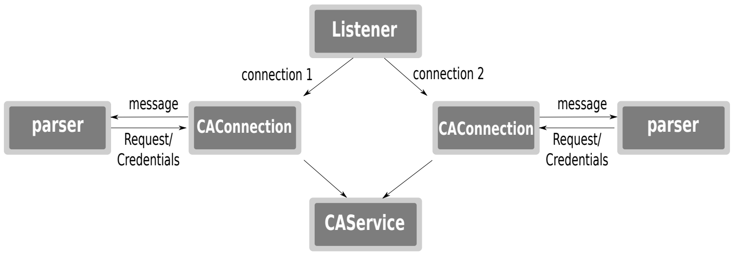


Figure 3.4: By enforcing that compartments can only be accessed via unforgeable references, stronger security guarantees can be guaranteed. Even if an attacker is able to exploit a vulnerability in a parser, she will be unable to access `Request/Credentials` compartments belonging to another connection.

By compartmentalizing the `Parser`, we wish to isolate possible vulnerabilities. Unfortunately, in this setup an attacker able to exploit a vulnerability in the parser may still be able to request certificates for domains that she does not own. The problem arises when the parser returns `Request` and `Credentials` compartments to `CAConnection`. Even though `CAConnection` is able to authenticate the `Parser`, it cannot verify that the received `Request` and `Credentials` compartments are based on the actual data passed to the parser. An attacker who successfully exploited a vulnerability in the parser may be able to scan[2] the entire memory and steal a `Credentials` compartment belonging to a different network connection.

To remedy the problem, we propose using unforgeable references to compartments. *Only* compartments with an unforgeable reference to a compartment have the *capability* to access it. Thus, even if a compartment was compromised, it cannot access or pass references to other compartments that it finds in memory. In our example, a compromised parser may still find a `Credentials` compartment in memory, but it is infeasible that it can guess the correct access token (i.e., it cannot create a correct *unforgeable reference* to it). Even a compromised parser can thus not return "stolen" credentials. This results in a strict separation between different connections.

While unforgeable references in higher programming languages are easily enforceable by a type system, we cannot apply the same approach. An attacker able to exploit a vulnerability in a compartment has assembly-level access and can simply scan the entire memory area to access other compartments. Instead

[2] An in-application level attacker may scan the entire memory in a number of ways. For example, by using Salus' service to request the layout of a compartment for likely compartment locations until a non-error result is returned, or by reading the entire program memory for telltale signs of entry points.

we propose establishing unforgeable references as (location, token) tuples. Newly created compartments must be assigned a cryptographic random number, which can serve as a key to access the compartment's public interface. If and only if a caller provides the correct access token, will a call to the compartment be serviced. This approach has four advantages: (1) with a sufficiently large random number, it is computationally infeasible to forge references, (2) references can be stored in the secret section of compartments, just like any other reference, (3) compartments can implement unforgeable references using the default Salus services, and (4) both standard and unforgeable references can exist in the same application. Section 3.4.4 describes in detail how compartments can implement support for unforgeable references.

3.3.6 Writing Compartmentalized Applications

Writing secure compartments is a non-trivial task; each compartment should keep track of it's own stack, callbacks to unprotected memory should return through a specific return entry point, etc. To ease the creation of such compartments, we developed a C compiler and linker that takes care of such considerations. Application developers can simply annotate functions indicating that they are entry points, reside in unprotected memory or are located in another compartment.

Unfortunately, our tool does not solve all problems at hand. The developer is still in charge to ensure that sensitive data stored in a compartment is never accessed from unprotected memory or by another compartment. The difficulty in ensuring this depends heavily on the programming language used and the quality of the source code. Applications written in C may not be very structured. Each function may allocate memory regions and pass pointers implicit (e.g., stored in allocated memory, or type casted as an integer) or explicit (e.g., as arguments) to other functions. Compartmentalizing such legacy applications may be difficult, but given that all compartments execute in the same address space, an incremental path exists. Developers may place functions that operate on the same sensitive data in the same compartment, while initially still storing the data in unprotected memory. When all functions are placed in the compartment and sensitive data is thus only accessed by a single compartment, it can be allocated securely inside the compartment. Tools such as logging access right violations during development instead of stopping the application (as proposed by [15]) may be helpful in this process but manual inspection of code is still required.

Object-oriented languages on the other hand, may already enforce strict data encapsulation; data may only be accessed through the object's public interface.

In such cases each class may be compiled as a separate compartment but to minimize overhead caused by crossing protection boundaries, multiple classes may be placed together in a single compartment.

3.4 Implementation

Access rights to compartment sections depend on the value of the program counter. For instance, only if execution is in the public section of a compartment, will the private section of that compartment be read/write accessible. This program counter-based memory access scheme is at the core of Salus' protection mechanism. Enforcing this scheme purely in software would have a huge performance impact as every memory access has to be checked. A pure hardware implementation of the scheme is possible [87, 89], but prohibits its use on commodity, off-the-shelf PC platforms. The approach taken for Salus combines the best of both alternatives, by using the key insight that memory access rights for compartments only need to change when execution crosses a compartment border. This allows Salus to use the standard memory management unit (MMU) to enforce the memory protection scheme.

A prototype for Salus has been implemented as a Linux kernel modification. Section 3.4.1 describes how the program counter-based access control mechanism is implemented in this prototype. Section 3.4.2 describes the API Salus provides to processes and Section 3.4.3 lists the Linux system calls that had to be modified in order to provide a secure implementation of the protection mechanism.

3.4.1 Program Counter-Based Access Control

By aligning compartment sections to pages, the standard MMU found on any recent commodity computer can be applied to enforce the required memory protection scheme. After a compartment is created (e.g. from unprotected memory), the MMU access rights for the pages of the new compartment are set up according to Table 3.1: the public section is world-readable while the private section is isolated completely.

When execution tries to enter a compartment (e.g., because of a call instruction), a page fault is generated by the MMU. Based on the memory location addressed and the access type (read, write or execute), Salus determines whether a valid entry point was called and, if necessary, modifies the access rights of the calling and called compartments' public and private sections, according to Table 3.1. Access rights of pages unrelated to the two involved compartments are not modified, which minimizes the number of page faults and access right modifications, thereby reducing the overall performance impact.

Because unprotected memory is always readable, writable and executable, no page fault is generated when execution returns from a compartment to unprotected memory. To restore the access rights of the exited compartment, the compartment itself must issue a system call to Salus.

Since all threads of the same process normally share the same page tables, our approach cannot guarantee the required security properties in case of multiple threads. However, this is not a fundamental limitation of our model. Support for multithreaded applications can be added by modifying the kernel in order to provide each thread with a separate set of page tables. All threads have identical virtual-to-physical mappings, but with different access rights depending on the currently executing compartment in each thread. Compartments also must be multithreading-aware and provide a separate stack per thread. Our prototype currently does not support multithreading.

The Linux page fault handler was modified to implement these access right modifications. To keep track of a process' compartments, the Linux process descriptor data structure was extended with a list of `comp_struct` structures. Each `comp_struct` describes a single compartment and contains:

- The (virtual) start address and length of the public and private sections
- The compartment's unique ID
- The compartment's saved stack pointer
- A list of the compartment's remaining system call privileges

3.4.2 System Call API

The following new system calls were implemented in the Linux kernel. These system calls represent the API Salus provides to processes.

`void salus_create(void* start, uint len_pub, uint len_priv)` Before a new compartment is created, the list of existing compartments is checked to ensure that the new compartment will not overlap with any existing ones. New compartments must also not overlap with the kernel or have their memory pages mapped to files. When these checks succeed, a new compartment is created and added to the current process' compartment list. It receives the same system call privileges as its parent.

`void salus_destroy(void)` Since compartments can only be destroyed from within their own public section, this system call does not require any arguments. This system call restores the original memory access rights on the memory region occupied by the executing compartment and then removes the compartment from the current process' compartment list.

`struct comp_layout* salus_layout(void* addr)` This system call returns the ID and memory layout of the compartment covering a given memory location. It can be implemented by simply iterating over the current process' compartment list until a matching compartment is found. A `null` pointer is returned when there is no compartment covering the given address.

`struct comp_layout* salus_caller(void)` This system call returns the ID and memory layout of the compartment that last called an entry point of the current compartment. A `null` pointer is returned when the current compartment was last called from unprotected memory.

`void salus_syscall_disable(uint syscall_id)` This system call disables further use of the specified system call, by removing it from the list of system call privileges in the `comp_struct` of the current compartment. Once a system call is revoked, it cannot be re-acquired.

`void salus_return(void* addr)` Before execution returns from a called compartment back to its caller (i.e. unprotected memory or another compartment), the access rights of the called compartment's pages need to be restored. This system call performs this access rights modification and then continues execution at the specified address.

3.4.3 Conflicting System Calls

Some existing system calls in the Linux kernel conflict with Salus' compartmentalization. Additional security checks had to be inserted for these conflicting system calls.

mprotect The `mprotect` system call can be used to change the access rights of pages in memory. Additional checks were added to prevent this system call from modifying the access rights of compartments.

mmap Existing system calls such as `mmap` or `mremap` modify the virtual address space of a process. An attacker could abuse these system calls to map a compartment's private section to a file, for instance. When the compartment then writes sensitive information to the newly mapped pages, this information may leak to an attacker. We prevent this attack by verifying that a compartment is mapped correctly before it is called. These checks were also added to the `salus_layout` API call.

personality In Linux, each process has a *personality*, which defines the process' execution domain. The personality includes, among other settings, a `READ_IMPLIES_EXEC` bit, which indicates whether read rights to a memory region should automatically imply executable rights as well. For compartments this would result in world-executable public sections, nullifying the use of designated compartment entry points. Therefore, Salus enforces that this bit is disabled for compartmentalized processes.

fork The `fork`, `vfork` and `clone` system calls can be used to create a new process or thread. As these processes or threads share parts of their page tables, the elevated access rights of the private section of a called compartment, affects all processes/threads and enable its access from unprotected memory. While these system calls could be modified to create copies of the page tables leading to the same virtual-physical address translation but with different access rights, our research prototype currently does not support this. Linux' existing `CLONE_VM` and `VM_DONTCOPY` flags are used to prevent compartments being mapped in the new process or thread. Checks were also added to the `madvice` system call, since it can be used to modify the `VM_DONTCOPY` flag.

3.4.4 Unforgeable references

Implementing support for unforgeable references consists of two steps: (1) newly created compartments must generate a cryptographic random number, and (2) whenever a compartment is called, it must check whether the caller did indeed have the capability to access it.

The first step can be achieved in two ways. One option is to modify Salus' `salus_create` service call (see Section 3.4.2). After creating the compartment, the kernel generates a new cryptographic random number (i.e., the access token) and stores it at a specific location in the compartment's private section. Finally the `salus_create` service call returns the (location, access token) tuple as the unforgeable reference.

Alternatively, newly created compartments can be taken ownership of on a first-call basis, by providing a `take_ownership` entry point that generates and returns an unforgeable reference on its first call. Only the first compartment that requests ownership will be provided with the unforgeable reference, subsequent calls to this entry point will be rejected. While malicous compartments may

```
take_ownership:
    if ( token != 0 )
        return -1;
    else
    {
        token = gen_rand();
        return token;
    }
```

Listing 3.1: An implementation of the `take_ownership` entry point.

"steal" newly created compartments by taking ownership as soon as possible, they do not gain any additional power, since compartments are created from unprotected memory and hence do not possess any sensitive information that may leak to an attacker. Listing 3.1 shows a sample implementation of the `take_ownership` entry point in pseudo code.

In the second step, a called compartment must check whether the caller did indeed have the capability to access the compartment. To perform this check, the caller must pass the access token of the unforgeable reference to the called entry point. If and only if the provided token is identical to the token stored in the compartment's private section, will the call be serviced. Otherwise an error value will be returned. Note that the compartment is able to specify for every entry point whether or not it requires the access token to access it. The `take_ownership` entry point, for example, will never require a capability.

3.5 Evaluation

The effectiveness of Salus' protection mechanisms is evaluated in Section 3.5.1 and its performance impact is discussed in Section 3.5.2.

3.5.1 Security Evaluation

To evaluate Salus' security, we make a distinction between memory-safe and memory-unsafe compartments. A memory-unsafe compartment can be exploited by an attacker using low-level attack vectors such as buffer overflows [6, 42, 122, 146], format string vulnerabilities [31] or non-control data attacks [25]. A memory-safe compartment does not contain such vulnerabilities, for instance because it was written in a memory-safe language or simply because the compartment doesn't contain any memory-safety bugs.

Since memory-safe compartments cannot be exploited directly, the only attack vector against them is through exploitation of another compartment in the same address space. However, recent research [4,96–98] has shown that memory protection mechanisms such as those offered by Salus, are able to provide full source code abstraction. This means that, even when other compartments have been successfully exploited, an attackers' capabilities are limited to interacting with the memory-safe compartment through its public interface. A carefully constructed interface can thus effectively limit the attack surface of a compartment. But in many cases, creating a secure interface is still a challenging problem [73]. Recall the example of a certificate signing compartment introduced in Section 3.3.1: even if the private cryptographic key is never exposed, an attacker could potentially still use the compartment's interface to sign arbitrary certificates [53]. By taking advantage of Salus' support for caller/callee authentication however, the risk of such an attack can be minimized by only servicing requests from compartments that would issue them as part of the normal operation of the application (e.g. in Figure 3.1, the signer compartment should only accept requests from the validator compartment).

Memory-unsafe compartments may still contain vulnerabilities that can be exploited by attackers. Even though Salus does not prevent such attacks, compartmentalization can still provide significant security benefits. Firstly, high-risk components can be identified and be placed in separate compartments. Effective but high-overhead countermeasures [5,147] can be used to harden such compartments. By only applying these countermeasures to likely vulnerable compartments, their performance impact remains limited.

Secondly, Salus' ability to provide unforgeable references and it's ability to restrict access to system calls, can be used to enforce fine-grained access policies. Enabling a compartment to issue `open/close` and `read/write` system calls, essentially provides it access to the entire file system[3]. Alternatively, small, secure compartments can be created that provide similar support but may limit access to a specific folder. Since the compartment cannot issue `open` system calls herself, it can only access the file system through the received "capability" compartment (see Section 3.1 for an example).

Thirdly, compartmentalization can automatically thwart certain types of attacks. For instance, limiting entrance of compartments to valid entry points significantly reduces the chance of an attacker finding enough gadgets to successfully execute a return-oriented-programming (ROP) attack [23,111].

Fourthly, compartmentalization can be used as a building block for new countermeasures. For instance, a custom loader could be implemented that loads compartments at different locations in memory for every program execution.

[3]Of course this is restricted by the access rights the application is executing in

This is similar to address space layout randomization (ASLR) [13], but can be applied at a much finer-grained level.

Finally, even when a compartment has been successfully exploited, Salus can still limit the impact of the attack. Because Salus provides entry point enforcement, caller/callee authentication and system call privilege containment, an attacker will likely have to compromise multiple vulnerable compartments before reaching her intended target. This significantly increases the effort an attacker must take to successfully exploit the application. The ability to confine attackers to the exploited compartment even allows implementing a tightly controlled sandbox where user-provided machine code can be executed securely.

3.5.2 Performance Evaluation

To evaluate the performance of Salus, we performed micro- and macrobenchmarks. All tests were run on a Dell Latitude E6510. This laptop is equipped with an Intel Core i5 560M processor running at 2.67 GHz and contains 4 GiB of RAM. A Ubuntu Server 12.04 distribution with (modified) Linux 3.6.0-rc5 x86_64 kernel was used as the operating system.

System-wide impact

To show that legacy applications not using the modularization technique are not impacted by our changes to the Linux kernel, we ran the SPECint 2006 benchmark. All tests finished within ±0.4% compared to the vanilla kernel.

Microbenchmarks

To measure the overhead caused by switching the access rights, we created a microbenchmark that measures the cost of a call to a secure compartment and compare it to the cost of calling a regular function and calling a system call. The compartment used in the benchmark immediately returns to the caller. The system call and function behave similarly.

Table 3.2 displays the results of this microbenchmark. Calling a compartment is about 677 times slower compared to calling a regular function. This overhead is attributed to the need to modify the access rights of pages. Compared to calling a system call, the compartment is only 20 times slower. Due to these high costs, there is a trade-off to be made between a low number of compartment transitions and small compartments with additional security guarantees.

Table 3.2: Compartment access overhead.

Type	CPU cycles	Relative
Function Call	5,944	1
System Call	193,970	32.63
Compartment Call	4,024,227	677.02

Table 3.3: Requests per second of an TLS-enabled webserver where every TLS session is protected in its own compartment, for an increasing number of clients.

Concurrency	Vanilla	Salus	Relative perf.
1	109.11	96.54	-11.52 %
2	165.56	153.62	-7.21 %
4	184.31	164.78	-10.60 %
8	199.98	175.35	-12.32 %
16	206.82	181.00	-12.48 %
32	207.78	181.50	-12.65 %
64	206.64	180.35	-12.72 %
128	206.49	180.97	-12.36 %

Secure Web Server

As a macrobenchmark, we compartmentalized an TLS-enabled web server based on an example provided by PolarSSL library[4]. For every new connection a new compartment is created, securing session keys even in the event that an attacker is able to inject shellcode in the compartment providing its own TLS session.

The secure compartment was built using the PolarSSL cryptographic library and a subset of the diet libc library. A simple static 74-byte page is returned to the clients over an TLS-connection protected by a 1024-bit RSA encryption key.

We used the Apache Benchmark to benchmark this web server for an increasing number of clients that are concurrently requesting pages. The results are shown in Table 3.3. The performance overhead tops at 12.72% and is mainly attributed to the many compartment boundaries crosses during the TLS negotiation phase.

[4]https://polarssl.org/

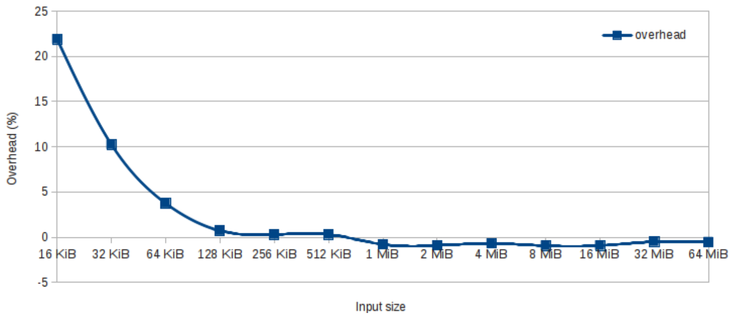


Figure 3.5: Salus' performance overhead on the gzip macro benchmark drops significantly as the input file size increases.

Compartmentalized parser

As input files are often under the control of an attacker and sanitation of their content can be difficult, parsers are a likely attack vector for many applications. As a second benchmark, we isolated the decompressing function of gzip (GNU zip). While disabling unused system calls for the entire process would result in similar security guarantees, we are interested in the impact of repeated compartment crossings in a parser setting. Applications that place their parser and the rest of the application in different compartments, would incur a similar overhead as only one additional compartment boundary needs to be crossed.

To benchmark the application, we created input files with randomized content, ranging from 16 KiB to 64 MiB in size, compressed them and measured the time taken to decompress the files with the hardened application. The application was run 100 times on each file. File I/O used a buffer of 32 KiB and the output was redirected to the null device. Figure 3.5 displays the results.

Given the relatively high overhead of a call to a compartment and the low computation cost of the decompressing function, it is unsurprising that for small input files the overhead can be as high as 21.9%. When the input size is increased however, the overhead drops steadily to -0.5% for 64 MiB input files, even though also the number of compartment-border crossings increases from 8 to 8200. We attribute this significant drop in overhead to the increased amount of slow disk I/O that needs to be performed as the input file size gets bigger, an effect that we predict to see in most parser-like compartments. The small performance gain of 0.5% can be attributed to cache effects.

The way an application is partitioned will have a significant impact on performance. Applications should be compartmentalized in logical blocks where each compartment has direct access to most of its required data. Once a logical block has finished, control and all data should be passed to the next compartment, reducing the number of inter-compartment calls. Smaller, heavily protected compartments such as an TLS compartment, provide strong security but may impact performance more significantly when called repeatedly. This makes the performance impact of compartmentalization difficult to predict. Therefore we advocate for automatic partitioning tools that analyze the application's call graph and information flow to reduce the number of compartment crosses and help the programmer decide which compartments should be hardened.

3.6 Related Work

Various security measures have been proposed to harden applications. Many of them aim to protect against very specific vulnerabilities such as buffer overflows [6, 42, 122, 146], format string vulnerabilities [31] or non-control data attacks [25]. While these countermeasures make it significantly more difficult for an attacker to compromise software applications, they cannot offer complete protection. Static verification of source code [60], in contrast, is able to provide such hard security guarantees, but typically comes at a significant economic cost in terms of programming and verification effort.

Singaravelu et al. [112] proposed to isolate security-sensitive parts of applications in complete isolation from the rest of the system. Many research proposals have since been filed based on this principle. Each of them provides some way of executing modules in isolation, relying on a trusted code base ranging from only a few thousands of lines of code [77, 119] to only the protected modules themselves and a small runtime library [11, 78]. More recently, specially tailored hardware support has been proposed in academia [87, 89, 120] and industry [7, 51, 81]. While these research prototypes offer provable security to the sensitive data that they protect [4, 96–98], they do not attempt to reduce the impact of a vulnerability elsewhere in the code by executing modules with the least amount of privileges possible [106]. An attacker who successfully gains control over the platform is still able to interact with other protected modules unrestrictedly.

Other work focuses on confining possible software vulnerabilities. Early work focused on reducing the size of the kernel itself [72], where process privileges are managed by capabilities. Recently Watson et al. [138] proposed applying a similar idea to partition applications themselves, where capabilities can be granted to each created partition. As partitions live in their own process, interaction takes place through remote procedure calls and passed data must be marshalled. Salus avoids these drawbacks by executing compartments in the same address space and unprotected memory can be used to gradually partition legacy applications (see section 3.3.6).

Provos et al. [100] and Brumley et al. [17] propose separating sensitive applications into a privileged monitor and one or multiple slave components. Monitor and slaves communicate through system sockets and thus also require arguments to be marshalled. Subsequent work by Provos [99] argues for finer grained access policies for system calls. Bittau et al. [15] also propose splitting applications into compartments (called sthreads) executing with least privilege. Developers can tag memory locations and a security policy enforces that a compartment can only access memory locations with a matching tag. When an sthread requires more privilege operations, it can request so by calling a callgate. A security policy enforces which callgates an sthread can call. Salus' unforgeable references enable a much more flexible security policy. Compartments can be provided temporary access to system resources by encapsulating them in a compartment. As all interaction to the resource passes by this compartment, the caller's access rights can easily be revoked at a later point in time [82].

Native Client (NaCl) [108,145], which builds upon the concepts of software fault isolation [137], takes another approach and attempts to completely sandbox x86 code. Accesses to the environment from within a sandbox are tightly controlled by runtime facilities. While NaCl focuses on downloaded, untrusted binary code, it could be used to partition entire applications. Interaction between two NaCl partitions is provided through a service similar to Unix domain sockets, making porting existing legacy applications a challenging undertaking. Salus on the other hand can provide a similar tightly controlled sandbox by placing such partitions in one compartment while the remaining legacy application is placed in another. A specially created wrapper can ensure that all system call privileges are revoked before execution control is given to the sandboxed code. There are however two major differences compared to NaCl. First, Salus only impacts performance when compartment boundaries are crossed. NaCl on the other hand places constraints on the binary code itself, resulting in a varying performance impact. Second, Salus employs a non-hierarchical separation of privilege, allowing compartments to be completely isolated from other compartments (possibly provided by other vendors) while compartments of the same vendor can co-operate easily.

Finally, our earlier work [119,120] is the most related to Salus. It also employs a program-counter based access control mechanism, but assumes a secure interface. Therefore it has the same limitation as other research prototypes [11, 77, 78] that provide strong isolation of sensitive data: it does not reduce the possible impact of exploited vulnerabilities.

3.7 Conclusion

Protected-module architectures isolate sensitive parts of applications. They guarantee that sensitive data can only be accessed via a well-defined interface. In practice, however, it is hard to isolate security-sensitive parts, as most code in an application is sensitive up to some level. As a result, modules of such platforms may need to provide insecure interfaces; an attacker may not access the sensitive data directly, but access to the provided interface may still lead to unwanted behavior.

We presented Salus, a new security architecture providing strong isolation guarantees of both sensitive data and software vulnerabilities. Salus significantly reduces the impact of insecure interfaces by (1) supporting the authentication of compartments and (2) enabling compartments to enforce that they can only be accessed through unforgeable references. This allows likely attack vectors and targets to be placed in different compartments, such that an attacker must successfully attack multiple compartments before an attack target can be reached.

3.8 Post-Publication Remarks

As already discussed in Section 2.8, hardware-based capability systems have been presented since the publication of Salus. Such systems are also able to prevent unrestricted access to “objects” (i.e., compartments [10], segments [144] or components [136]) and can be applied in similar use-cases as Salus. Moreover CHERI’s [144] and CODOMs’ [136] hardware implementation can provide much faster protection-domain crosses, at increased hardware costs. Revocation of capabilities however, is much more costly and requires scanning large memory ranges. The approach taken by Salus provides an interesting alternative, but has the disadvantages that it relies on the unlikelyhood that an attacker is able to correctly guess an access token.

Chapter 4

ICE: A Passive, High-Speed, State-Continuity Scheme

Publications & Acknowledgments

Strackx, R., Jacobs, B., and Piessens, F. ICE: A passive, high-speed, state-continuity scheme. *Accepted for publication in Annual Computer Security Applications Conference* (2014), ACSAC'14.

Strackx, R., Jacobs, B., and Piessens, F. ICE: A passive, high-speed, state-continuity scheme (extended version). CW Reports CW672, Department of Computer Science, KU Leuven, September 2014.

The main research effort of this work was lead by Raoul Strackx with the help of Bart Jacobs to provide a formal proof of correctness and the guidance of Frank Piessens.

Abstract

The amount of trust that can be placed in commodity computing platforms is limited by the likelihood of vulnerabilities in their huge software stacks. Protected-module architectures, such as Intel SGX, provide an interesting alternative by isolating the execution of software modules. To minimize the amount of code that provides support for the protected-module architecture,

persistent storage of (confidentiality and integrity protected) states of modules can be delegated to the untrusted operating system. But precautions should be taken to ensure *state continuity*: an attacker should not be able to cause a module to use stale states (a so-called *rollback attack*), and while the system is not under attack, a module should always be able to make progress, even when the system could crash or lose power at unexpected, random points in time (i.e., the system should be *crash resilient*).

Providing state-continuity support is non-trivial as many algorithms are vulnerable to attack, require on-chip non-volatile memory, wear-out existing off-chip secure non-volatile memory and/or are too slow for many applications.

We introduce ICE, a system and algorithm providing state-continuity guarantees to protected modules. ICE's novelty lies in the facts that (1) it does not rely on secure non-volatile storage for every state update (e.g., the slow TPM chip). (2) ICE is a passive security measure. An attacker interrupting the main power supply or any other source of power, cannot break state-continuity. (3) Benchmarks show that ICE already enables state-continuous updates almost 5x faster than writing to TPM NVRAM. With dedicated hardware, performance can be increased 2 orders of magnitude.

We present a machine-checked proof of ICE's security guarantees and evaluate a prototype implementation on commodity hardware.

4.1 Introduction

Protection of sensitive data in commodity computing platforms is extremely challenging. Modern operating systems provide process isolation primitives, but the kernel itself is too large to be implemented free from vulnerabilities. Moreover the operating system's functionality is extended continuously to support new file systems, process scheduling algorithms, peripherals, etc. Commodity systems are also prone to physical attacks, even by ill-equipped and resource-constrained home users [22, 50]. These vulnerabilities limit the amount of trust that can be placed in commodity systems. In servers these limitations are remedied by programmable hardware security modules (HSMs). On client devices, highly-sensitive applications such as online banking or e-government often resort to smart cards. Unfortunately, these solutions are expensive, cumbersome and the security guarantees that they can provide to the overall applications are limited.

Two recent advances in computer security indicate that this situation may change in the near future. First, protected-module architectures (PMAs) have been developed that provide strong isolation directly to modules running at application level [11, 21, 77, 78, 87, 105, 119, 120, 134]. The OS is still relied upon to provide services such as disk and network access, but they are *not* trusted. Protected modules' memory regions cannot be accessed from unprotected memory; modules are in complete control over their own content and can only be accessed through the interface they expose. Last year Intel disclosed Software Guard eXtension (SGX) [7, 51, 81], its hardware-implemented protected module architecture for commodity processors. SGX goes even further than other state-of-the-art protected-module architectures and also provides protection against hardware attacks; modules (called enclaves in SGX[1]) are only stored in plaintext within the CPU package. When they are evicted to main memory they are confidentiality, integrity and version protected.

Second, Agten et al. [4] and Patrignani et al. [98] proposed fully-abstract compilation techniques to such protected module architectures. While the strong isolation guarantees offered by these architectures is vital, they are difficult to implement without compiler support. Care must be taken not to introduce software vulnerabilities during compilation. Fully-abstract compilation ensures just this; machine-code-level attacks exists if and only if also a corresponding attack at source-code level exists. This enables easy reasoning and verification of the security guarantees these modules provide.

Unfortunately an important attack vector has been largely overlooked. Protected-module architectures, including SGX, only provide strong isolation guarantees *while the system executes continuously.* Without support for state continuity, protected modules need to remain stateless, significantly hampering their applicability. Consider as a running example a password-checking module. To defend against dictionary attacks, the user will be locked out indefinitely after three failed attempts. The module confidentiality and integrity protects its state before handing it to the untrusted operating system for storage. But when the module needs to recover its state after a reboot, it cannot distinguish between a fresh and a stale state and the guess-limited security measure cannot be guaranteed.

While having similarities with replay attacks at first glance, the state itself is replayed in a rollback attack. Providing support for state continuity is therefore much harder, especially when practical limitations are considered. Parno et al. [93] show that many seemingly obvious algorithms are flawed. Others are prone to simple hardware attacks. An uninterruptible power source (UPS), for example, may simply be disconnected. Or an in-kernel attacker may prevent the execution of the interrupt handlers the security measure relies upon. Adding non-volatile memory on-chip could simplify a solution, but requires

[1] We will use the term "protected module" when referring to isolated memory areas in *any* protected-module architecture and use "enclave" when referring to SGX specifically.

modification of manufacturing processes leading to increased manufacturing costs. Alternatively, using non-volatile memory off-chip (e.g., isolating disk space) may be susceptible to a clone attack where a hardware-level attacker may easily overwrite the state with a previously recorded stale state. Using TPM NVRAM or TPM monotonic counters instead, would foil such attacks, but would significantly impact performance and usability. Most implementations only provide 1,280 bytes of NVRAM that supports only 100,000 write cycles over the chip's lifetime [93]. Accessing NVRAM every second, would wear it out in less then 28 hours. Monotonic counters, on the other hand, only need to be incrementable every 5 seconds [132].

Hardware upgrades to the TPM chip could reduce some of these architectural constraints, at an economic cost. However, any solution placing the TPM on the performance-critical path, would require additional upgrades over time to bridge the ever growing TPM/CPU performance gap. We present ICE, an alternative solution that only requires TPM accesses at boot time and is thus *not* affected by TPM speed.

ICE avoids architectural challenges (1) by proposing a simple implementation technique where on-chip dedicated registers are backed off-chip by a capacitor and persistent memory. Upon a sudden loss of power, the contents of the dedicated registers is written to persistent memory. (2) ICE is a *passive* protection scheme; in the event of a crash or power loss, security is guaranteed instantly. A hardware attacker may disconnect the capacitor, but state continuity remains guaranteed. (3) At the moment freshness information is backed up to persistent storage, it is considered public data. Overwriting it with stale freshness information will be detected upon recovery.

In summary, we make the following contributions:

- We present ICE, the first algorithm providing state-continuity guarantees with a minimal TCB that does *not* rely on the speed of secure, non-volatile memory (e.g., the (slow) TPM chip) nor does it rely on an uninterruptible power source.
- We formally verify and machine check the security properties of ICE using the Coq proof assistant.
- Because SGX-enabled machines or emulators are not yet available, we validate our claims based on a prototype implementation on top of Fides [119], an existing hypervisor-based protected module architecture similar to SGX. Benchmarks show that states can already be stored almost 5x faster on commodity hardware than writing to TPM NVRAM. Dedicated hardware support would increase performance substantially.

- We provide new insights that can steer the future design of hardware security modules e.g., the TPM.

The remainder of this chapter is structured as follows. First we detail our attack model and the security properties that we need to guarantee. Next in Sections 4.3 and 4.4, we present our algorithm and discuss three possible implementations. Finally, we evaluate the security and performance of ICE and discuss how it can affect future directions of hardware security modules.

4.2 Problem Definition

4.2.1 Attacker Model

ICE can defend against an attacker with three powerful capabilities. First, we assume that an attacker is able to compromise the entire software stack, with the exception of ICE-implementing modules. This enables versatile attacks ranging from modifying the contents of the hard drive to preventing enclaves from ever resuming execution.

Second, we assume that an attacker has control over the system's power supply or is able to launch attacks leading to a similar result. Power-interruption attacks differ from kernel-level crashes as they also affect software modules executing in complete isolation from the rest of the system: modules may stop executing before they can commit their new state. SGX enclaves are especially vulnerable to such attacks. In order to prevent denial-of-service attacks by malicious enclaves that never return control to the kernel, SGX supports interruption of enclaves [57]. When the interrupt is handled in the untrusted kernel, an in-kernel attacker can easily prevent the enclave from ever resuming execution.

Third, we consider hardware attacks. We implement ICE as a library that modules can be statically linked with and take advantage of the security guarantees provided by the protected-module architecture. In case of SGX this implies that an attacker may place probes on memory buses or perform cold boot attacks [50]. Defending against physical attacks against the CPU package itself or the TPM chip [114, 126, 141] are orthogonal problems and not considered.

With respect to cryptographic capabilities of the attacker, we assume the standard Dolev-Yao model [37]: cryptographic messages can be manipulated, for instance by duplicating, re-ordering or replaying them, but the underlying cryptographic primitives cannot be broken.

We do not consider side-channel attacks in general (e.g., attacks based on cache behavior [90, 133]) but similar to Parno et al. [93] make one exception: we do consider attacks where an attacker prevents the module from recording its new state when she is able to infer (e.g., based on timing differences) that this state would be unpreferable. When input is given to a module, it should either complete the computation or no valuable information should be deducible from it.

4.2.2 Security Properties

State continuity can be factored into two properties: safety[2] and liveness. To ensure safety, ICE must be resilient against a *rollback attack* where an attacker provides the module with a valid, but stale state. A rollback attack is related to a replay attack but it is much harder to defend against. Where in a replay attack identical input is provided, the state of the module itself is replayed in a rollback attack.

The second property, liveness, states that benign events should never force the system into a state from which it cannot progress. In practice this means that the system should be allowed to crash at any time during the operation of the algorithm, including when it is recovering from a previous crash. Note that this is not the same as protection against denial-of-service. Protection against denial-of-service is not in scope; in-kernel attackers can easily prevent the system from progressing (e.g., by removing the fresh state from disk, or by breaking the kernel altogether). Liveness only ensures progress is not hampered by *random* crashes. This is important, since random crashes (or power loss) may occur even when a system is not under attack.

4.2.3 Applicability

ICE's high-speed state-continuity guarantees enable a large range of applications:

Fortified Applications. Almost all non-trivial applications need to keep some kind of state: login credentials must not be rolled back to a stale state, firewall settings must not be revertible and systems must be able to prove that stored log files are fresh and have never been tampered with. State-continuity guarantees can also enable more privacy friendly applications. Many use cases (e.g., road pricing or smart electricity meters) require sensitive data to be collected and

[2]We deviate from the definition of "safety" by Avizienis et al. [9] and follow the same terminology of closely related work by Parno et al. [93]

sent to a remote server. With strong hardware isolation, attestation and state-continuity guarantees, a (possibly malicious) user can download a software module that collects sensitive data and only sends an aggregated value to a remote party. Privacy sensitive data does never have to leave the user's system. We will discuss in Section 4.7 how low-end devices can also benefit from ICE.

A Building Block for Protocols. High-performance, state-continuous storage enable protocols to provide stronger security guarantees. Consider distributed algorithms as an example. Fault-tolerant algorithms have been proposed to reduce the impact of failing network participants (e.g., they may crash, process inputs incorrectly or their local state may get corrupted), but there is a theoretical upper bound that at most one third of the participants may be faulty. Chun et al. proposed append-only memory (A2M) [27] to harden existing distributed algorithms and applications such as NFS. Acting as a trusted log, this memory protects against equivocation; the ability of a network node to make contradicting statements to different entities. The authors propose hardened versions of PBFT [20], SUNDR [69] and Q/U [2], but leave implementation with a small TCB as future work. ICE is able to implement fast, append-only memory almost trivially.

Avoiding the TPM Chip as a Bottleneck. Many applications and protocols could also be implemented based on guarantees provided by the TPM chip [93]. Unfortunately the TPM was never designed with performance as a main requirement and a wide application of this approach would result in a severe bottleneck, especially in a server setting where each client connection requires TPM access. ICE avoids this bottleneck and other TPM constraints; a virtually unlimited number of monotonic counters of variable length can be provided, almost unlimited, never wearing-out NVRAM can be offered, etc.

4.3 State-Continuity as a Library

Before introducing a running example and describing ICE in full detail, we first introduce the system hardware we rely on and discuss how freshness information is recorded.

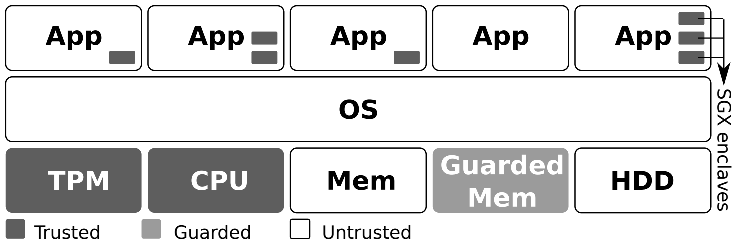


Figure 4.1: ICE provides state-continuity guarantees to isolated modules while trusting only a few key components.

4.3.1 Architecture

Assuming ICE is implemented on top of Intel SGX, we only need to place trust in the CPU package and TPM chip (see Figure 4.1). Attacks against any other component cannot compromise security.

Enclaves. Intel SGX, as any other protected-module architecture [11,21,77,78, 87,105,119,120,134], provides enclaves with total control over their own code and data by enforcing a specific access control mechanism; *only* when executing within the boundaries of an enclave can its content be accessed. Access attempts from code running at *any* privilege level outside the enclave (including from other enclaves), will be blocked. Enclaves can only be accessed through an interface they expose explicitly.

SGX can also make hardware attacks against enclaves significantly more challenging by ensuring that their content is only stored in plaintext inside the CPU package. The operating system may choose to write pages of enclaves to RAM memory or swap disk, but only after they are confidentiality and integrity protected. Freshness data is included as well to ensure that no stale pages can be swapped back in. However, when the system shuts down or goes into hibernation or sleep mode, enclaves are destroyed and this freshness information is lost [57]. Since enclaves live in the same address space and maintain their state between invocations, they can be seamlessly integrated in applications.

TPM. We store long term secrets and freshness information in TPM NVRAM. These secrets should only be accessible from the SGX enclave that provided them.

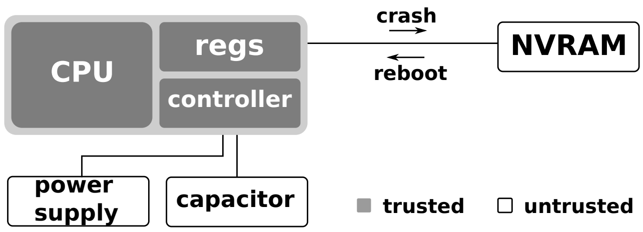


Figure 4.2: Architecture of guarded memory. When power suddenly fails on-chip dedicated registers are backed up to off-chip, shadow memory (NVRAM). *Only* on-chip components need to be trusted. Hardware attacks against NVRAM, main power supply or the capacitor cannot break state continuity.

Guarded Memory. To enable fast state updates, we propose the addition of a small amount of *guarded memory*; dedicated registers on-chip that are backed off-chip by shadow, non-volatile memory (NVRAM) and a capacitor (see Figure 4.2). When a controller detects that the main power supply is disconnected from the CPU package, it copies the registers' content to non-volatile memory. When power is re-applied, the controller restores the register's content from shadow memory. Note that *only* on-chip components need to be trusted. An attacker who gained full access to shadow memory nor one who is able to disconnect the capacitor or the system's power supply at any moment in time, cannot break state continuity.

The controller must also guarantee that guarded memory can be used to store sensitive data in a way that is inaccessible to an attacker. This is achieved by implementing an exclusive access mechanism. At boot time guarded memory is publicly accessible. The first enclave that requests exclusive access will receive it until the next reboot. From then on only that enclave can access guarded memory. Access requests from other enclaves or unprotected memory will be blocked. When power goes down, exclusive access is lost and data stored in guarded memory must be considered as being public.

Persistent Storage. ICE uses operating system services to access persistent storage. These services are not trusted: an attacker may copy, replace and destroy files. To differentiate between the actual state of a module and states stored on disk, we call the latter *(ICE) cubes* whenever ambiguity might arise.

4.3.2 Guards: Storing Freshness Info

Just as message authentication codes (MACs) can be used to guarantee message integrity, we will use *guards* to prove that a cube is fresh. Guards are 2-tuples:

$$\text{guard}_i(n) = (\underbrace{\text{Hash}^i(n)}_{\text{guard value}}, \underbrace{\quad i \quad}_{\text{guard index}})$$

where the first element, the *guard value* represents the hash value after hashing the base value i times, the *guard index*.

A guard is incremented by hashing the guard value and incrementing the index:

$$\text{guard}_i(n) = (\text{Hash}^i(n), i)$$

$$\text{guard}_{i+1}(n) = (\text{Hash}^{i+1}(n), i+1)$$

Based on the construction of guards, they possess two important properties: (1) two guards can be compared based on the guard index:

$$(n, i) \leq (m, j) \Leftrightarrow \begin{cases} n = m & \text{if } i = j \\ (\text{Hash}(n), i+1) \leq (m, j) & \text{if } i < j \end{cases}$$

and, (2) an attacker is unable to calculate any preceding guard as this would imply inverting the hash function.

4.3.3 ChkPassword: A Running Toy Example

Guaranteeing state-continuity is non-trivial and can only be accomplished by a module provider taking the required security precautions. We only provide a library offering state-continuous storage. To demonstrate the subtle vulnerabilities that need to be resolved, consider as a running toy example `ChkPassword`, a password-checking module displayed in listing 4.1. It exposes an interface of two functions: `set_passwd` that modifies the user's password and `check_passwd`[3] that handles login attempts. To prevent dictionary attacks, `ChkPassword` will lock out a user indefinitely after 3 incorrect attempts. We assume that when the module is created, the `INIT` function is called before any service call is handled. When `ChkPassword` executes on the platform for the first time, a default password is selected (line 7), otherwise its previous state is restored (line 10).

[3]Calling `ChkPassword` from unprotected memory would enable an attacker to intercept the provided password before it reaches the module. Users of `ChkPassword` should establish a secure channel from another module before exchanging sensitive data [7, 119]. Such considerations are out of scope.

```
   static int attempts_left;
 2 static char *password;

 4 void INIT( void ) {
     State *state;
 6   if (retrieve( &state ) == UNINITIALIZED){
         password = "default";
 8       attempts_left = 3;
     } else
10       restore_and_restart( state );
   }
12
   int ENTRY_POINT check_passwd(char *guess) {
14   State *state = new State();

16   //store (input, state) tuple
     collect_state( state );
18   collect_input( state, "CHECK_PASSWD" );
     collect_input( state, guess );
20   store( state );

22   //check passwd
     if ( attempts_left > 0 &&
24        strcmp( password, guess ) == 0 ) {
       attempts_left = 3;
26     return OK;
     } else {
28     attempts_left = max(attempts_left-1,0);
       return INCORRECT;
30   }
   }
32
   int ENTRY_POINT set_passwd(char *oldpwd, char *newpwd) { ... }
```

Listing 4.1: ChkPassword: A running example.

To ensure state continuity, `ChkPassword` needs to fulfill three requirements. First, it must protect against subtle timing attacks. When an attacker is able to infer that the provided password is incorrect based on timing differences between a correct and incorrect password,[4] she may be able to crash the system before the login attempt could be recorded. Ensuring that each execution path takes exactly the same amount of CPU cycles is hard. Similar to Parno et al. [93], we take a much simpler approach and store the state with the newly provided input *before* it is used in any computation. Hence, `ChkPassword` stores its current state (the number of attempts left and the correct password) together with the provided guess (line 17-20) before checking the provided password.

[4]A similar attack exists when a (unique) callback to unprotected memory is made before an undesirable state is stored.

An unexpected crash while the password is being verified (i.e., after line 20), will then result in the current state being restored and execution is restarted; another attempt is made to check the *same* provided password.[5] We assume `restore_and_restart` restores the current state and restarts execution of the last called entry point (line 10). Alternatively, if the system crashed before the input could be recorded and thus was never used in any meaningful computation (i.e., before line 20), the guess can simply be discarded.

Second, in order to guarantee that re-execution of the same input on the same state always leads to an identical result, modules must be deterministic. This implies that modules must consider all sources of non-determinism (e.g., the result of a random number generator) as input and thus store such data before using it in any computation.

Third, an attacker must not be able to infer any value from the size of the stored states on disk; modules must ensure that all cubes are equal in size.

4.3.4 ICE Libraries

We will provide state-continuous storage in two steps. In Section 4.3.4 we introduce `libice0`, a library providing support at the cost of scarce platform resources for every instance. Then in Section 4.3.4, we present `libicen` that alleviates resource pressure by storing freshness information in a single, state-continuous module `ice0`. As all `libicen` library instances connect to the same, unique `ice0` instance, a virtually unlimited number of modules is supported.

Both `libice0` and `libicen` provide the same interface: `store(State *)` and `retrieve(State **)`. To avoid repeated TPM or `ice0` accesses, `libice0` and `libicen` keep a cached copy. In order to distinguish between these copies and explicitly state where they are stored, we will reference them similarly to fields of a struct. For example, the encryption and MAC keys stored in the TPM chip will be referenced as `tpm.keys`. The variables used by the ICE algorithm are referenced as `ice.keys` and so on. Besides storing keys and the guard we also keep track of the state of the algorithm using a `mode` variable. Stored inside the TPM chip (`tpm.mode`), this variable indicates whether ICE was once initiated correctly. In `libice0` (`ice.mode`) this variable is used to indicate whether ICE was initiated or recovered since reboot. We assume that when a module is resurrected after a crash, `ice.mode` is initialized with value `Clear`. As a shorthand, we also assume that setting this variable takes exclusive access of guarded memory. Listing 4.2 uses these variables to differentiate between

[5]Decrementing the counter only after the password check failed does not impact the module's security properties for the same reason, assuming that the check is implemented securely.

```
void store( State *state ) {
  switch (ice.mode) {
    case Clear:
      return _init_state( state );
    case Activated:
      return _update_state( state );
} }

int retrieve( State **state ) {
  switch (tpm.mode) {
    case Clear:
      return UNINITIALIZED;
    case Activated:
      *state = _recovery_step();
      return RECOVERED;
} }
```

Listing 4.2: `libice0` relies on `tpm.mode` and `ice.mode` to distinguish between storing an initial state, updating a stored state and recovery.

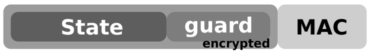


Figure 4.3: Stored states are confidentiality and integrity protected. Freshness is based on the enclosed guard.

an initial state being stored and a state being updated. Similarly, `tpm.mode` is used to determine whether a state was ever stored.

`libice0`: State-Continuous Storage for One Module

In order to provide state continuity, we must guarantee that an attacker is not able to fabricate recorded states (called cubes) and that no stale cubes can be provided as being fresh. The former is trivially guaranteed by including a message authentication code in each cube (see Fig. 4.3). Guaranteeing freshness is more challenging, but as modules maintain their state between invocations, we only need to consider power off and reboot events. Let's call events during such power cycles an *execution stream*. An execution stream starts by either storing an initial state of a module or when the state of a module is recovered after a crash. It ends when the system crashes or when it is shut down properly.

To keep track of the fresh cube, we will generate a (base) guard when the execution stream starts and store it securely in TPM NVRAM. For every state the module requests storage of in the current execution stream, we will increment the guard and include it in the generated cube. Using guarded memory we will

```
void _init_state( State *state ) {
  ice.guard = gen_guard();
  ice.keys = gen_keys();
  hdd.write( new Cube( ice.guard, ice.keys, state ));
  ice.mode = Activated;
  gmem.guard = ice.guard;
  tpm.guard = ice.guard;
  tpm.keys = ice.keys;
  tpm.mode = Activated;
}
```

Listing 4.3: `libice0`: Storing the initial state.

ensure that *only* the guard included in the last (and thus fresh) cube is leaked at the moment the system crashes. As no preceding guards were leaked (and cannot be calculated), it serves as a pointer to the fresh cube. Upon recovery, if the guard[6] that is stored in the provided cube can be provided to `libice0`, then this proves that the cube is fresh.

Creation of an Initial State. When storage of the initial state of the module is requested, a new base guard and keys are generated (see listing 4.3). Next, a new cube is constructed and written to disk. Exclusive access of guarded memory is taken by setting the `ice.mode` variable to `Activated` and the fresh guard is written to guarded memory. In case exclusive access cannot be assigned (i.e., another module already received it), the module simply stops its execution. For clarity, such error handling is not displayed. Finally the keys and guard are stored in the TPM's NVRAM and `tpm.mode` is set to `Activated`, committing the start of a new execution stream.

Updating a State. When storage of a new input-state pair is requested in the same execution stream, the previously used guard and keys are still stored in `libice0`'s memory and no TPM accesses are required. To safely store the input-state pair, a new cube is created with the subsequent guard and stored on disk (see listing 4.4). Finally the step is committed by writing the fresh guard to guarded memory.

Recovering from a Crash. Recovering from a crash is more challenging and is achieved in two steps (see listing 4.5, error handling is omitted for clarity).

[6] We must also check that this guard was created during the last execution stream as a matching guard/cube becomes public at the end of every execution stream.

```
void _update_state( State *state ) {
  ice.guard = ++ice.guard;
  hdd.write( new Cube( ice.guard, ice.keys, state ));
  gmem.guard = ice.guard;
}
```

Listing 4.4: `libice0`: Updating a state.

First, the last stored cube is read from disk. By verifying three properties its freshness is ensured:

- *Validity*: Cubes must not have been forged by an attacker. This is ensured by the MAC stored in each cube and the accompanying key stored securely in the TPM chip (line 17).
- *Correct execution stream*: The cube received from the untrusted OS must have been created during the last execution stream. At the start of each execution stream a new base guard is generated and stored in TPM NVRAM. All guards used during this execution stream are successors of this base guard. Hence, the cube was created during the last execution stream if and only if (line 18):

$$\text{tpm.guard} \leq \text{cube.guard}$$

- *Public guard*: `libice0` ensures that guarded memory always contains the same guard as the last (fresh) cube stored on disk[7], and that no preceding guards leak or can be calculated. Hence, if the guard stored in guarded memory matches the guard included in the cube at hand and the two previous properties hold as well, it is guaranteed that the cube is fresh (line 19).

In the second step the fresh state is re-stored as part of a new execution stream: `libice0`'s variables are restored from TPM NVRAM, a new base guard is generated, the fresh state packaged in a new cube and the base guard is written to guarded and TPM NVRAM memory. To ensure that after an unexpected crash during the execution of this step, recovery can be restarted, `libice0` must (1) backup the previous fresh guard before overwriting it in guarded memory. As this value is public, any persistent storage can be used (for clarity not displayed in listing 4.5). (2) The new base guard is written to TPM NVRAM as the last step.

[7]There is one exception as writing cubes to disk and updating guarded memory cannot be executed atomically. This exception is resolved later in this section.

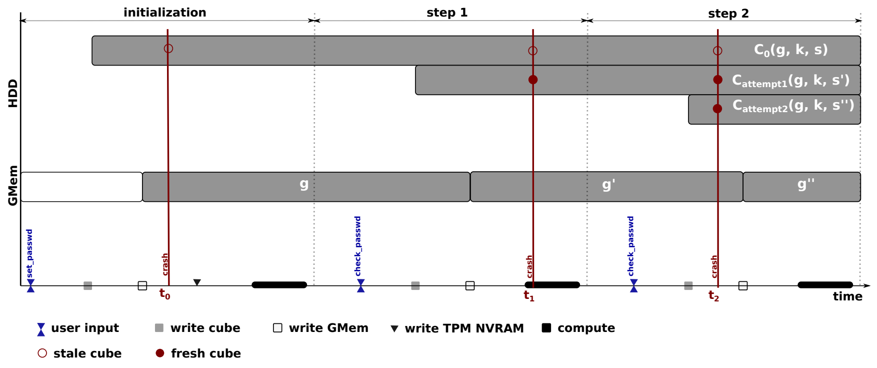


Figure 4.4: Graphical overview of the steps taken by a module storing its initial state and taking two steps in the same execution stream. Depending on the timing of a crash, three distinct situations can occur.

```
State *_recovery_step() {
  Cube cube = hdd.read();
  if ( is_fresh( &cube ) ) {
    State *state = extract( cube, tpm.keys );
    ice.guard = gen_guard();
    ice.keys = tpm.keys;
    hdd.write( new Cube( ice.guard, ice.keys, state ) );
    ice.mode = Activated;
    gmem.guard = ice.guard;
    tpm.guard = ice.guard;
    return state;
  }
  else abort();
}

bool is_fresh( Cube *cube ) {
  return ( check_mac( cube, tpm.keys ) &&
    tpm.guard ≤ cube->guard &&
    gmem.guard.value == cube->guard.value);
}

bool operator≤( Guard g1, Guard g2 ) {
  while ( g1.index < g2.index ) {
    g1.value = Hash( g1.value );
    ++g1.index;
  }
  return g1.value == g2.value;
}
```

Listing 4.5: `libice0`: Recovering from a crash.

Let's reconsider ChkPassword and discuss how crashes are resolved. Depending on the timing of a crash, we can differentiate between three main situations. Fig. 4.4 displays them graphically. One, ChkPassword was just created and the user called set_passwd to change the default password. This led to the execution of _init_state but the system crashes before tpm.mode could be set (see listing 4.3 line 9, t_0 in Fig. 4.4). When ChkPassword is re-created, it requests its previous state (listing 4.1, line 6). As tpm.mode still read Clear (listing 4.2 line 3), the module will restart from its default settings. As no input was ever used, state-continuity is guaranteed trivially.

Two, the system didn't crash when the user modified the module's default password and now calls check_passwd providing "attempt1" as password. After libice0 stores a new cube $C_{\texttt{attempt1}}$ on disk and updates guarded memory, the system crashes while the password is being verified (listing 4.1 line 23, t_1 in Fig. 4.4). The module is re-created and execution flow eventually executes _recovery_step (listing 4.5) As only a single cube is available containing the leaked guard from guarded memory (or a successor thereof), only cube $C_{\texttt{attempt1}}$ is considered fresh. After returning the stored input-state tuple in $C_{\texttt{attempt1}}$, ChkPassword will restore the attempts_left and password variables and execution is restarted with input "attempt1" (listing 4.1 line 10).

Three, assume that the previous password was incorrect and the user enters "attempt2" for her second attempt. After storing the new cube $C_{\texttt{attempt2}}$ on disk, the system crashes *before* the incremented guard could be written to guarded memory (listing 4.4 line 4, t_2 in Fig. 4.4). This is an interesting point of failure as both cubes $C_{\texttt{attempt1}}$ as $C_{\texttt{attempt2}}$ can be considered fresh[8]. However, recovery based on either will preserve state continuity. This is obvious for cube $C_{\texttt{attempt2}}$ as this is the latest cube written to disk. Recovery from $C_{\texttt{attempt1}}$, however will purge any record of the login attempt made using "attempt2". This is also safe as it was never used in any valuable computation (instructions after listing 4.1 line 24 were not executed yet). Hence, an attacker is not able to deduce any valuable information.

libicen: State-Continuous Storage for n Modules

By depending on scarce resources such as TPM NVRAM and guarded memory, libice0 can in practice only provide state-continuous storage to a limited number of modules. libicen will alleviate this strain by using a single, unique ice0 module to store freshness information on behalf of other modules. To

[8]The recovery step as displayed in listing 4.5, line 19, only accepts cube $C_{\texttt{attempt1}}$ as fresh. However, an attacker incrementing the guard stored in guarded memory, will trick libice0 to accept cube $C_{\texttt{attempt2}}$ as being fresh as well.

```
void _init_step( State *state ) {
  mod.guard = gen_guard();
  mod.keys = gen_keys();
  hdd.write( new Cube( mod.guard , mod.keys , state ));
  mod.mode = Activated;
  ice0.store( mod.id , mod.keys , mod.guard);
}
```

Listing 4.6: `libicen`: Initialization of a new module.

```
void _update_state( State *state ) {
  ++mod.guard;
  hdd.write( new Cube( mod.guard , mod.keys , state ));
  ice0.store( mod.id , mod.guard );
}
```

Listing 4.7: `libicen`: Updating a state.

safely exchange sensitive information between `libicen` and the `ice0` module, inter-module communication must guarantee endpoint authentication and confidentiality, integrity and freshness of messages. We will state this explicitly by passing a module identifier to `ice0` calls.

Creation of an initial state. Similarly to `libice0`, an initial state of the module is stored by generating a new guard and cryptographic keys and writing a new cube to disk (see listing 4.6). Finally the `ice0` module is requested to (state-continuously) store the keys and guard.

Updating a state. To update a state, `libicen` first writes a new cube to disk, before the updated fresh guard is stored in `ice0` (see listing 4.7).

Recovering from a crash. To recover from a crash, the (presumably) fresh cube is read from disk (see listing 4.8). Next, the keys and guard are requested from the `ice0` module. As the fresh guard is always stored safely in `ice0`, a cube with a correct MAC and that contains the fresh guard, must be fresh. Once the cube's freshness has been validated, `libicen` needs to generate a new guard, create and write a new cube to disk and store the new guard in `ice0` before a new step is taken.

The fact that a new guard is generated may be surprising since `libicen`'s guards never leak. But if this security measure is omitted, state continuity

```
State *_recovery_step() {
  Cube cube = hdd.read();
  ice0.retrieve( mod.id, &mod.keys, &mod.guard)
  if ( is_fresh( &cube ) ) {
    State *state = extract(cube.state,
                           tpm.keys)
    mod.guard = gen_guard();
    hdd.write( new Cube( mod.guard,
                         mod.keys,
                         cube.state ) );
    mod.state = Activated;
    ice0.store(mod.id, mod.keys, mod.guard)
    return state;
  }
  else abort();
}

bool is_fresh( Cube *cube ) {
  return check_mac( cube, mod.keys ) &&
    mod.guard.value == cube.guard.value;
}
```

Listing 4.8: `libicen`: Recovering from a crash.

cannot be guaranteed. Let's reconsider `ChkPassword` and show that if no new guards are created upon recovery, an attacker can create (fresh) cubes for every password in a dictionary attack and later tests them one by one. To explain the first step, recall that modules are required to first store input-state tuples before processing input. This enables an attacker to input a password, store the resulting cube on disk and then crash the system *before* the input is committed; the system keeps crashing before writing the new guard to guarded memory (listing 4.7 line 4). These instructions are repeated for every password in the dictionary. In the second step of the attack, the module is finally allowed to check a password. If it is incorrect, the attacker crashes the system. Upon recovery the fresh cube is requested from disk, but as all cubes contain the *same* guard value, all are considered fresh and another guess can be made. This example shows that seemingly obvious state-continuous algorithms may be susceptible to subtle bugs and should be formally verified.

4.4 Implementations

To be feasible in practice, any system providing state-continuous storage needs to be (1) small enough to allow formal verification (2) operate seamlessly with legacy software (3) incur a low performance overhead and (4) not wear out TPM

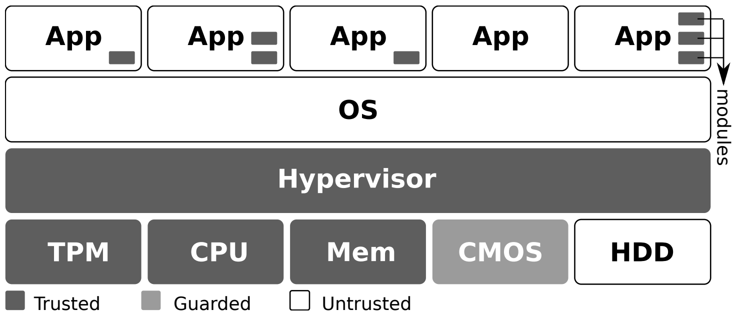


Figure 4.5: ICE can provide state-continuity guarantees to isolated modules of many state-of-the-art protected-module architectures on commodity hardware.

NVRAM. ICE is able to meet all these conditions, but depending on hardware support available, it is able to withstand different levels of hardware attacks. We describe implementations on platforms ranging from existing, commodity hardware platforms to distant future architectures.

4.4.1 ICE on Commodity Hardware

Given that SGX-enabled platforms will only become available in the near future, we implemented[9] a prototype of ICE on a commodity platform. A hypervisor-based protected-module architecture provided support for module isolation and we used CMOS memory as guarded memory. Obviously, since the hypervisor cannot prevent isolated modules to be evicted from the CPU cache to main memory in plaintext and CMOS memory is an easy attack vector, this implementation cannot protect against sophisticated hardware attacks.

Architecture

We opted to implement our prototype on top of Fides [119], a pre-existing PMA architecture. Its support for secure communication between modules enables an elegant implementation of `libicen`, where `ice0` can be implemented as a protected module. Alternatively, other protected-module architectures could implement similar secure communication primitives, or include `ice0` as part of the security platform. Fides' architecture with trusted/untrusted components are displayed in Fig. 4.5.

[9]Our research prototype is publicly available at https://distrinet.cs.kuleuven.be/software/sce/

While *any* non-volatile memory can serve as an alternative, CMOS memory is an interesting candidate for guarded memory. As it stores wall-clock time, it is updated every second and it must support a large number of write operations over its entire lifespan. Second, as it does not require a special communication protocol, it can be accessed easily and without much overhead. Being only accessible through direct I/O, it can also be isolated easily by hardware virtualization support.

Prototype Implementation

We added support to Fides for isolating and accessing CMOS memory. Using virtualization support to isolate programmed I/O, only 21 lines of code (LOC) had to be added to the hypervisor. Another 61 LOCs were required to implement system calls to access CMOS memory from the module. This totals the size of the hypervisor to 9,492 LOCs. While Fides at this moment does not support TPM chip accesses, we estimate, based on the Flicker [78] source code[10], that this straightforward effort would require an addition of less than 2,000 LOCs. As ICE only accesses the TPM at boot time, this does not impact performance.

While CMOS memory can be read/write accessed by the hypervisor, BIOS support for real-world implementations is required to allocate memory and exclude the area from its checksum to avoid that values written to CMOS memory are cleared on reboot. In practice, we must also ensure that the guard stored in CMOS memory is not lost when the system crashes while the previous guard is being overwritten. This can be solved by implementing a two-phase commit protocol where first a new guard is written before overwriting the previous one. In the event of an unexpected crash, the old guard may not have been overwritten yet and both guards leak to an attacker. However, this does *not* impact security as such an event is similar to an unexpected loss of power before the guard is updated; an attacker can easily calculate the new guard based on the one leaked. This situation is covered by the formal, machine-checked proof.

As available space in CMOS memory is BIOS-specific, some systems may have insufficient space to store two hash values. There are two options (1) they may use an alternative hash function with a smaller hash value [16] or (2) only partially store the hash value (e.g. the 2 least significant bytes). As the value is only compared to the expected value, only the `operator<=` function of `libice0` needs to be trivially modified (listing 4.5, line 27). To prevent an attacker from guessing a correct value, the number of recovery attempts can be tracked by storing a counter in TPM NVRAM. This counter can be decremented before the provided guess is tested, preventing a crash attack. Full support for state

[10]https://sparrow.ece.cmu.edu/group/flicker.html

Table 4.1: Breakdown of lines of codes for `libice0` and `libicen`.

	`libice0`		`libicen`	
	asm	C	asm	C
ICE	0	372	0	341
SHA-512	0	371	0	371
AES-NI	1,566	176	1,566	176
Total	1,566	919	1,566	888

continuity is in this case *not* required as the chances of repeated crashes during recovery are expected to be low in practice and thus losing a few recovery attempts is not an issue. To conservatively evaluate performance for a strong security implementation, we used the SHA-512 hash algorithm to create guard values. In Section 4.6 we will show that writing to CMOS is 5x more time consuming than `libice0`'s computations. Hence, writing less data to CMOS will have a positive impact on performance.

To implement `libice0` and `libicen`, we used the polarssl[11] library to calculate SHA-512 hash values and the Intel AES-NI reference implementation to take advantage of AES hardware support. This totals to 2,485 LOCs and 2,454 LOCs for `libice0` and `libicen` respectively (Table 4.1).

4.4.2 SGX-Based Implementation

By relying on SGX' guarantees that enclaves never leave the CPU package in plaintext, ICE can also withstand physical attacks but some security measures must be taken. First, dedicated hardware support for guarded memory is required. As proposed in Section 4.3.1, dedicated registers can be added to the CPU storing the fresh guard while the system is powered on. When power is suddenly lost, the content of these registers can be backed up to non-volatile memory using a small capacitor. Attackers who disconnect the capacitor or are able to read and write from/to non-volatile memory may prevent modules to ever advance again, but state-continuity remains guaranteed. Only dedicated registers need to be protected from inspection by a (hardware) attacker but as they are included in the CPU package, they share the same protection as enclaves residing in one of the CPU's caches. To prevent that the contents of these dedicated registers can be accessed by a (software-level) attacker, we propose a simple permission mechanism; the first enclave that requested access is granted it exclusively. Attempts to access these registers from any other locations will be prevented.

[11]http://polarssl.org/

Second, we must guarantee that the base guard stored in TPM NVRAM at the beginning of every execution stream, will not leak to an attacker. This can be established by setting up a secure channel from the `ice0` module to the TPM chip [132]. Authentication data can be sealed to the `ice0` module using SGX seal functionality [7, 57].

Third, in Section 4.3.4 we assumed that modules can easily and safely interact with one another. Fides supports such interaction explicitly by allowing modules to be authenticated and called from other modules. Unfortunately SGX enclaves cannot be called from other enclaves [57]. While this issue could be resolved with another hardware modification, it is not required. Using local attestation [7], enclaves can authenticate each other and set up a secure channel. As messages need to be passed in unprotected memory, they need to be confidentiality and integrity protected. A packet number must also be included to prevent replay attacks.

4.4.3 Distant Future Architectures

The wide deployment of TPM chips makes them a logical location to store freshness information and cryptographic keys over execution streams. Unfortunately, hardware attacks against the chip have been presented [114, 126, 141]. Moreover, the TPM chip is overly complex for our use case. This increases the risk of software vulnerabilities in the chip. We propose hardware replacements that provide stronger protection against physical attacks with only minimal hardware support.

To prevent ICE's cryptographic keys to leak to a hardware attacker launching offline attacks, we can replace it with a physically uncloneable function (PUF) [91]. PUFs are based on tiny variations in the manufacturing process of every individual hardware chip. This guarantees that PUFs are unique and are hard to copy. It has been shown that PUFs can be used to extract cryptographic keys. Their primary advantages over non-volatile memory, are (1) that they can only be read while they are powered and (2) physical tampering will destroy the PUF's intrinsic data. This makes them much more resilient against hardware attacks.

If we closely examine the `libice0` algorithm, we can observe that the base value of guards is always randomly selected. This enables it to be replaced by a reconfigurable PUF (rPUF) [67]. Similar to PUFs, rPUFs can be used to safely store random bits of data from an offline hardware attacker. But rPUFs have the additional functionality that they can be reconfigured; upon instruction the intrinsic secret data can be randomly modified. Once a rPUF has been reconfigured, it can never be reverted. Unfortunately, rPUFs still are a theoretical concept. Logical rPUFs have been proposed [61], where a PUF is combined with the contents of a non-volatile register. Only when the register contents is unchanged, will the logical rPUF return the same result. Upon reconfiguration, the register content is hashed and cannot be reverted. In contrast to "real" rPUFs, security sensitive information is still present after power-off and may be susceptible to attack.

While PUFs and rPUFs protect against a hardware attacker, we must ensure that malicious software on the platform cannot access these hardware primitives. This can be guaranteed by loading the `ice0` module on power-on [40] in memory and only allowing PUF/rPUF accesses from that memory region. In case the PUF/rPUF are implemented on a separate chip, a secure channel to the CPU needs to be established as well.

4.5 Security Evaluation

Since state-continuous algorithms must deal with sudden system crashes at any point in time, they are prone to subtle vulnerabilities. To guarantee ICE is indeed secure against a powerful attacker, we developed formal proofs of correctness.

4.5.1 Safety Properties

One of the properties that a safe state-continuous algorithm must guarantee, is that once a module starts computing with user-provided input, it must complete the initiated step or never advance at all. Provided input that was not yet used in any computation, however, may simply be discarded. To prove `libice0`'s safety, we first consider deterministic modules that only take their last state as input and later extend the proof to modules that also take non-deterministic (user) input. Finally we formally prove `libicen`'s safety guarantees.

Safety of `libice0` in the event of deterministic modules was proven by a machine-checked proof[12] with the Coq proof assistant [12]. The proof required 118 definitions, 201 lemmas and totals 37,726 lines. For non-deterministic modules and `libicen`, we created formal proofs, but leave machine-checks as future work.

[12]The correctness proof is publicly available at https://distrinet.cs.kuleuven.be/software/sce/

```
void ice_program( void ) {
 while ( true ) {
   switch (ice.mode, tpm.mode) {
     case (Activated, _):
       normal_step();
       break;
     case (Clear, Clear):
       init_step();
       break;
     case (Clear, Activated):
       recovery_step();
       break;
} } }
```

Listing 4.9: To prove ICE^0's correctness, we created a small application that will keep advancing a module.

Model

We modeled the state of a concrete system as a 7-tuple (T, N, I, H, P, t, g) where record T holds the contents of the TPM's secure storage. To be able to model cryptographically secure random numbers, a monotonic counter is also kept in T. N keeps the content of the guarded memory. Whether it can be accessed by an attacker depends on the mode of the countermeasure that is stored in the I record together with the current guard, cryptographic key and the current state of the module. H models the hard disk drive of the machine and stores the cubes. P models all public data, including cubes that were ever stored on disk and leaked guards from guarded memory. The algorithm itself is represented as a small program (see listing 4.9) that keeps advancing a module φ and is written as a term t. Finally a ghost element g tracks the last executed state and facilitates checking the proof.

Based on the state space $\mathcal{S}$ of these 7-tuples, we built a state machine with a step relation $S_M \subseteq \mathcal{S} \times \mathcal{S}$ where every step is either a program step or an attack step. A program step takes one evaluation step on term t. To ensure that our proof also holds in practice, we ensured that all instructions are feasible on commodity systems. For example, we do not consider that the TPM chip and guarded memory can be accessed simultaneously. We modeled the module requiring state continuity as a deterministic function φ. Whenever the module is executed with input state i, this argument is copied to ghost element g.

In an attack step an attacker is able to perform 4 operations:

- *Crash the system*: The system may crash at any point. This will (1) lift the protection of guarded memory and its contents becomes public knowledge. (2) I (i.e. `libice0`'s memory area) is cleared and does not leak. (3) `libice0` is restarted (term t is set to its initial value).

- *Modify HDD*: The contents of the hard disk drive may be modified by an attacker at any time. Cubes can be deleted, restored from public information in P, or cubes can be crafted by an attacker using publicly known cryptographic keys in P.
- *Modify guarded memory*: When the protection of guarded memory is down, an attacker is able to modify its contents and set it to any publicly derivable guard value. A guard $g' = (v', i')$ is publicly derivable from guard $g = (v, i)$ when its guard value v' can be computed from v:

  $$\exists n \in \mathbb{N}, v' = \text{Hash}^n(v)$$

  The guard index is not considered and can be chosen arbitrarily.
- *Use random number*: At any time an attacker is able to request a new random number from the TPM chip.

`libice0`'s State-Continuity Guarantees

To prove `libice0`'s correctness, we differentiate between modules that take non-deterministic input and modules that only operate on their last state.

`libice0`: State Continuity of Deterministic Modules. Before we discuss the proof in more detail, we first introduce some definitions. We define R^* as the reflexive-transitive closure of a state relation R i.e. $R^* = \cup_{n \in \mathbb{N}} R^n$. The image $R(X)$ of a set X under a relation R is defined as $R(X) = \{s' | \exists s \in X.(s, s') \in R\}$. We also define the composition $P; R$ of a state predicate P and a state relation R as $P; R = (P \times \mathcal{S}) \cap R$. Similarly, we define $R; P$ as $R; P = R \cap (\mathcal{S} \times P)$

To prove state continuity we use rely-guarantee reasoning and reason about reset, interference and program steps separately. We say a submachine with step relation S is *safe* under a precondition P, a rely condition R, and a guarantee condition G, denoted safe(P, R, G, S), if, when starting from a state that satisfies the precondition, all steps by the submachine satisfy the guarantee condition, assuming that all steps by the environment satisfy the rely condition:

$$\text{safe}(P, R, G, S) = (S \cup R)^*(P); S \subseteq G$$

Finally we define S_M as a step relation $S_M \subseteq \mathcal{S} \times \mathcal{S}$ where each step is either a program step or an attacker step.

Theorem 1 (`libice0`'s deterministic safety)**.** *We wish to prove the following:*

$$safe(\{s_0\}, Id, A, S_M)$$

where the precondition allows just the initial state s_0, the rely condition is the identity relation (since there is no environment), and the guarantee condition is the set of allowed steps. A step is allowed when `libice0` *either calls φ with its last output or it is a stutter step where φ is not called or it re-executes with the last input:*

$$A = \{(s, s') \in S_M | g(s') = g(s) \vee g(s') = \varphi(g(s))\}$$

First we separate reset steps from non-reset steps: $S_M = S_{\text{Reset}} \cup S_{\text{Nonreset}}$. We can do so using the following inference rule:

$$\text{Reset}\,\frac{\begin{array}{cc} S = S_{\text{Reset}} \cup S_{\text{Nonreset}} & I_{\text{Reset}}; R \subseteq R; I_{\text{Reset}} \\ I_{\text{Reset}}; S_{Reset} \subseteq G; Q_{\text{Reset}} & P \subseteq Q_{\text{Reset}} \\ \text{safe}(Q_{\text{Reset}}, R, (G; I_{\text{Reset}}), S_{\text{Nonreset}}) & Q_{\text{Reset}} \subseteq I_{\text{Reset}} \end{array}}{\text{safe}(P, R, G, S)}$$

Here, the reset postcondition Q_{Reset} is a state predicate that always holds immediately after a reset. It must also hold initially ($P \subseteq Q_{Reset}$). The reset invariant I_{Reset} is a state predicate that holds in every reachable state, i.e. it is preserved by the reset steps as well as the non-reset steps. It follows that a reset step always starts in a state that satisfies the reset invariant. The reset postcondition must imply the reset invariant.

We can further categorize the non-reset steps S_{Nonreset} into program steps S_{Prog} and interference steps S_{Itf}. During an interference step, an attacker may, for example, modify the contents of the hard disk, or of guarded memory when its protection is not enabled. Again, we wish to reason about these steps separately. We can do so using the following inference rule:

$$\text{Interference}\,\frac{\begin{array}{cc} S = S_{\text{Prog}} \cup S_{\text{Itf}} & I; S_{\text{Itf}} \subseteq G; I \cap G_{\text{Itf}} \\ I; R \subseteq R; I & P \subseteq I \\ \multicolumn{2}{c}{\text{safe}(P, R \cup G_{\text{Itf}}, G; I, S_{\text{Prog}})} \end{array}}{\text{safe}(P, R, G, S)}$$

Here, a global invariant I is established by the precondition P and maintained by all steps. Furthermore, interference steps satisfy an interference guarantee G_{itf}. Program steps are verified under a rely that is the union of the global rely and the interference guarantee.

Now that we have isolated the program steps, we wish to perform simple forward reasoning to verify these. For this purpose, we define the following auxiliary safety judgment:

$$\begin{aligned} \text{safe'}(s, R, G, S)_0 &= \text{true} \\ \text{safe'}(s, R, G, S)_{n+1} &= \forall s' \in R(s). \forall s'' \in S(s'). \\ &\quad (s', s'') \in G \wedge \text{safe'}_n(s'', R, G, S) \end{aligned}$$

We have the following inference rule:

$$\text{Program}\,\frac{\begin{array}{c} Id \subseteq R' \qquad R \subseteq R' \qquad R'; R' \subseteq R' \\ \forall n \in \mathbb{N}, s \in P \;.\; \text{safe'}_n(s, R', G, S) \end{array}}{\text{safe}(P, R, G, S)}$$

Note that safe' assumes that the environment performs a single R' step before every program step. Therefore, R' must subsume the reflexive-transitive closure of R. This is expressed by the first three premises of the inference rule.

If the program contains a loop, such as the `while`-loop in `libice0`'s main function (see listing 4.9), we can verify it as follows:

$$\text{Loop}\,\frac{\begin{array}{c} s \in I \\ \forall m \in \mathbb{N}, (\forall k < m, s' \in I \;.\; \text{safe}'_k(s', R, G, S)) \Rightarrow \\ \forall s' \in I \;.\; \text{safe}'_m(s', R, G, S) \end{array}}{\text{safe'}_n(s, R, G, S)}$$

Using loop invariant I we prove that, if the program is safe starting from I for less than m steps, then it is safe starting from I for m steps. This is a classical inductive proof.

The invariants that we had to come up with were reasonably big and contained a lot of information relating to how we modeled the secure random number generator. For example, we had to prove for every step that guards that were leaked all were created using the TPM's SRNG. This ensured that newly generated guards were not yet in the public domain. More interesting was the cube invariant stating that any cube in the public domain and that is seen by the algorithm as fresh, contains as module state either the ghost state g or the result of $\varphi(g)$. This led to a case split of the normal step, where a new state is stored that depends on the previous round of `libice0`'s main function (see listing 4.9) . Either the algorithm was initialized or took a normal step in the previous round, or has recovered from a crash. In the former case, fresh public cubes contain the last state (and equal the `libice0`'s internal memory) that was given to φ and the algorithm will advance the module to a new state. Alternatively, the last round was a recovery step in which case `libice0` may execute a stutter step, depending on when exactly the system was reset: before or after a new cube was written to disk and the step committed by writing the successor guard to guarded memory. In case the cube was not yet stored, fresh cubes contain the same state as is stored in the ghost state g. Also note that when the system was reset after the cube was written to disk but before the step was committed, both options are feasible since the contents of guarded memory is now publicly accessible and an attacker could complete the step.

`libice0`: State-Continuity of Non-Deterministic Modules. In the previous paragraph we proved state continuity for deterministic modules. Using an alternative formulation, we proved that when an application step φ is taken from application state a in machine state s and followed by any number of attack steps, then for the next application step it follows that either a new step was taken ($a' = \varphi(a)$) or the module re-executed the last step ($a' = a$):

$$s \xrightarrow{\varphi(a)} s' \xrightarrow{\text{att}}{}^{*}\ s'' \xrightarrow{\varphi(a')} s''' \Rightarrow a' = a \vee a' = \varphi(a)$$

Based on this theorem we can easily extend our model to allow non-deterministic inputs to the module. Given a module that operates on its previous state and (user) input, we prove that once it commits to an input, it will either use that input to advance its state or it will never advance (e.g., because the attacker crashes the system and erases the fresh cube).

Theorem 2 (`libice0`'s non-deterministic safety)**.** *More formally, consider a module φ that accepts input i on application state a and machine state s followed by any number of attack steps. When the module takes another step, it will either advance to the next state, or it will re-execute its previous step with the same input:*

$$\begin{aligned} s \xrightarrow{\varphi(a,i)} s' \xrightarrow{att}{}^{*}\ s'' \xrightarrow{\varphi(a',i')} s''' \Rightarrow \quad & a' = a \wedge i' = i \\ & \vee a' = \varphi(a,i) \quad (i' \in I) \end{aligned}$$

where I is the set of possible input values.

The core principle of `libice0`'s deterministic state-continuity proof is that we know that a state update is committed when we successfully incremented the guard in guarded memory. At that point we can feed module φ other input. We use this knowledge to partition the non-deterministic module φ as $\varphi = \varphi_i \circ \varphi_c$ where φ_i requests input from the user and φ_c deterministically computes a new state with the input state. By requesting the user's input after a computation step and storing it in a new state, only two possibilities arise. One possibility is that the system crashes after the result of φ_i was committed. In that case the input is stored and will be provided to φ in the next invocation where φ_c first computes the result. The other possibility is that the state was not committed. In that case φ_c will recompute the last state and request the user again for input. This does *not* violate state continuity since the input value is not used in any computation of the module. Hence, no information about the result of φ_c can leak to an attacker.

`libicen`'s State-Continuity Guarantees

While `libice0` only provides state-continuity properties for a single module, `libicen` supports a virtually unlimited number of modules. As we did for `libice0` we will focus on safety guarantees for deterministic modules. Support for non-determinism can be provided in a similar fashion as was explained in the previous paragraph.

Theorem 3 (`libicen`'s safety). *Given a deterministic module φ, we will argue that* `libicen` *ensures state continuity: φ will only re-execute the last step or take a new step. More formally we wish to prove:*

$$\mathit{safe}(\{s_0\}, \mathit{Id}, A, S'_M)$$

where we define S'_M as a step relation $S'_M \subseteq \mathcal{S} \times \mathcal{S}$ where each step is either a program step of `libicen` *or an attacker step. For s_0, Id and A, we use the same definitions as earlier.*

In order to prove `libicen`'s correctness, we could take the same approach as we did for `libice0`. Given the close similarity between `libice0` and `libicen`, however, we could also use the proof of `libice0` and reason that the modifications of `libicen` do not affect state continuity. We will take this approach using three consecutive transformation steps α, β, γ and prove that each transformation preserves state continuity:

$$\mathcal{S} \xrightarrow{\alpha} \mathcal{S}_1 \xrightarrow{\beta} \mathcal{S}_2 \xrightarrow{\gamma} \mathcal{S}_n$$

First consider transformation α that transforms a state $\mathcal{S}$ of `libice0` into a state $\mathcal{S}_1$ where TPM NVRAM is used instead of guarded memory. It is obvious that this does not affect the safety guarantees of `libice0` since contents of this memory will never leak to an attacker. Thus the attacker grows weaker instead of stronger.

After the α transformation, two guards are stored in the TPM's NVRAM: the base guard $(n, 0)$ and a guard $(\text{Hash}^i(n), i)$ used to determine which cube is fresh. Since neither of them can be modified by an attacker and the latter is always a successor of the former, the base guard can be omitted. The second transformation β will remove the base guard and modify the code of the recovery step. Since all changes happen within the module's memory area which cannot be influenced by an attacker in any new way, modifications of these instructions do not affect state continuity.

Finally consider an abstraction function γ that abstracts states $\mathcal{S}_2$ in states $\mathcal{S}_n$ where all interactions with the TPM's NVRAM are replaced with a call to an `libice0` module. Hence, we can rely on `libice0` to ensure integrity, confidentiality and state continuity of the stored data. It is also obvious that combining multiple steps into a single atomic step does not affect state continuity.

4.5.2 Liveness Properties

Given our strong attacker model, it is infeasible that *any* state-continuity algorithm on commodity hardware is able to guarantee that a module is always able to advance. An attacker could, for example, always delete the fresh cube. Such situations could be resolved in ICE by re-executing the initial step, at the cost of losing all previously stored states.

We do however wish to guarantee that progress can always be made in the event of benign events, such as a sudden loss of power during any step in the execution of the algorithm.

`libice0`'s liveness properties. To guarantee liveness under benign events, `libice0` needs to be able to recover from a crash during every step of its execution. An important distinction can be made based on the value of the `tpm.mode` field. This value indicates whether the algorithm has been initialized correctly. A crash before this value is set, will result in a re-execution of the initialization step. After setting this value, all crashes will result in the execution of the recovery step. To ensure that the initialization step may be re-executed in the event of a sudden crash, the `tpm.mode` value is set last.

After initialization we may update the state or we have to recover the fresh state. In the former case we make sure to first store the cube before we update the content of guarded memory. Recovery of a state is more challenging since we have to modify the guards in both guarded and TPM NVRAM memory. After creating a new cube with the module's fresh state and storing it on disk, we enable protection of guarded memory and write the new guard to it before we update TPM NVRAM memory. This has an important consequence: in case the system crashes before the recovery step is completed, the old guard may already have been overwritten. This would prevent the re-execution of the recovery step. Therefore we require that a backup of this guard is stored on disk *before* the recovery step is called.

`libicen`'s liveness properties. Ensuring liveness of `libicen` is straightforward since we only have to deal with two non-volatile data objects: cubes and calls to `libice0`. For obvious reasons we ensure to first store new cubes on disk. The `libice0` algorithm guarantees that its modifications are done atomically and are always retrievable.

4.6 Performance Evaluation

In this section we evaluate the performance of our prototype implementation. To compare the performance impact of a solid state drive (SSD) against a rotating hard drive (HDD), we used two machines with comparable hardware. The first machine, a Dell Latitude E6510, a mid-end consumer laptop, is equipped with an Intel Core i5 560M processor running at 2.67 GHz and 4 GiB of RAM. It is also equipped with a magnetic hard disk (HDD), a Broadcom TPMv1.2 chip and CMOS memory. The second testing laptop is a Dell Latitude E6520, has an Intel Core i5-2520M CPU running at 2.50GHz and is equipped with an SSD.

Hardware Benchmarks. To better understand the performance cost of ICE compared to TPM operations, we performed 4 benchmarks on the Latitude E6510: read/write accessing TPM NVRAM, extending PCR registers and generating random numbers. To perform these tests, we developed small TPM applications using the TrouSerS[13] open-source software stack. We also modified the tpm_tis driver to keep timing measurements. Each test was run 100 times and transferred 128 bytes to/from the TPM. Figure 4.6 displays the median time for each test graphically. All operations take a significant amount of time to complete. Especially writing to TPM NVRAM takes 4x longer than reading from it. Related work shows similar results for TPM chips from other vendors [93].

We also performed a similar benchmark on CMOS memory. We performed 10,000 one-byte write operations and measured the time using the `rdtscp` instruction. Writing to CMOS takes about 3μs/byte, significantly faster than writing to TPM NVRAM. We attribute this difference to the fact that CMOS memory

[13]http://trousers.sourceforge.net/

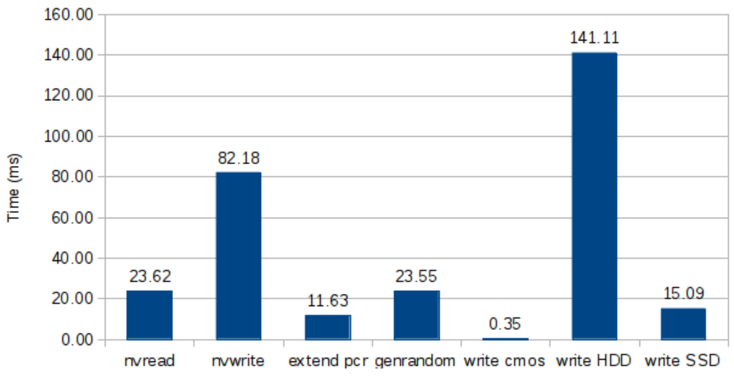


Figure 4.6: Microbenchmarks of various TPM operations show a significant difference in performance cost over CMOS and (SSD) disk accesses. Where applicable, 128 bytes were transfered.

is connected to the SPI-bus [56] and does not require a heavy communication protocol as does the LPC-connected TPM chip.

Finally, we measured the median time of writing 10,000 128 bytes files to both HDD and SSD disks. As Figure 4.6 shows, accessing the SSD disk is 5.4 times faster than writing to TPM NVRAM. Writing to a magnetic disk is more costly.

Microbenchmarks. To measure the performance of both `libice0` and `libicen` libraries, we implemented two modules. The first module implements a password verification function and limits the number of attempts that can be made before the user is locked out indefinitely. The benchmark provided this module with 10,000 wrong password guesses and measured the median time per guess. Measurements show (see Table 4.2) that for a single step only 0.06 ms (0.43%) were spent on computation when the module was linked with the `libice0` library. When we used `libicen`'s services, two cubes need to be created and computation time increased to 0.13 ms (0.71%). To securely write guards to CMOS memory, 0.33 ms were spent (2.17% and 1.82% for `libice0` and `libicen` resp.). This shows a much higher cost to write guards to CMOS compared to calculation time. But most of the time was spent committing cubes to solid state disk (97.40% and 97.47% for `libice0` and `libicen` resp.). `libicen` does *not* spend twice the amount of time writing cubes to disk. Cubes only need to be committed before a guard is incremented. Hence, `libicen`'s cubes can be stored temporarily in memory and transferred to disk together with `ice0`'s new cube *without* modifying the algorithm (see listing 4.7, lines 3-4), reducing disk access times.

While most TPM chips NVRAM area is limited to 1,280 bytes [93], it could be used to provide (state-continuous) storage to a single module to avoid disk overhead. To show that such a module would still benefit from ICE, we implemented a second benchmark called Noop. It does not perform any computation but only stores a state of 1,280 bytes. As expected given the performance of SHA-512 and Intel's AES hardware support, the increase in computation cost is negligibly with only 0.01 ms. As cubes are still smaller than disk sectors, costs of disk accesses are comparable to the Password benchmark. This totals the cost of storing new data in Noop at 15.05 ms to 17.65 ms for `libice0` and `libicen` resp; significantly faster than 82.18 ms to access TPM NVRAM. Finally we performed these tests on the Latitude E6510 which is equipped with a magnetic HDD. As expected, the cost of writing cubes to disk increased significantly and now accounts for 99.63%-99.74%. For both benchmarks `libicen` consistently takes more time writing cubes to disk than `libice0`. We attribute this behavior to the way we implemented its write function: merging `ice0`'s and `libicen`'s cubes takes us 3 `write` system calls before system buffers are flushed.

Expected Impact of Dedicated Hardware. These benchmarks show that only up to 0.14% of time is spent on computation. With dedicated hardware performance can be increased significantly.

Writing guards to CMOS memory is about 2.4 times more costly than computation and takes up to 0.31% of the time in case of a revolving HDD and up to 2.17% on our SSD testing platform. Hardware support for guarded memory, as described in detail in Section 4.3.1, would reduce overhead of this operation to almost zero.

But committing cubes to disk forms the real bottleneck, requiring up to 97.47% (for SSD) to 99.74% (for HDD) of the time. Recently Viking Technology [128] and Micron Technology [127] announced that they will ship capacitor-backed RAM to market. Operating similar to guarded memory, these hardware components contain fast, volatile memory that is written to flash memory when power is suddenly lost. Adding these hardware components to our system would eliminate disk access completely.

In summary, benchmarks show that our prototype implementation on commodity hardware already outperforms TPM NVRAM write operations by almost 5 times. Adding dedicated hardware support for guarded memory and capacitor-backed RAM, may even enable state updates 587 times faster than TPM NVRAM accesses!

Table 4.2: Microbenchmarks for `libice0` and `libicen` (time in ms).

	Password		Noop	
SSD (in ms)	`-lice0`	`-licen`	`-lice0`	`-licen`
computation	0.06	0.13	0.07	0.14
writing guard	0.33	0.33	0.33	0.33
writing cubes	14.61	17.42	14.65	17.19
total	15.00	17.87	15.05	17.65
HDD (in ms)	`-lice0`	`-licen`	`-lice0`	`-licen`
computation	0.06	0.12	0.07	0.13
writing guard	0.35	0.35	0.35	0.35
writing cubes	112.80	183.23	111.54	183.83
total	113.21	183.71	111.96	184.31

4.7 Implications Towards Hardware Security Modules

The TPM chip does not allow software modules to be executed within its protected boundaries. Instead, it is shipped with all supported security primitives stored in dedicated memory. This results in a number of drawbacks. First, as more resources are required, power usage and hardware costs increase. To minimize the costs of the PC platform, the TPM is equipped with a slow operating processor and limited NVRAM. Other specifications have been developed for various platforms such as the Mobile Trusted Module (MTM) for mobile devices. Unfortunately power and economic constraints still form an obstacle for low-end applications such as sensor networks. Second, it increases the possibility of software vulnerabilities in the TPM chip. Some vendors already struggle with its complexity [104] and as functionality is added, the specification may become even more complex.

Instead of using a separate chip, recent research shows that strong security guarantees can also be provided when modules execute on the same processor as untrusted, legacy software. Agten et al. [4] describe a fully abstract compilation scheme and proof that modules at machine code level can only be attacked *iff* vulnerabilities also exist at source code level. Furthermore, Noorman et al. [87] apply minimal hardware modifications to implement a protected-module architecture on a low-end processor.

ICE shows that with little additional hardware support, various security primitives can be implemented as protected modules. This has several benefits:

(1) By executing protected modules on the same processor as legacy software, overall power consumption and economic cost is reduced, enabling security measures to be ported to low-end devices (2) Software primitives can be added and updated after the chip was manufactured or deployed. (3) Hardware optimizations will increase performance of the overall chip, protected modules and untrusted software alike. (4) The overall complexity of the system is reduced. We believe that these advantages can affect future versions or revisions of hardware security modules, such as the TPM chip and the security of low-end devices in general.

4.8 Related Work

Most research prototypes do not consider state continuity, leaving them vulnerable to attack. Others propose special-purpose solutions without addressing resource constraints. We divide these research results into four categories: results proposing hardware modifications, results that isolate persistent storage, module-isolation architectures that only require a minimum TCB and special-purpose applications.

Hardware Modifications. XOM [70] protects against an attacker that is able to snoop buses and modify memory by encrypting data and code before it is sent to memory. While it makes it significantly more difficult to successfully attack the system, Suh et al. [123] argue correctly that it is vulnerable to a memory replay attack where stale memory pages are returned to the processor. Their Aegis architecture mitigates this replay attack by storing hash trees of memory pages in a secure location. When a memory page is loaded into the processor's cache, its freshness is checked by recalculating and comparing the hash values. Subsequent research results also defend against replay attacks [21, 57, 139].

Memory replay attacks differ from rollback attacks in that memory contents is replayed *while* the system is up and running. This enables much easier security measures.

Schellekens et al. [107] propose an embedded-systems architecture to store a trusted module's persistent state in invasive-attack-resistent, non-volatile memory. Their solution implements a light-weight authenticated channel between the trusted module and non-volatile memory. Freshness of the stored data is guaranteed per read/write instruction and based on a monotonic counter. As their approach assumes that write instructions to non-volatile memory and increments of the monotonic counter are atomic, unexpected loss of power enables a rollback attack. We believe that their approach can be fixed by keeping a log of instructions in secure non-volatile memory that need to be completed in case power suddenly fails. On higher-end systems however, only the TPM NVRAM can be used for such purposes and their approach would lead to significant performance overhead. ICE eliminates the TPM chip on the performance critical path altogether.

Research Systems Isolating Persistent Storage. Many architectures rely on a large TCB that includes isolation of persistent storage [47, 112, 125]. In such cases protection against rollback attacks are trivial: modules/programs can overwrite their state on disk. In practice however, software vulnerabilities in their TCB may be exploited and state-continuity support is hard to guarantee. These systems are also not able to defend against disk clone attacks. In contrast, ICE provides strong guarantees while only relying on a very limited TCB.

Protected-Module Architectures. In recent years many security architectures have been proposed that attempt to minimize the TCB by only providing strong isolation guarantees of modules [11, 57, 77, 78, 105, 119]. Persistent storage can only be accessed via services provided by the untrusted legacy operating system. None of them address the issue of state continuity.

Many of these systems can be adapted to use the state-continuity approach presented by Parno et al. [93]. This seminal work called Memoir, is to the best of our knowledge the first and only work that addresses the issue of state continuity in protected-module architectures. Based on Flicker [78], Memoir uses TPM NVRAM to store freshness information upon every state update. This significantly limits the applicability of their solution as NVRAM is slow and only required to support up to 100K writes. The authors acknowledge this constraint and propose two solutions: (1) adding capacitor-backed RAM to the TPM chip and (2) Memoir-Opt, an alternative approach that stores freshness information in (volatile) TPM PCR registers that are written to NVRAM when power is lost unexpectedly. Both solutions rely on an uninterruptible power source to safely store freshness when power suddenly fails. Failure in this mechanism can lead to a rollback attack. ICE, in contrast, is a passive state-continuity system that does not rely on an uninterruptible power source to guarantee security; detaching the capacitor would only prevent stateful modules from recovering their state but states could not be rolled back. Moreover, in ICE the speed of updates to state-continuous modules is only limited by the processor and (untrusted) non-volatile memory, not by the TPM chip.

Special-Purpose Applications. Chun et al. proposed the creation of append-only memory [27] to harden existing distributed algorithms to defend against equivocation: making different statements to different nodes in the network. An implementation with a minimal TCB was left as future work.

Levin et al. propose TrInc [68], a specialized system to attest successive monotonic counters, to achieve similar results. TrInc assumes a dedicated device that is able to locally store attestation requests of monotonic counters. In the event of an unexpected loss of power, clients can request the last signed attestations. This approach is similar to solutions where disk space is isolated, but incurs only a limited TCB. ICE provides a more generic, low-overhead alternative with only limited hardware modifications.

More recently Kotla et al. proposed a system [66] that allows offline data access while guaranteeing that (1) a user cannot deny offline accesses without failing an audit and (2) after proving that a user did not access the data, it cannot be accessed in the future. While their solution is interesting and does not require any software to be trusted, it only solves state-continuity in this specific setting.

4.9 Conclusion

Providing support for state continuity is challenging as including non-volatile memory on-chip requires modification of fabrication processes. But off-chip storage of freshness information can be slow (e.g. TPM NVRAM) or vulnerable to attack. We presented ICE, a state-continuous system and algorithm with two important properties: (1) only at boot time is the (slow) TPM chip accessed. State updates after the system booted only require updates to dedicated registers backed off-chip by a capacitor and non-volatile memory. (2) ICE is a passive security measure. An attacker interrupting the main power supply or any other source of power, cannot break state-continuity. We believe that the importance of ICE lies in the fact that it shows that with only limited and cheap hardware support, it enables the development of software-only implementations of trusted computing primitives. This presents an interesting direction for future versions or revisions of hardware security modules (e.g., the TPM chip) and may provide an interesting approach to increase security in low-end, resource-constrained applications such as sensor networks.

4.10 Post-Publication Remarks

Protected-module-enabled processor architectures such as OASIS [89] and Intel SGX [57] provide strong security guarantees to protected modules, including in the face of a hardware attacker. This leads to the question what is minimally required from external hardware security modules such as the TPM chip. ICE takes a valuable step to answer this question by eliminating high-performance requirements of external non-volatile memory. Our own research in progress takes another step by only relying on monotonic counters to provide state-continuity guarantees. With added hardware support for ICE, the TPM's slow monotonic counters could be used for these purposes and provide an easy migration path.

Chapter 5

Conclusion

World society ever increasingly depends on the security of computing devices. In recent years, e-learning, e-government and online banking have become commonplace. In the next decennia we can expect a sharp increase in e-health applications such as applications monitoring the user's heart rate and the distance she has run. Also networked embedded applications, what is commonly described as "the Internet of things", will result in many more security sensitive applications.

At the same time we have seen a shift in the goal of cybercriminals. In the '90s their primary aim was to disrupt systems and gain media attention. Today whole underground marketplaces have been created where vulnerabilities can be bought and sold, traffic to malicious websites can be traded, user credentials are auctioned, etc. Vulnerabilities are not exploited anymore solely to gain fame, but to make profit, spy on industrial competition and military intelligence.

The increasing reliance on computing devices and increasing sophistication of attacks, call for security systems with a strong, formal base. In this chapter we discuss the progress that has been made towards this goal during this PhD, elaborate on ongoing work and give our view on the long-term future of protected-module architectures.

5.1 Contributions

Protection of legacy computing devices is non-trivial. Many security measures are never applied in practice and/or fail to provide strong security guarantees.

The goal of this dissertation was to develop the required security primitives to create secure subsystems in existing devices that could become widely applied (see Section 1.3). We review each developed key primitive, discuss their weak and strong points and propose new research directions.

Isolation As an initial key primitive, we proposed [120] a program-counter-based access control mechanism to guarantee strong isolation of protected modules. An attacker who gains access to the system at application or kernel level, cannot directly access the module's memory region. Modules can only be called via an explicitly exposed interface and have complete control over sensitive data they store. In contrast to Flicker [78], SPMs maintain their state between invocations.

We developed two research prototypes based on this isolation mechanism with a minimal TCB. Sancus [87] targeted low-end, embedded systems – a platform whose security has previously been described as "a mess" [135] – and showed that the access control mechanism can be enforced with minimal performance, energy and die-size overhead. Fides on the other hand showed that by taking advantage of virtualization technology, the same access control mechanism can also be applied on commodity x86 platforms. While performance overhead of SPM/unprotected memory boundary crosses is obviously significantly higher than in Sancus, Fides can be used to easily fortify legacy applications today. Isolating every TLS connection in its own SPM instance, for example, would have protected against exploitation of the Heartbleed vulnerability [28, 83]; an attacker would still have been able to cause a buffer overread but as sensitive information of other connections are located in separate modules, sensitive information such as session keys or the server's long-term key would not have leaked.

Both prototypes assume a single-threaded, uninterruptable execution of SPMs and buggy or malicious modules that never return control to unprotected code can easily result in a denial-of-service. Various implementations have already been proposed to provide interruption support to Sancus [35] or similar PMAs [65] but its impact on fully-abstract compilation has not been fully addressed yet. When a module is called while its interrupt is still handled in unprotected memory, care must be taken to avoid race conditions. Variables may end up in an inconsistent state and invariants may be broken [96]. An easy solution would be to keep track of a `busy` boolean that is set when the module is entered and reset when control is (explicitly) returned to unprotected memory. When an attempt is made to enter the module while an interrupt is handled, the boolean is still set and the module can refuse to handle the call. While this would resolve the problem at hand, it is a crude solution and modules are still not truly multithreaded. More complete solutions may have to modify the isolation mechanism to provide special support for threads and synchronization.

Inter-Module Communication Secure communication between modules is paramount to build large, complex networks of protected modules. Placing all components in a single module is usually infeasible, since it would require complete mutual trust of all vendors involved. The proposed program-counter-based access control mechanism was specially designed with communication between protected modules in mind; SPMs can be authenticated and called easily from another SPM and secrecy and integrity of arguments passed in registers are trivially ensured.

Protected-module architectures provide strong security guarantees regarding secrecy of stored sensitive data. But an in-application attacker has unrestricted access to the provided interface and may launch versatile attacks. An X509 certificate-signing service implemented in a protected module, for example, could easily guarantee that its private key will never leave the module. An attacker with full access to its interface however, could request the signing of certificates for domains that she does not own. This still poses a major security threat [53]. In Chapter 3, we showed that secure subsystems could be fortified by only servicing requests made through an unforgeable reference. Only callers that ever received the capability (i.e., the unforgeable reference) to do so, can access the protected module. The presented Salus prototype, used this approach to compartmentalize legacy applications and isolate likely attack vectors from potential attack targets.

Current implemented prototypes handle inter-module communication graciously, but some aspects could still be improved. First, at the moment data can only be passed through registers. To transfer large amounts of data, it needs to be split into chunks small enough to be placed in registers and passed using multiple call and returns. Alternatively, the integrity and confidentiality protected data could be stored in unprotected memory and the cryptographic keys passed in registers. In a much more elegant solution, unprotected memory could be reserved and used as an extension of the register file; only the currently executing module can access the memory region and must ensure that all sensitive information is cleared before calling another module or unprotected code.

Second, calling modules must ensure that the called module is still present in memory. Current prototypes provide the required primitives, but when modules can be interrupted, these constructs may be susceptible to time-of-check-to-time-of-use (TOCTOU) attacks. In a much more elegant solution, a module identifier (e.g., the `spm_id`) of the called SPM could be passed in a dedicated register. When the platform detects that the called SPM is no longer (correctly) loaded in memory, an exception internal to the calling SPM could be thrown and handled accordingly.

Finally, the construction of unforgeable references could be improved. Current implementations rely on random numbers, but the creation of random number generators is non-trivial, especially when energy and die-size overhead need to be minimized. Explicit hardware support for capabilities, similar to data/address separation in some processors (e.g., the MC68000 [84]), could pose a viable alternative. When SPMs are created an access capability (i.e., a pointer to its location) could be placed in a specially marked "capability" register. Unless specified otherwise at creation time, SPMs are only accessible through capability registers. To ensure that content of such registers cannot be forged but only loaded from and stored to a newly-created "capability" section of SPMs, special checks need to be added to existing processor hardware instructions as well. Unfortunately this approach would have the same disadvantage as CHERI [144] and CODOMs [136] of requiring memory to be scanned (eventually) to revoke capabilities when SPMs are destroyed.

State-Continuous Execution Using strong isolation primitives and capability-based inter-module communication, subsystems can be built consisting of small modules with only limited mutual trust. Such subsystems are secure while they execute continuously but in practice systems crash, lose power or need to be rebooted. Only integrity and confidentiality protecting their state when it is written to disk is insufficient. An attacker who gained access to these states on disk, could execute a rollback attack. Similarly, in the event of a (possibly unexpected) loss of power, state continuity must be preserved and the system must always be able to advance. Both attacks are well within the attack model that only a specific, limited set of primitives need to be trusted.

We have presented ICE in Chapter 4, a state-continuity algorithm and architecture that provides state-continuity guarantees to protected-module architectures. Unlike related work [93], it does not reduce the attack model and avoids wearing out TPM NVRAM. With appropriate hardware support, ICE removes secure and non-volatile storage as a performance bottleneck allowing state-continuous, protected modules to be used on time-critical execution paths [116, 117].

Unfortunately, ICE assumes more than the minimal amount of security features required to create secure subsystems. Non-volatile memory must be available to store the initial guard and care must be taken that its contents is only returned to the protected module that initially requested storage.

We are currently working on an alternative with much fewer security requirements; only the state of a monotonic counter needs to be attestable. Its value is not sensitive information and even when incremented (at any point in time) by an attacker, state-continuity remains guaranteed. On commodity platforms a TPM monotonic counter could be used for such purposes. This makes the approach easily integratable with Intel SGX without reducing its attack model or requiring any architectural changes. As TPM implementations throttle the number of increments that can be made given a specific time period to avoid wearing out memory or rolling over the counter, a significant performance impact is to be expected. However, when used in combination with ICE, only the initial guard needs to be stored state-continuously and overhead is only incurred at boot-time. Subsequent state updates are handled by ICE and its performance remains unaffected.

Proven-Secure Modules Development of secure software is hard. Logical errors are easily made at source-code level, even by the best and most experienced developers. To make matters worse, some programming languages require developers to abide to strict rules. Non-compliance may not immediately lead to software crashes or incorrect results, but may pose a security threat. Buffer overflows and overreads are the most well-known examples of such vulnerabilities and were discussed in detail in Section 1.1.

Writing software for protected modules is even more challenging. An attacker who exploited an in-application or in-kernel vulnerability, may interact with the protected module at machine-code level. Care must be taken that no sensitive information can leak from the protected module, for example by lingering sensitive information in registers not used to pass return values. Security checks also must be inserted to enforce correct use of the exposed interface. Restrictions guaranteed by the type system at source-code level are no longer present after compilation and attackers may pass illegal values as arguments.

As a final key primitive, we presented a fully-abstract compilation mechanism that guarantees that such attacks will never lead to more powerful attacks than source-code level attacks; low-level attacks only exists iff also a high-level attack against the module exists [4]. This is a significant step towards verification of protected modules. Unfortunately not all aspects were addressed yet. Secure communication between objects, for example, has not been considered.

To ease the development of protected modules, we also implemented the security requirements in a fully-abstract tool chain. A fully abstract compiler was written that automatically inserts the required security measures. Also a linker was developed to layout the protected modules in memory according to the protected-module architecture used (i.e., Fides). But more low-level security checks had to be inserted as well. Consider as an example a protected module whose source code is split over multiple files. In such cases variables may be defined in one file while referenced in another. Such situations are usually handled using an added indirection. At compilation time code is generated that uses a Global Offset Table (GOT) to locate the correct variable. The GOT table is then filled by the linker, possibly at run-time. While this is a flexible solution when the location of variables may only be known at runtime (e.g., when the variable is defined in a library), this forms a security vulnerability in a PMA setting. A GOT table stored in unprotected memory may be overwritten by an attacker, and result in the use of unprotected memory to store a security sensitive variable. We solved this problem by instructing a secure linker to store a GOT table inside the SPM. When the module is called for the first time, the GOT table is initialized. Addresses of variables located within the compiled module are calculated based on the exact location of the module. Variables in unprotected memory are fetched from an untrusted GOT table and verified that their address lay outside the module's boundaries. We are confident that this solution solves the problem at hand, but it has not yet been formally verified.

5.2 Near Future

Next to continuing work on key primitives, new features and applications will be developed in the near future. We give a short overview of our expectations.

5.2.1 New Security Guarantees

Existing PMAs already provide strong security guarantees, but some applications require additional security primitives such as secure I/O and availability guarantees. We expect that these features will be added to PMAs in the next few years.

Secure I/O In Chapter 2 we presented Fides [119] and discussed its benchmark results of an TLS-enabled webserver. Secrecy of both the long-term cryptographic key as well as the session keys were guaranteed by the protected modules they were stored in. The untrusted TCP/IP stack was still used to set up connections with untrusted clients but the power of in-application or in-kernel attackers was reduced to that of a network-level attacker; exchanged messages could be dropped resulting in a denial-of-service, but any modification, re-ordering or replaying of network packages would be detected and handled accordingly.

Benchmarks showed that this setup is certainly feasible in a cloud-computing setting where a trusted client device wishes to connect to protected modules running on an otherwise untrusted server. But also a user may not fully trust all software running on her own device and in- and output peripherals connected to her system need to be able to communicate securely with protected modules.

This raises some interesting research questions. How can (existing) peripherals be used to set up secure channels to protected modules with minimal architectural or software modifications? Enforcing that I/O ports and memory mapped I/O regions can only be accessed via a specific protected module acting as a sentry, seems to be an obvious solution and this approach was already taken in Sancus [87]. The x86 architecture and the operating systems that run on them are much more complex than the embedded devices targeted by Sancus. Many attack vectors exists [79, 80, 150, 151]. For instance, I/O ports assigned to peripherals may be reconfigured by a compromised OS, interrupts may be spoofed, or vulnerabilities in firmware of connected devices may be exploited by attackers [24, 38]. Effective solutions will also have to inform the user which protected module will receive the sensitive information. Similarly, protected modules need to be guaranteed that the provided input originated from an input device and not from malware executing in unprotected memory.

An interesting approach currently taken by Noorman et al. supplies peripherals with a Sancus-enabled processor. A lightweight secure channel between the input device and a protected module is established using Sancus' attestation features. To prevent malware vendors from connecting their own, legitimate input device over a network and establishing "secure" I/O channels [46, 92], a (version protected) list of known connected peripherals must also be kept.

Availability Strong security guarantees can be provided when isolation of sensitive data and secure execution of code is ensured. But these security primitives do not suffice in all settings. Control mechanisms may also have to be timely executed. Consider as an example, a pharmaceutical company shipping medicine to drug stores. To verify that the shipped medicine never reaches a temperature above a specific threshold (which may have a negative impact on its effectiveness), each shipped container is supplied with an embedded device running a temperature-checking module. The timely execution of this protected module needs to be attested to the pharmaceutical company. Failure to do so, may leave exceedingly high temperatures to go undetected.

In Section 5.1 we already acknowledged that Fides' and Sancus' lack of support for interrupts and threads may result in a denial-of-service when malicious or buggy modules never relinquish execution control. We presented subsequent work in this area, discussed its good and weak points and highlighted possible solutions. But proper support for threads and interrupts may not only ensure responsiveness of the system, it may also guarantee timely execution of modules. A real-time scheduler can be implemented in a protected module that is given execution control every few milliseconds. Added protected "process" modules can request time quanta per time period and, when granted, are guaranteed that they will be dispatched for the requested amount of time. Protected process modules that call unprotected code may not finish execution within their provided time quantum and are preempted. Modules only calling other protected modules (if any) on the other hand, may abide by strict timing rules and can be used for time-critical parts of applications.

5.2.2 Writing Modules

When protected-module architectures become widely applied, one of the interesting questions that requires answering is how protected modules will be written. One option is to isolate sensitive parts of existing, legacy applications and place them in protected modules. The ability to access unprotected memory from protected modules significantly simplifies this effort and porting some legacy libraries[1] may be fairly simple. Other legacy code however, may exhibit repeated, complex protected-module-boundary crosses and strong security guarantees may be much harder to guarantee. Especially when the code is written in a low-level language, or when input needs to be stored before it is processed to ensure state-continuous execution. However, Salus showed that provable security guarantees may not always be required. The bar for successful attacks can be raised significantly by isolating possible attack vectors from likely attack targets.

Strong security guarantees can probably be provided much easier when protected modules are implemented in a higher-level language. Although not supported by research yet, we expect that even more complex security guarantees such as required for state-continuous execution can be enforced and hidden from the developer using a more expressive type system. A "state-continuity" monad, for example, could enforce that input-state tuples are always written to disk before functions are executed.

An interesting research approach would be to develop a common, intermediate language with built-in support for PMA primitives such as isolation. This would simplify the development of fully-abstract compilers and ease sharing of code with proven security guarantees between different programming languages.

[1]The PolarSSL TLS implementation, for example, has already been ported fairly easily [119].

5.2.3 Applications

That protected-module architectures – especially when they also protect against hardware-based attacks – can be used to implement software security-tokens with strong security guarantees, seems too trivial to mention. But with the expected arrival of Intel SGX in the near future, also some other interesting applications become feasible. We give a short overview.

Trust Assessment Modules Many security measures have been developed to prevent or detect attacks against vulnerabilities in legacy applications. Many of these require sensitive information to be stored out of reach of an attacker and resort therefore to a lower level: the kernel [85, 149]. HeapSentry, for example, places a canary before and after each allocated block of memory. The kernel verifies their correct value every few system calls. When an incorrect value is detected, a buffer overflow has occurred and the application is stopped.

A similar approach could be applied to protect the kernel by storing copies of guards at hypervisor level [129]. But also the hypervisor may be too large to be implemented free from vulnerabilities and may also require security measures against low-level exploits. It is clear that in practice the same approach cannot be applied in perpetuity. An interesting alternative would be to apply PMA features to increase security of the most privileged levels; guards could be verified by protected modules. The checking module itself can be expected to be small enough to be written securely. Hence, its integrity does not have to be checked, breaking the need of an ever more privileged layer.

For such a solution to succeed, an attacker must not be able to prevent the execution of protected modules. Architectural support guaranteeing timely execution of subsystems of protected modules, as discussed earlier, seems to be an obvious option.

Inverted Cloud Most Internet-based companies rely on advertisements and the sale of user profiles to turn a profit. At the same time, user's processors grow ever more powerful but run idle for long periods of time. Using strong security primitives, as provided by Intel SGX, an inverted cloud service could be built that uses client's processing power in exchange for services. This would significantly reduce e-waste, and reduce operational costs for companies.

Applications already exists that invite users to donate processing power but due to a lack of strong security guarantees, such applications can only send work packages of public data to clients and security measures need to be taken to ensure that returned results have not been manipulated. SETI@Home[2], for example, instructs two clients to execute the same work packages and compares their results. This unfortunately reduces the available processing power significantly.

An alternative architecture has already been proposed [118]. Salus' compartments are used to protect the client's machine from potentially malicious work packages. SGX' strong isolation and attestation primitives are on the other hand used to protect against malicious users. Unfortunately, a more detailed evaluation is still required. Are users willing to exchange processing power (and possibly battery life) for an ad-free experience? What is the latency of such a system? Can for any type of problem work packages be constructed?

Obfuscated Software The strong isolation properties provided by PMA architectures can not only provide strong security guarantees to sensitive data, but also to code. Protected modules could obfuscate code in two steps. First, the isolated generation of a public-private key pair is attested to a remote party. This key pair is used by the remote entity to integrity and confidentiality protect code and returns it to the protected module. In the second step, the protected module checks the received obfuscated code and starts its execution.

Intel SGX is especially well equipped for such use cases since it allows enclave pages to be simultaneously write and executable. Fides and Salus do not allow such behavior, but SPMs may implement an interpreter or, more stealthily, support Turing-complete [18, 23] return-oriented programming based on its own code.

Software obfuscation is sometimes used to harden application binaries against manipulation [30]. Removing limitations of a trial version or cheating in online games, for example, become much harder. It is also applied to implement digital rights management (DRM). Unfortunately, it can also be applied to implement malware that is much harder to investigate. Dunn et al. [39] showed that also TPM features already can be applied for the same purposes. However, widespread use of PMA architectures such as Intel SGX may significantly simplify these attacks.

An interesting research direction to prevent such malicious use cases, is the creation of verifiable proofs that code will never load and execute third party instructions. Only if such a proof can be presented and verified, should it be turned into a protected module. To allow legitimate use cases, we may have to resort to whitelisting (vendors of) trusted protected modules.

[2]http://setiathome.berkeley.edu/

5.3 Long-Term Ambitions

Long term predictions often fail miserably, but usually are interesting and entertaining to read – especially in hindsight. As a final section of this dissertation, we discuss possible long-term evolutions of PMA architectures but acknowledge that it is unlikely that they will take place outside the lab in the next decade, if ever.

One of the main advantages of PMA architectures is that they are evolutionary. Legacy applications can apply them to selectively harden security sensitive parts. Over time protected modules may form inter-connected secure "islands" whose numbers grow ever larger. The main question then becomes what the end result of this evolution will be.

One possibility is that some equilibrium is eventually reached. Security sensitive parts of legacy applications have become isolated and adding even more protected modules to applications may not be practically feasible. For example because the performance overhead would become too large, because isolating remaining parts of unprotected memory would be too cumbersome or simply because applications have almost become impossible to attack and there is no need to partition unprotected code any further. To which state legacy applications finally evolve, likely depends on the third-party libraries that are available and to the security guarantees they require.

Another possibility is that wide-spread PMA support leads to the development of modules written in a high-level language that provide proven security guarantees. Bindings to low-level languages could be provided so they can be easily integrated in many existing applications. More interestingly, these proven-to-be-secure modules could become important building blocks for security sensitive applications and become key reasons to continue development in high-level languages. Eventually new applications are written in high-level languages based on verified components.

Operating systems may follow a similar approach. First, security sensitive parts could be isolated. After which bindings with high-level languages could be provided to kernel module vendors. Eventually the entire kernel could be re-implemented in a higher-level language. This would also enable applications written in higher-level languages to share the same address space with one another and the kernel. The few legacy applications remaining, could be executed in a sandbox. Hunt et al. [54] already took a similar approach by implementing a complete kernel in a memory-safe language. While their approach is infeasible in practice given the huge amount of legacy kernel modules that would have to be discarded, it shows an interesting goal to strive for.

Many features are much easier to support when the operating system and applications are written in memory-safe languages. Inter-application communication, for example, becomes as trivial as calling an object as processes can share the same virtual address space. Also the kernel/user space separation can be lifted, making communication with the kernel and kernel modules much easier.

Before such systems could become reality, many research questions still need to be resolved. How can free and allocated memory be tracked, for example? Applications that finished their execution should relinquish all allocated memory. This becomes more complex as certain parts of the applications may still be shared by other applications. Shared objects could simply be discarded, but calls to non-existing objects should be handled graciously. Other questions involve how objects should be created. Should a single protected module be created per object, or should objects of the same class all be stored in a single protected module? Another research question that requires answering is how objects could be swapped out of memory to swap disk. ICE may be applied in such situations, but how should available memory be managed? How can objects be selected to be swapped to disk with a minimal amount of overhead? A full-hardware implementation as used by Sancus may be too rigid in such situations. A security-kernel approach as used by Fides in combination with limited hardware support, may enable a much more flexible design but comes at the cost of a software TCB.

Appendix A

Intel Software Guard eXtensions

In June 2013 Intel publicly disclosed its work on Software Guard eXtensions (SGX), its own x86 hardware support for a protected module architecture. This work shows some remarkable similarities with already published research but has some features that were much more advanced than state-of-the-art work in academia. We give a very short overview on Intel SGX and how it relates to work discussed in this thesis. The interested reader is kindly referred to the SGX programming manual [57] and related papers [7, 51, 81] for further details.

Like any other protected module architecture, SGX offers strong security guarantees against a software-level attacker. It supports the creation of strongly isolated protected modules – called *enclaves* in SGX terminology – that live in the same address space as the application that created them. SGX enforces that an enclave's memory region is only accessible from inside the enclave. Any access attempt from unprotected memory or from another enclave, will result in a memory access violation. This enables transparent integration in applications as input arguments do not have to be marshalled, but can simply be passed as a pointer to unprotected memory. Also similar to Fides and Sancus, enclaves maintain their state between invocations.

In contrast to most other PMA architectures, Intel SGX also considers hardware-level attacks against the entire platform but the CPU package. Most notably, it assumes that an attacker has direct, physical access to main memory and is able to snoop on memory buses. To prevent an attacker to modify enclave content or extract sensitive data while stored in main memory, enclave memory is stored in plaintext only in CPU cache within the CPU package. Before it can be off-loaded to main memory or swap file by the untrusted OS, it is confidentiality, integrity and version protected. Support to state-continuously store enclave states between reboots or power-down, however, is *not* available.

SGX's support for attestation of enclaves and sealing of sensitive data to a particular enclave, is also closely related to academic work, but some novel features were added. For instance, data can be sealed to an enclave's sealing identity, enabling future versions of the enclave to access data stored by older versions. SGX also provides a more privacy friendly attestation technique based on a special "quoting enclave" and a group signature scheme, avoiding that an attestation can uniquely identify a platform or link multiple attestations. At the core of attestation and sealing primitives however is a key derivation technique very similar to Sancus. The used cryptographic keys are derived from the enclave identity – its layout and initial content – and a key unique to the platform. Unfortunately, as enclaves cannot directly call other enclaves, these hardware primitives must also be used to implement intra-platform communications between enclaves. This has the significant downside that exchanged messages must be confidentiality, integrity and version protected and that, if required, called enclaves must signal the receipt of messages to their sender.

When details about SGX were made public, some of its features were clearly more advanced than state-of-the-art work on protected module architectures of the academic research community. For instance, SGX does provide support for interruption of enclaves. This prevents a badly implemented or malicious enclave from hogging the CPU and ensures that availability of the system remains under full control of the legacy operating system.

Another notably difference is that enclaves support multi-threading: enclaves can be executed on multiple cores or multiple hardware threads simultaneously. This is beneficial when the provided enclave functionality is called from multiple threads and queuing execution requests would be too time-consuming.

Appendix B

Protected-Module Architectures vs Microkernels

The operating system's core responsibility is simple: provide applications with a sound, abstract view of the underlying platform. For instance, applications should be presented with a uniform, virtual address space. Their view on memory should not depend on which physical memory regions it got assigned nor on other applications that also happen to be present in memory.

The operating system must also ensure that these abstractions cannot be broken. A buggy or otherwise misbehaving application must not result in the crash of the entire system, nor should it affect the correct execution of other applications or the kernel itself.

This problem has received much research attention and resulted in two main approaches. Monolithical kernels implement most abstractions in the same address space and privilege ring (i.e., kernel space). The kernel's functionality can be extended with kernel modules, for example, to implement a new file system or to provide communication with newly added peripherals, but no isolation[1] between these modules is enforced.

Microkernels take the complete opposite approach. The kernel itself only provides isolation of processes, inter-process communication (IPC) and basic scheduling. All other abstractions are implemented by isolated processes executing in user space. In contrast to monolithical kernels, this approach

[1]Some research prototypes exist that protect the kernel against buggy kernel modules and provide some kind of light-weight isolation [124]. These architectures however, do not provide strong security guarantees and do not protect against malicious kernel modules.

ensures that vulnerabilities in one process do not automatically affect the rest of the operating system.

Microkernels and PMA architectures share key properties. Both provide strong isolation guarantees and support secure local communication. But microkernels have never been widely applied. Why won't PMAs share the same fate?

B.1 Why Microkernels Failed

From a research perspective, microkernels are clearly more secure than monolithical kernels. There are however, a number of practical limitations that prevented them from becoming pervasively applied.

First, in early microkernel implementations inter-process communication, one of the most important mechanisms of microkernels, incurred a significant performance overhead. Later versions managed to reduce this overhead 20 fold [71, 72], but microkernels already got a bad reputation of being too slow.

Second, microkernels are difficult to program. All required abstractions need to be placed within their own process executing in their own address space. Consequently, when processes need to be called, their arguments need to be marshalled. This makes sharing data between processes much more difficult. Processes need to keep their own copy and care must be taken to keep all these copies coherent. This virtually turns the microkernel into a distributed system [131].

Monolithical kernels, on the other hand, don't share this fate. Kernel and all kernel modules execute in the same address space and data can be shared as easily as passing a pointer. An unfortunate consequence of this is that monolithical kernels may be exploited more easily. However, security at that time ('80s and early '90s) was of less importance. Buffer overflow vulnerabilities, for example, only got major attention after the Morris worm in 1988 [113]. In addition, monolithical kernels that enforced strong isolation of kernel and user applications already showed to be relatively robust. Earlier processor and operating system architectures could not enforce such isolation and a misbehaving application often resulted in a complete system crash. It seemed feasible that (eventually) security measures could be developed that would significantly raise the bar for attackers and may even make attacks practically infeasible.

Third, monolithical kernels simply already existed. When Liedtke in 1993 [71] finally presented how fast IPC calls could be practically implemented, the market of personal computers was already booming. MS Windows 3.11, for example, was released in the same year. Soon MS Windows dominated the market, luring new users and applications vendors to a single, monolithical operating system. Fast, secure microkernels may simply have arrived too late.

B.2 Why PMAs Won't Share the Same Fate as Microkernels

Even though PMAs share key properties, we don't expect that they will share the same fate as microkernels for a number of reasons. First, microkernels required a complete re-implementation of the kernel and applications to make optimal use of the provided isolation mechanisms. Early implementations transformed monolithical kernels into microkernels but this was one of the causes of their slow IPC performance [71]. PMAs on the other hand, can be integrated easily in existing, legacy platforms. Operating systems do not have to be re-implemented and legacy applications that do not take advantage of the provided security properties, do not incur any performance overhead.

Second, PMAs can provide provably strong security guarantees to selected sensitive parts of applications. Easy integration with legacy applications is ensured as protected modules execute in the same address space. Consequently, input arguments can simply be provided as pointers to unprotected code. No marshaling is required.

Third, current PMA research projects focus not only on providing the required security primitives, but also on how they can be easily applied. We expect that future development of fully-abstract compilation of higher-level languages, make it almost trivial for developers to write programs with strong security guarantees.

Fourth, user's expectations have changed. Recent years security sensitive applications such as e-banking have been widely adopted by the general public. Simultaneously, far-reaching security breaches have become public. Vendors of security-sensitive applications will have to start focusing on security of their own systems as well as their clients'. Strong, hardware-based security guarantees may become a key selling point for application vendors. Recent public announcements by Intel shows that processor vendors are willing to invest in hardware security technology. Architectural changes to provide strong security guarantees with minimal overhead, (e.g., guarded memory as presented in Chapter 4) may be feasible. With Intel SGX, a significant step towards much more secure systems will already be taken in the near future.

Bibliography

[1] Abadi, M., and Plotkin, G. D. On protection by layout randomization. In *Proceedings of the 25th Computer Security Foundations Symposium* (Los Alamitos, CA, USA, 2010), CSF'10, IEEE Computer Society, pp. 337–351.

[2] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, Oct. 2005), SOSP'05, ACM, pp. 59–74.

[3] Agten, P., Jacobs, B., and Piessens, F. Sound modular verification of c code executing in an unverified context. In *Accepted for publication in Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)* (Jan. 2015).

[4] Agten, P., Strackx, R., Jacobs, B., and Piessens, F. Secure compilation to modern processors. In *Proceedings of the 25th Computer Security Foundations Symposium* (Los Alamitos, CA, USA, 2012), CSF'12, IEEE Computer Society, pp. 171–185.

[5] Akritidis, P., Costa, M., Castro, M., and Hand, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium* (2009), SSYM'09, USENIX Association, pp. 51–66.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack magazine 7*, 49 (1996).

[7] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), vol. 13 of *HASP'13*, ACM.

[8] Appel, A. W. *Compiling with Continuations.* Cambridge University Press, New York, NY, USA, 1992.

[9] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (Jan. 2004), 11–33.

[10] Avonds, N., Strackx, R., Agten, P., and Piessens, F. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *Security and Privacy in Communication Networks (SecureComm'13)* (Sept. 2013), T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, Eds., vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer International Publishing, pp. 252–269.

[11] Azab, A., Ning, P., and Zhang, X. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), CCS'11, ACM, pp. 375–388.

[12] Bertot, Y., and Castéran, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[13] Bhatkar, S., DuVarney, D. C., and Sekar, R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX security symposium* (Berkeley, CA, USA, 2003), vol. 12 of *SSYM'03*, USENIX Association, pp. 105–120.

[14] Billington, R. A., and Ridge, M. *American History After 1865.* Rowman & Littlefield, 1981.

[15] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 309–322.

[16] Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varici, K., and Verbauwhede, I. Spongent: a lightweight hash function. In *Proceedings of the 13th international conference on Cryptographic hardware and embedded systems* (Berlin, Heidelberg, 2011), CHES'11, Springer-Verlag, pp. 312–325.

[17] Brumley, D., and Song, D. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Conference*

on USENIX Security Symposium (Berkeley, CA, USA, 2004), vol. 13 of *SSYM'04*, USENIX Association.

[18] Buchanan, E., Roemer, R., Shacham, H., and Savage, S. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)* (New York, NY, USA, 2008), CCS '08, ACM, pp. 27–38.

[19] Carter, N. P., Keckler, S. W., and Dally, W. J. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1994), ASPLOS'94, ACM, pp. 319–327.

[20] Castro, M., and Liskov, B. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), vol. 99 of *OSDI'99*, USENIX Association, pp. 173–186.

[21] Champagne, D., and Lee, R. Scalable architectural support for trusted software. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture* (Los Alamitos, CA, USA, 2010), HPCA'10, IEEE Computer Society, pp. 1–12.

[22] Chan, E. M., Carlyle, J. C., David, F. M., Farivar, R., and Campbell, R. H. BootJacker: Compromising computers using forced restarts. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2008), CCS'08, ACM, pp. 555–564.

[23] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS'10, ACM, pp. 559–572.

[24] Chen, K. Reversing and exploiting an apple firmware update. In *Blackhat USA* (2009), pp. 1–9.

[25] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2005), vol. 14 of *SSYM'05*, USENIX Association, pp. 177–192.

[26] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. K. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th conference on architectural support for programming languages and operating systems* (New York, NY, USA, 2008), ASPLOS'08, ACM, pp. 2–13.

[27] Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP'07, ACM, pp. 189–204.

[28] Codenomicon. The heartbleed bug. http://heartbleed.com, Apr. 2014. Accessed: 2014-11-05.

[29] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009, pp. 23–42.

[30] Collberg, C., and Nagra, J. *Surreptitious Software – Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Addison-Wesley Professional, 2009.

[31] Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., and Lokier, J. Formatguard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2001), SSYS'01, USENIX Association, pp. 1–9.

[32] Cowan, C., Beattie, S., Day, R. F., Pu, C., Wagle, P., and Walthinsen, E. Protecting systems from stack smashing attacks with stackguard. In *Proceedings of Linux Expo* (May 1999), pp. 1–11.

[33] Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium* (Berkeley, CA, USA, 1998), vol. 81 of *SSYM'98*, USENIX Association, pp. 346–355.

[34] Datta, A., Franklin, J., Garg, D., and Kaynar, D. A logic of secure systems and its application to trusted computing. In *Proceedings*

of the 30th IEEE Symposium on Security and Privacy (Washington, DC, USA, 2009), S&P'09, IEEE, IEEE Computer Society, pp. 221–236.

[35] de Clercq, R., Piessens, F., Schellekens, D., and Verbauwhede, I. Secure interrupts on low-end microcontrollers. In *25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2014)* (2014), ASAP'14, IEEE, pp. 1–6.

[36] Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3 (Mar. 1966), 143–155.

[37] Dolev, D., and Yao, A. C. On the security of public key protocols. In *IEEE Transactions on Information Theory* (Piscataway, NJ, USA, Sept. 1983), vol. 29, IEEE Press, pp. 198–208.

[38] Duflot, L., Perez, Y.-A., Valadon, G., and Levillain, O. Can you still trust your network card? Presented at CanSecWest'10, 2010. Presentation: http://www.ssi.gouv.fr/en/the-anssi/publications-109/press-releases/can-you-still-trust-your-network-card-185.html.

[39] Dunn, A. M., Hofmann, O. S., Waters, B., and Witchel, E. Cloaking malware with the trusted platform module. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association.

[40] El Defrawy, K., Aurélien Francillon, D., and Tsudik, G. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Proceedings of the Network & Distributed System Security Symposium* (Feb. 2012), NDSS'12.

[41] England, P., Lampson, B., Manferdelli, J., and Willman, B. A trusted open platform. *Computer 36*, 7 (July 2003), 55 – 62.

[42] Erlingsson, Ú. Low-level software security: Attacks and defenses. In *Foundations of Security Analysis and Design IV*, A. Aldini and R. Gorrieri, Eds., vol. 4677 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007, pp. 92–134.

[43] Erlingsson, U., Younan, Y., and Piessens, F. Low-level software security by example. In *Handbook of Information and Communication Security*, P. Stavroulakis and M. Stamp, Eds. Springer-Verlag, 2010, pp. 633–658.

[44] Etoh, H., and Yoda, K. Protecting from stack-smashing attacks. Tech. rep., IBM Research Divison, Tokyo Research Laboratory, 2000.

[45] Ferrucci, D. A. Introduction to "this is watson". *IBM Journal of Research and Development 56*, 3/4 (May/July 2012), 15.

[46] Fink, R. A., Sherman, A. T., Mitchell, A. O., and Challener, D. C. Catching the cuckoo: Verifying tpm proximity using a quote timing side-channel. In *Trust and Trustworthy Computing* (2011), J. M. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, Eds., vol. 6740 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 294–301.

[47] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In *Operating Systems Review* (New York, NY, USA, 2003), vol. 37 of *OSR'03*, ACM, pp. 193–206.

[48] Graham-Cumming, J. Searching for the prime suspect: How heartbleed leaked private keys. http://blog.cloudflare.com/searching-for-the-prime-suspect-how-heartbleed-leaked-private-keys, Apr. 2014. Accessed: 2014-11-05.

[49] Grawrock, D. *Dynamics of a Trusted Platform: A Building Block Approach*, 1st ed. Intel Press, Feb. 2009.

[50] Halderman, J., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., and Felten, E. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium* (New York, NY, USA, 2008), SSYM'08, ACM, pp. 45–60.

[51] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., and Del Cuvillo, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP'13, ACM, p. 11.

[52] Honeywell Information Systems Inc. *Multics - The Multics Virtual Memory.* Honeywell, 1972.

[53] Hoogstraten, H., Prins, R., Niggebrugge, D., Heppener, D., Groenewegen, F., Wettinck, J., Strooy, K., Arends, P., Pols, P., Kouprie, R., Moorrees, S., van Pelt, X., and Hu, Y. Z. Black Tulip - report of the investigation into the DigiNotar certificate authority breach. Tech. rep., FoxIT, 2012.

[54] Hunt, G. C., and Larus, J. R. Singularity: Rethinking the software stack. In *SIGOPS Operating Systems Review* (New York, NY, USA, Apr. 2007), vol. 41, ACM, pp. 37–49.

[55] IBM. Deep blue. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/, 1997. Accessed: 2014/11/05.

[56] Intel Corporation. *Intel 6 Series Chipset and Intel C200 Series Chipset*, 2011.

[57] Intel Corporation. *Software Guard Extensions Programming Reference*, 2013.

[58] ITF. Procedures for obtaining 2014 ITF approval of player analysis technology. http://www.itftennis.com/media/166067/166067.pdf, 2013. Accessed: 05/11/2014.

[59] ITF. Player analysis technology approval report. http://www.itftennis.com/media/166644/166644.pdf, 2014. Accessed: 05/11/2014.

[60] Jacobs, B., and Piessens, F. The verifast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008.

[61] Katzenbeisser, S., Koçabas, U., Leest, V., Sadeghi, A.-R., Schrijen, G.-J., Schröder, H., and Wachsmann, C. Recyclable PUFs: Logically reconfigurable PUFs. In *Cryptographic Hardware and Embedded Systems (CHES'11)* (2011), B. Preneel and T. Takagi, Eds., vol. 6917 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 374–389.

[62] Kauer, B. OSLO: improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium* (Berkeley, CA, USA, 2007), SSYM'07, USENIX Association, pp. 1–9.

[63] King, S., Chen, P., Wang, Y., Verbowski, C., Wang, H., and Lorch, J. SubVirt: Implementing malware with virtual machines. In *Symposium on Security and Privacy* (2006), S&P'06, IEEE Computer Society.

[64] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP'09, ACM, pp. 207–220.

[65] Koeberl, P., Schulz, S., Sadeghi, A.-R., and Varadharajan, V. Trustlite: a security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys'14, ACM, p. 10.

[66] Kotla, R., Rodeheffer, T., Roy, I., Studei, P., and Wester, B. Pasture: secure offline data access using commodity trusted hardware. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association, pp. 321–334.

[67] Kursawe, K., Sadeghi, A.-R., Schellekens, D., Skoric, B., and Tuyls, P. Reconfigurable physical unclonable functions-enabling technology for tamper-resistant storage. In *Proceedings of the International Workshop on Hardware-Oriented Security and Trust* (2009), HOST'09, IEEE, pp. 22–29.

[68] Levin, D., Douceur, J. R., Lorch, J. R., and Moscibroda, T. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2009), vol. 9 of *NSDI'09*, USENIX Association, pp. 1–14.

[69] Li, J., Krohn, M. N., Mazières, D., and Shasha, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2004), vol. 6 of *OSDI'04*, USENIX Association.

[70] Lie, D., Chandramohan, T., Mark, M., Patrick, L., Dan, B., John, M., and Mark, H. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), vol. 35 of *ASPLOS'00*, ACM, pp. 168–177.

[71] Liedtke, J. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1993), SOSP'93, ACM, pp. 175–188.

[72] Liedtke, J. Toward Real Microkernels. *Communications of the ACM 39*, 9 (1996), 77.

[73] Longley, D., and Rigby, S. An automatic search for security flaws in key management schemes. *Computers & Security 11*, 1 (1992), 75–89.

[74] Martignoni, L., Paleari, R., and Bruschi, D. Conqueror: tamper-proof code execution on legacy systems. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'10)* (July 2010), Lecture Notes in Computer Science, Springer-Verlag, pp. 21–40. Bonn, Germany.

[75] Martignoni, L., Poosankam, P., Zaharia, M., Han, J., McCamant, S., Song, D., Paxson, V., Perrig, A., Shenker, S.,

and Stoica, I. Cloud terminal: Secure access to sensitive applications from untrusted systems. In *Procceedings of the 2012 USENIX Annual Technical Conference* (2012), ATC '12.

[76] Martin, A. The ten page introduction to trusted computing. Tech. Rep. RR-08-11, Oxford University, Computing Laboratory, Dec. 2008.

[77] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (Washington, DC, USA, May 2010), S&P'10, IEEE Computer Society, pp. 143–158.

[78] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems* (New York, NY, USA, Apr. 2008), EuroSys'08, ACM, pp. 315–328.

[79] McCune, J. M., Perrig, A., and Reiter, M. K. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, June 2006), ATEC '06, USENIX Association.

[80] McCune, J. M., Perrig, A., and Reiter, M. K. Safe passage for passwords and other sensitive data. In *Proceedings of the Symposium on Network and Distributed Systems Security* (Feb. 2009), NDSS'09, Internet Society.

[81] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP'13, ACM, p. 8.

[82] Miller, M., Yee, K.-P., and Shapiro, J. S. Capability myths demolished. Tech. Rep. SRL2003-02, Johns Hopkins University, 2003.

[83] Mitre. Cve-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, Dec. 2013. Accessed: 05/11/2014.

[84] Motorala. *Programmer's Reference Manual*, 1992.

[85] Nikiforakis, N., Piessens, F., and Joosen, W. HeapSentry: Kernel-assisted protection against heap overflows. In *Proceedings of the 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'13)* (July 2013), K. Rieck, P. Stewin, and J.-P.

Seifert, Eds., vol. 7967 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 177–196.

[86] NIST. National vulnerability database. https://nvd.nist.gov. Accessed: 2014-11-05.

[87] Noorman, J., Agten, P., Daniels, W., Strackx, R., Herrewege, A. V., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium* (Aug. 2013), SSYM'13, USENIX Association.

[88] Omer Levy, Arvind Kumar, P. G. Advanced Security Features of Intel vPro Technology. *Intel Technology Journal 12*, 4 (Dec. 2008), 229–238.

[89] Owusu, E., Guajardo, J., McCune, J., Newsome, J., Perrig, A., and Vasudevan, A. OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (New York, NY, USA, 2013), CCS'13, ACM, pp. 13–24.

[90] Page, D. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, University of Bristol, April 2002.

[91] Pappu, R., Recht, B., Taylor, J., and Gershenfeld, N. Physical one-way functions. *Science 297*, 5589 (2002), 2026–2030.

[92] Parno, B. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd Conference on Hot Topics in Security* (Berkeley, CA, USA, 2008), HOTSEC'08, USENIX Association, pp. 1–6.

[93] Parno, B., Lorch, J. R., Douceur, J. R., Mickens, J., and McCune, J. M. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy* (Washington, DC, USA, May 2011), S&P'11, IEEE Computer Society, pp. 379–394.

[94] Parno, B., Mccune, J. M., and Perrig, A. Bootstrapping trust in commodity computers. In *Proceedings of the IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), S&P'10, IEEE Computer Society, pp. 414–429.

[95] Parno, B., McCune, J. M., and Perrig, A. *Bootstrapping Trust in Modern Computers*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[96] Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., and Piessens, F. Secure compilation to protected module architectures. In *Accepted for publication in Transactions on Programming Languages and Systems (TOPLAS)* (New York, NY, USA), ACM.

[97] Patrignani, M., and Clarke, D. Fully abstract trace semantics of low-level isolation mechanisms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (Mar. 2014), SAC'14, ACM, pp. 1562–1569.

[98] Patrignani, M., Clarke, D., and Piessens, F. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)* (2013), C.-c. Shan, Ed., vol. 8301 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 176–191.

[99] Provos, N. Improving host security with system call policies. In *Proceedings of the 12th Conference on USENIX Security Symposium* (Berkeley, CA, USA, 2003), SSYM'03, USENIX Association.

[100] Provos, N., Friedl, M., and Honeyman, P. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium* (Berkeley, CA, USA, 2003), SSYM'03, USENIX Association.

[101] Reynolds, J. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference* (New York, NY, USA, 1972), ACM, pp. 717–740.

[102] Rosenberg, J., and Abramson, D. A. MONADS-PC: A capability based workstation to support software engineering. In *Proceedings of the 18th Hawaii International Conference on System Sciences* (1985), pp. 222–231.

[103] Rutkowska, J. Subverting VistaTM Kernel For Fun And Profit. In *Blackhat Briefings* (2006).

[104] Sadeghi, A.-R., Selhorst, M., Stüble, C., Wachsmann, C., and Winandy, M. TCG inside?: a note on TPM specification compliance. In *Proceedings of the first ACM workshop on Scalable trusted computing* (New York, NY, USA, 2006), STC'06, ACM, pp. 47–56.

[105] Sahita, R., Warrier, U., and Dewan, P. Protecting Critical Applications on Mobile Platforms. *Intel Technology Journal 13*, 2 (June 2009), 16–35.

[106] Saltzer, J., and Schroeder, M. The protection of information in computer systems. In *Proceedings of the IEEE* (1975), vol. 63, IEEE, pp. 1278–1308.

[107] Schellekens, D., Tuyls, P., and Preneel, B. Embedded trusted computing with authenticated non-volatile memory. In *First International Conference on Trusted Computing and Trust in Information Technologies (TRUST'08)* (2008), P. Lipp, A.-R. Sadeghi, and K.-M. Koch, Eds., Lecture Notes in Computer Science, Springer-Verlag, pp. 60–74.

[108] Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the 19th USENIX Security Symposium* (2010), SEC'10.

[109] Seshadri, A., Luk, M., Qu, N., and Perrig, A. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), SOSP'07, ACM, pp. 335–350.

[110] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles* (New York, NY, USA, Oct. 2005), SOSP'05, ACM, pp. 1–15.

[111] Shacham, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.

[112] Singaravelu, L., Pu, C., Härtig, H., and Helmuth, C. Reducing TCB complexity for security-sensitive applications: three case studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (New York, NY, USA, 2006), EuroSys'06, ACM, pp. 161–174.

[113] Spafford, E. H. The internet worm program: An analysis. *SIGCOMM Computer Communication Review 19*, 1 (Jan. 1988), 17–57.

[114] Sparks, E. R. A security assessment of trusted platform modules. Tech. Rep. TR2007-597, Dartmouth College, Department of Computer Science, June 2007.

[115] Strackx, R., Agten, P., Avonds, N., and Piessens, F. Salus: Kernel support for secure process compartments. In *Accepted for publication in Endorsed Transactions on Security and Safety*, EAI.

[116] Strackx, R., Jacobs, B., and Piessens, F. ICE: A passive, high-speed, state-continuity scheme. In *Accepted for publication in Annual Computer Security Applications Conference* (2014), ACSAC'14.

[117] Strackx, R., Jacobs, B., and Piessens, F. ICE: A passive, high-speed, state-continuity scheme (extended version). CW Reports CW672, Department of Computer Science, KU Leuven, September 2014.

[118] Strackx, R., Philippaerts, P., and Vogels, F. Idea: Towards an Inverted Cloud. In *Accepted in Engineering Secure Software and Systems (ESSoS'15)* (Mar. 2015), Lecture Notes in Computer Science, Springer Berlin Heidelberg.

[119] Strackx, R., and Piessens, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM conference on Computer and Communications Security* (New York, NY, USA, October 2012), CCS'12, ACM, pp. 2–13.

[120] Strackx, R., Piessens, F., and Preneel, B. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *Security and Privacy in Communication Networks (SecureComm'10)* (2010), S. Jajodia and J. Zhou, Eds., vol. 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer Berlin Heidelberg, pp. 344–361.

[121] Strackx, R., Younan, Y., Philippaerts, P., and Piessens, F. Efficient and effective buffer overflow protection on ARM processors. In *Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices – Proceedings of the 4th IFIP WG 11.2 International Workshop (WISTP'10)* (Apr. 2010), P. Samarati, M. Tunstall, J. Posegga, K. Markantonakis, and D. Sauveron, Eds., vol. 6033 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 1–16.

[122] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security* (New York, NY, USA, 2009), EuroSec'09, ACM, pp. 1–8.

[123] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing* (New York, NY, USA, 2003), ICS'03, ACM, pp. 160–171.

[124] Swift, M. M., Martin, S., Levy, H. M., and Eggers, S. J. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (New York, NY, USA, 2002), EW'10, ACM, pp. 102–107.

[125] Ta-Min, R., Litty, L., and Lie, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI'06, USENIX Association, pp. 279–292.

[126] Tarnovsky, C. Deconstructing a "secure" processor. In *Blackhat* (2010).

[127] Technology, M. Hybrid memory - bridging the gap between DRAM speed and NAND nonvolatility. http://www.micron.com/products/dram-modules/nvdimm. Accessed: 06/11/2014.

[128] Technology, V. NV-DIMM: Achieving greater ROI from SSDs. http://www.vikingtechnology.com/uploads/NV_DIMM_ROI.pdf. Accessed: 06/11/2014.

[129] Tian, D., Zeng, Q., Wu, D., Liu, P., and Hu, C. Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring. In *Proceedings of the Network & Distributed System Security Symposium* (Feb. 2012), NDSS'12, Internet Society.

[130] Tiobe Software. Tiobe software: Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed: 2014-11-06.

[131] Torvalds, L. Hybrid kernel, not NT. forum comment http://www.realworldtech.com/forum/?threadid=65915&curpostid=65936, May 2006. Accessed: 2014-11-06.

[132] Trusted Computing Group. *Design Principles Specification Version 1.2*, 2011.

[133] Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., and Miyauchi, H. Cryptanalysis of des implemented on computers with cache. In *Proceedings of the 5th international conference on cryptographic hardware and embedded systems (CHES'03)* (2003), C. D. Walter, c. K. Koç, and C. Paar, Eds., vol. 2779 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 62–76.

[134] Vasudevan, A., Chaki, S., Jia, L., McCune, J., Newsome, J., and Datta, A. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), S&P'13, IEEE Computer Society, pp. 430–444.

[135] Viega, J., and Thompson, H. The state of embedded-device security (spoiler alert: It's bad). *IEEE Security and Privacy 10*, 5 (Sept. 2012), 68–70.

[136] Vilanova, L., Ben-Yehuda, M., Navarro, N., Etsion, Y., and Valero, M. CODOMs: Protecting software with code-centric memory domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, June 2014), ISCA '14, IEEE Press, pp. 469–480.

[137] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1993), SOSP '93, ACM, pp. 203–216.

[138] Watson, R. N., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security symposium* (Berkeley, CA, USA, 2010), SSYM'10, USENIX Association.

[139] Williams, P., and Boivie, R. CPU support for secure executables. In *Trust and Trustworthy Computing (TRUST'11)* (2011), J. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, Eds., vol. 6740 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 172–187.

[140] Wilson, P., Frey, A., Mihm, T., Kershaw, D., and Alves, T. Implementing embedded security on dual-virtual-cpu systems. *IEEE Design & Test of Computers 24*, 6 (Nov. 2007), 582–591.

[141] Winter, J., and Dietrich, K. A hijacker's guide to the LPC bus. In *Public Key Infrastructures, Services and Applications (EuroPKI'11)* (2012), S. Petkova-Nikova, A. Pashalidis, and G. Pernul, Eds., vol. 7163 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 176–193.

[142] Witchel, E., and Asanović, K. Hardware works, software doesn't: Enforcing modularity with mondriaan memory protection. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2003), vol. 9 of *HOTOS'03*, USENIX Association.

[143] Witchel, E., Cates, J., and Asanović, K. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ASPLOS'02, ACM, pp. 304–316.

[144] Woodruff, J., Watson, R. N. M., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture* (2014), ISCA'14.

[145] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30 IEEE Symposium on Security and Privacy* (2009), S&P'09, IEEE, pp. 79–93.

[146] Younan, Y., Joosen, W., and Piessens, F. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Tech. Rep. CW386, Department of Computer Science, KULeuven, 2004.

[147] Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., and Joosen, W. Paricheck: an efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2010), ASIACCS '10, ACM, pp. 145–156.

[148] Younan, Y., Pozza, D., Piessens, F., and Joosen, W. Extended protection against stack smashing attacks without performance loss. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), ACSAC'06, IEEE Computer Society, pp. 429–438.

[149] Zeng, Q., Wu, D., and Liu, P. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI'11, ACM, pp. 367–377.

[150] Zhou, Z., Gligor, V. D., Newsome, J., and McCune, J. M. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2012), S&P'12, pp. 616–630.

[151] ZHOU, Z., YU, M., AND GLIGOR, V. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (May 2014), S&P'14.

List of Publications

Articles in International Reviewed Journals

- Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., and Piessens, F. Secure compilation to protected module architectures. In *Accepted for publication in Transactions on Programming Languages and Systems (TOPLAS)* (New York, NY, USA), ACM.
- Strackx, R., Agten, P., Avonds, N., and Piessens, F. Salus: Kernel support for secure process compartments. In *Accepted for publication in Endorsed Transactions on Security and Safety*, EAI.

Papers at International Conferences and Symposia, Published in Full in Proceedings

- Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security* (New York, NY, USA, 2009), EuroSec'09, ACM, pp. 1–8.
- Strackx, R., Younan, Y., Philippaerts, P., and Piessens, F. Efficient and effective buffer overflow protection on ARM processors. In *Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices – Proceedings of the 4th IFIP WG 11.2 International Workshop (WISTP'10)* (Apr. 2010), vol. 6033 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 1–16.
- Strackx, R., Piessens, F., and Preneel, B. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *Security and Privacy in Communication Networks (SecureComm'10)* (2010), vol. 50 of *Lecture*

Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer Berlin Heidelberg, pp. 344–361.

- Agten, P., Nikiforakis, N., Strackx, R., Groef, W. D., and Piessens, F. Recent developments in low-level software security. In *Information Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems (WISTP'12)* (2012), vol. 7322 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 1–16. WISTP.

- Agten, P., Strackx, R., Jacobs, B., and Piessens, F. Secure compilation to modern processors. In *Proceedings of the 25th Computer Security Foundations Symposium* (Los Alamitos, CA, USA, 2012), CSF'12, IEEE Computer Society, pp. 171–185.

- Gadaleta, F., Strackx, R., Nikiforakis, N., Piessens, F., and Joosen, W. On the effectiveness of virtualization-based security. In *Current Issues in IT Security* (2012), Max Planck Institute. Security'12.

- Strackx, R., and Piessens, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM conference on Computer and Communications Security* (New York, NY, USA, October 2012), CCS'12, ACM, pp. 2–13.

- Swinnen, A., Strackx, R., Philippaerts, P., and Piessens, F. Protoleaks: A reliable and protocol-independent network covert channel. In *Proceedings of the International Conference on Information System Security* (December 2012), ICISS'12, Springer Verlag, pp. 119–133.

- Strackx, R., Noorman, J., Verbauwhede, I., Preneel, B., and Piessens, F. Protected software module architectures. In *Securing Electronic Business Processes*, H. Reimer, N. Pohlmann, and W. Schneider, Eds., ISSE'13. Springer-Verlag, 2013, pp. 241–251.

- Noorman, J., Agten, P., Daniels, W., Strackx, R., Herrewege, A. V., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium* (Aug. 2013), SSYM'13, USENIX Association.

- Avonds, N., Strackx, R., Agten, P., and Piessens, F. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *Security and Privacy in Communication Networks (SecureComm'13)* (Sept. 2013), vol. 127 of *Lecture Notes of the Institute* *for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer International Publishing, pp. 252–269.

- Strackx, R., Jacobs, B., and Piessens, F. ICE: A passive, high-speed, state-continuity scheme. In *Annual Computer Security Applications Conference* (2014), ACSAC'14.
- Strackx, R., Philippaerts, P., and Vogels, F. Idea: Towards an Inverted Cloud. In *Accepted in Engineering Secure Software and Systems (ESSoS'15)* (Mar. 2015), Lecture Notes in Computer Science, Springer Berlin Heidelberg.
- Strackx, R., and Lambrigts, N. Idea: State-Continuous Transfer of State in Protected-Module Architectures. In *Accepted in Engineering Secure Software and Systems (ESSoS'15)* (Mar. 2015), Lecture Notes in Computer Science, Springer Berlin Heidelberg.

Technical Reports

- Agten, P., Strackx, R., Jacobs, B., and Piessens, F. Secure compilation to modern processors: Extended version. CW Reports CW619, Department of Computer Science, KU Leuven, April 2012.
- Strackx, R., Jacobs, B., and Piessens, F. ICE: A passive, high-speed, state-continuity scheme (extended version). CW Reports CW672, Department of Computer Science, KU Leuven, September 2014.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMINDS - DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
raoul.strackx@cs.kuleuven.be
http://www.cs.kuleuven.be/