Trusted Systems in Untrusted Environments: Protecting against Strong Attackers

Vertrauenswürdige Systeme in nicht vertrauenswürdigen Umgebungen:
Schutz vor starken Angreifern

Der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Doktorgrades Dr.-Ing.
vorgelegt von

Johannes Götzfried

aus Amberg

Als Dissertation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Tag der mündlichen Prüfung:	7. Dezember 2017
Vorsitzender des Promotionsorgans:	Prof. Dr.-Ing. Reinhard Lerch
Gutachter:	Prof. Dr.-Ing. Felix Freiling
	Prof. Dr. Ir. Ingrid Verbauwhede

Abstract

In this thesis, we investigate possibilities of operating trusted systems within untrusted environments in the presence of strong attackers. We consider attackers with physical access to a system as well as root-level attackers which are able to execute code on the target system at the highest privilege level possible. Our proposed defense mechanisms range from hardware-based trusted computing architectures over hardware-assisted software solutions to pure software-based encryption schemes.

First, we design two hardware-based trusted computing architectures for embedded devices, called Soteria and Atlas, which both protect against root-level attackers and focus on code and data confidentiality. At its heart, Soteria is a lightweight program-counter based memory access control extension which provides integrity, authenticity, and confidentiality guarantees for software modules. To guarantee the confidentiality of code at any given point in time against all software attacks, we design a scheme consisting of initially encrypted software modules and loader modules. Mutual integrity checks between software modules and the loader ensure that a software module is only decrypted if the system behaves with integrity. Atlas protects software modules with the help of an encryption unit placed between the cache and main memory. Because Atlas alone only guarantees confidentiality, but no integrity, it is usually combined with traditional memory protection mechanisms managed by the operating system. In case of a compromised operating system, however, Atlas still maintains confidentiality for protected modules.

We then exploit existing hardware-based trusted computing architectures of general purpose devices by presenting two hardware-assisted software solutions built on top of them. However, we also demonstrate how secret information can be inferred from such an architecture if the attacker assumptions are violated. We leverage the Trusted Platform Module (TPM) to support the boot process for consumer devices in order to provide secure full disk encryption. In detail, we implement an authentication protocol which defends against traditional evil maid attacks with repeated physical access. Trust is bootstrapped from an external active USB drive which verifies the boot process by utilizing sealed nonces. The underlying principle is that the user and the computer are mutually authenticating each other with the help of this active USB drive. While with the TPM the whole software stack is part of the Trusted Computing Base (TCB), Intel's Software Guard Extensions (SGX) allow to dynamically establish roots of trust on general purpose hardware. Although not being designed to work in kernel mode, we found a way to deploy SGX to shield kernel components from each other and user-mode applications, even in the event of a full system compromise. However, we also show that SGX is generally vulnerable against cache attacks by practically demonstrating an access driven cache attack on AES when running inside an SGX enclave. In fact, the attack surface for side-channels increases dramatically in the scenario of SGX due to the power of root-level attackers, for example, by exploiting the accuracy of Intel's Performance Monitoring Counters (PMC) which are usually restricted to kernel code.

Finally, we demonstrate that certain security guarantees can also be given when trust is not rooted in hardware by describing two software-based memory encryption solutions to mitigate memory disclosure attacks. RamCrypt targets the problem by providing mechanisms to transparently encrypt whole process address spaces and has been developed as an operating system kernel patch. It can be deployed and enabled on a per-process basis without recompiling user-mode applications. In every enabled process, data is only stored in cleartext for the moment it is processed, and otherwise stays encrypted in RAM. Because encrypting process address spaces is operating system specific, we also present HyperCrypt, a hypervisor-based solution which encrypts the entire kernel and user space while being transparent for the guest operating system and all applications running on top of it.

Zusammenfassung

In der vorliegenden Arbeit untersuchen wir Möglichkeiten vertrauenswürdige Systeme innerhalb nicht vertrauenswürdiger Umgebungen in der Gegenwart starker Angreifer zu betreiben. Wir betrachten sowohl Angreifer mit physischem Zugriff auf ein System als auch Root-Level-Angreifer, die auf dem Zielsystem Code mit maximalen Berechtigungen ausführen können. Unsere Verteidigungsmechanismen reichen von hardwarebasierten Trusted-Computing-Architekturen über hardwareunterstützte Softwarelösungen bis zu rein softwarebasierten Verschlüsselungsmaßnahmen.

Zuerst entwerfen wir zwei hardwarebasierte Trusted-Computing-Architekturen für eingebettete Systeme, Soteria und Atlas genannt. Beide schützen gegen Root-Level-Angreifer, wobei der Fokus auf der Vertraulichkeit von Code und Daten liegt. Soteria ist im Herzen eine leichtgewichtige programmzählerbasierte Speicherzugriffskontrollerweiterung, die Integritäts-, Authentizitäts- und Vertraulichkeitsgarantien für Softwaremodule anbietet. Um die Vertraulichkeit von Code zu jeder Zeit gegen alle Software-Angriffe zu gewährleisten, konzipieren wir ein Schema bestehend aus anfangs verschlüsselten Softwaremodulen und Lademodulen. Gegenseitige Integritätsprüfungen zwischen Softwaremodulen und dem Lader sorgen dafür, dass ein Softwaremodul nur entschlüsselt wird, wenn das Gesamtsystem integer ist. Atlas schützt Softwaremodule mit Hilfe einer Verschlüsselungseinheit zwischen Cache und Hauptspeicher. Weil Atlas selbst nur Vertraulichkeit jedoch keine Integrität garantiert, wird es in der Regel mit traditionellen vom Betriebssystem verwalteten Speicherschutzmechanismen kombiniert. Im Falle eines kompromittierten Betriebssystems erhält Atlas jedoch weiterhin die Vertraulichkeit für geschützte Module aufrecht.

Als Nächstes nutzen wir bestehende hardwarebasierte Trusted-Computing-Architekturen von Endbenutzergeräten aus. Dazu präsentieren wir zwei hardwareunterstützte Softwarelösungen. Wir zeigen allerdings auch, wie geheime Informationen einer solchen Architektur gewonnen werden können, wenn die Angreiferannahmen verletzt sind. Wir nutzen das Trusted Platform Module (TPM) um den Bootvorgang für Verbrauchergeräte zu unterstützen und eine sichere Festplattenvollverschlüsselung bereitzustellen. Genauer gesagt implementieren wir ein Authentifizierungsprotokoll, das gegen traditionelle Evil-Maid-Angriffe mit wiederholtem physischem Zugriff schützt. Vertrauen wird dabei ausgehend von einem aktiven USB-Gerät aufgebaut, das den Bootvorgang unter der Verwendung von versiegelten Einmalnummern überprüft. Das zugrundeliegende Prinzip ist, dass sich der Benutzer und der Computer gegenseitig mit Hilfe dieses aktiven USB-Geräts authentifizieren. Während bei der Benutzung des TPM der gesamte Software-Stack Teil der Trusted Computing Base (TCB) ist, ermöglichen Intel's Software Guard Extensions (SGX) dynamisch Vertrauensanker auf handelsüblicher Hardware zu errichten. Obwohl SGX nicht zur Verwendung im Kernelmodus ausgelegt wurde, haben wir einen Weg gefunden SGX einzusetzen um Kernelkomponenten untereinander und gegen Usermode-Anwendungen auch im Falle einer vollständigen Systemkompromittierung abzuschotten. Wir zeigen allerdings auch, dass SGX im Allgemeinen anfällig gegen Cache-Angriffe ist, indem wir praktisch einen zugriffsgesteuerten Cache-Angriff gegen AES demonstrieren, während es innerhalb einer SGX-Enklave läuft. Tatsächlich vergrößert sich die Angriffsoberfläche für Seitenkanäle im Falle von SGX aufgrund der Mächtigkeit von Root-Level-Angreifern drastisch, zum Beispiel durch die Ausnutzung der Genauigkeit von Intel's Performance Monitoring Counters (PMC), deren Verfügbarkeit üblicherweise auf Kernelcode beschränkt ist.

Zuletzt demonstrieren wir, dass bestimmte Sicherheitsgarantien auch gegeben werden können, wenn das Vertrauen nicht in Hardware verankert ist, indem wir zwei softwarebasierte Speicherverschlüsselungslösungen vorstellen, die Angriffe auf den Hauptspeicher mildern. RamCrypt adressiert das Problem, indem es Mechanismen zur transparenten Verschlüsselung gesamter Prozessadressräume bereitstellt und wurde als Betriebssystem-Kernel-Patch entwickelt. Es kann pro Prozess ohne die Notwendigkeit Usermode-Applikationen neu zu übersetzen eingesetzt und aktiviert werden. Für jeden aktivierten Prozess werden Daten nur so lange im Klartext gespeichert, wie sie verarbeitet werden, und verbleiben ansonsten verschlüsselt im RAM. Weil das Verschlüsseln von Prozessadressräumen betriebssystemspezifisch ist, präsentieren wir zusätzlich HyperCrypt, eine Hypervisor-basierte Lösung, die Kernel- und Userspace vollständig verschlüsselt, wobei sie transparent gegenüber dem Gastbetriebssystem und allen darauf laufenden Applikationen ist.

Acknowledgments

I would like to thank my doctoral advisor Felix Freiling for offering me to work with him at his Chair for IT-Security Infrastructures at the Department of Computer Science at the FAU Erlangen-Nuremberg, his time for various project meetings, and his continuous support. Without him, I probably would not have started as a PhD candidate. Furthermore, I would like to thank my advisor and group leader Tilo Müller for pushing me towards becoming a PhD candidate while I still was a master student, discussing plenty of more or less sophisticated research ideas, and teaching me the mysterious art of publishing academic papers. I also want to thank all my colleagues at the Chair for a friendly and casual working atmosphere including Nerf-Guns, coffee sessions, and the one or the other beer.

In addition, I want to thank my Belgian colleagues, namely Ingrid Verbauwhede for the fruitful collaboration and her agreement to become the second reviewer of this thesis, and Ruan de Clerq for his support regarding the SOFIA toolchain and a climbing trip to Fontainebleau. However, especially I would like to thank Pieter Maene for his great hospitality every time I was travelling to Leuven, for lending me his bike, and for dealing with all issues which were too close to the hardware.

Last but not least, I want to thank my parents for their encouragement and incessant support of my dissertation project as well as my entire studies.

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

Contents

1 Introduction 1
- 1.1 Contributions 3
- 1.2 Related Work 5
- 1.3 Publications 7
- 1.4 Outline 10

2 Background 11
- 2.1 Trusted Computing 11
  - 2.1.1 Security Properties 12
  - 2.1.2 Architectural Features 14
- 2.2 Hardware-based Trusted Computing Architectures 15
  - 2.2.1 Sancus 15
  - 2.2.2 Trusted Platform Module 16
  - 2.2.3 Software Guard Extensions 18
- 2.3 Full Disk Encryption 20
  - 2.3.1 Cold Boot and Evil Maid Attacks 20
  - 2.3.2 Bootkit Attacks 21
- 2.4 CPU-bound Encryption 23

3 Designing Trusted Computing Architectures 25
- 3.1 Offline Software Protection within Low-cost Embedded Devices 26
  - 3.1.1 Attacker Model 27
  - 3.1.2 Soteria Architecture 27
  - 3.1.3 System Deployment 28
  - 3.1.4 Access Control 29
  - 3.1.5 Security Properties 30
  - 3.1.6 Hardware Implementation 31
  - 3.1.7 Software Implementation 32
  - 3.1.8 Toolchain Implementation 33
  - 3.1.9 Performance, Area, and Power Analysis 33
  - 3.1.10 Discussion 35
- 3.2 Application Confidentiality in Compromised Embedded Systems 36
  - 3.2.1 Attacker Model 36
  - 3.2.2 System Model 37
  - 3.2.3 Atlas Architecture 37
  - 3.2.4 Hardware Implementation 38
  - 3.2.5 Software Implementation 40
  - 3.2.6 Performance and Area Analysis 41
  - 3.2.7 Discussion 42
- 3.3 Summary 43

4 Exploiting Existing Trusted Computing Architectures **45**
4.1 Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption . 46
4.1.1 The Stark Protocol 47
4.1.2 Security Argument 48
4.1.3 Design Components of Stark 49
4.1.4 Mark: A Practical Implementation of Stark 50
4.1.5 Hardware Implementation 52
4.1.6 Software Implementation 53
4.1.7 Formal Security Analysis 56
4.1.8 Discussion 59
4.2 Isolating Operating System Components with Intel SGX 61
4.2.1 Architecture 62
4.2.2 Prototype Implementation 62
4.2.3 Correctness, Performance, and Security Analysis 64
4.2.4 Discussion 66
4.3 Cache Attacks on Intel SGX 67
4.3.1 Attacker Model 67
4.3.2 Gladman Implementation of AES 68
4.3.3 Neve and Seifert's Elimination Method 69
4.3.4 Cache Architecture 70
4.3.5 Cache Priming and Probing 71
4.3.6 Attack Setup 73
4.3.7 Performance Analysis 75
4.3.8 Practicability Analysis 77
4.3.9 Discussion 79
4.4 Summary 80

5 Software-based Memory Encryption **81**
5.1 Kernel-based Address Space Encryption for User-mode Processes 82
5.1.1 Linux Virtual Memory Management 83
5.1.2 RamCrypt Architecture 84
5.1.3 RamCrypt Implementation 86
5.1.4 Runtime Performance 88
5.1.5 Binary Compatibility and Practical Security Analysis 91
5.1.6 Discussion 92
5.2 Hypervisor-based Encryption of Kernel and User Space 94
5.2.1 BitVisor Memory Management 94
5.2.2 HyperCrypt Architecture 95
5.2.3 HyperCrypt Implementation 97
5.2.4 Runtime Performance 99
5.2.5 Practical Security Analysis 100
5.2.6 Discussion 102
5.3 Summary 103

6 Conclusion **105**

Bibliography **107**

Acronyms

AES Advanced Encryption Standard
AESNI AES New Instructions
AEX Asynchronous Enclave Exit
AIK Attestation Identity Key
ALU Arithmetic Logic Unit
API Application Programming Interface
ASCI Anti Side-Channel Interference
ASCII American Standard Code for Information Interchange
BIOS Basic Input/Output System
BSS Block Started by Symbol
CAN Controller Area Network
CBC Cipher Block Chaining
CCM Counter with CBC-MAC
COW Copy-on-Write
CPU Central Processing Unit
CVE Common Vulnerabilities and Exposures
DEK Data Encryption Key
DMA Direct Memory Access
DoS Denial-of-Service
DRAM Dynamic RAM
DRM Digital Rights Management
DRoT Dynamic RoT
EEPROM Electrically Erasable Programmable ROM
EK Endorsement Key
ELF Executable and Linkable Format
EPC Enclave Page Cache
EPCM Enclave Page Cache Map
EPID Enhanced Privacy Identifier
EPT Extended Page Tables
FDE Full Disk Encryption
FIFO First-In-First-Out
FPGA Field Programmable Gate Array

GCC GNU Compiler Collection

GCM Galois/Counter Mode

GDB GNU Debugger

GPIO General-Purpose Input/Output

GPL General Public License

GRUB GRand Unified Bootloader

GUI Graphical User Interface

HDD Hard Disk Drive

HMAC Hash-based MAC

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

ID Identifier

IP Infrastructure Provider

ISA Instruction Set Architecture

IV Initialization Vector

KEK Key Encryption Key

KVM Kernel-based Virtual Machine

LED Light-Emitting Diode

LiME Linux Memory Extractor

LKM Loadable Kernel Module

LLVM Low Level Virtual Machine

LoC Lines of Code

LPC Low Pin Count

LRU Least Recently Used

LUT Look-Up Table

MAC Message Authentication Code

MBR Master Boot Record

MEE Memory Encryption Engine

MitM Man-in-the-Middle

MMU Memory Management Unit

MPU Memory Protection Unit

MSR Model-Specific Register

NIST National Institute of Standards and Technology

OCB Offset Codebook Mode

OS Operating System

PBKDF2 Password-based Key Derivation Function 2

PCI Peripheral Component Interconnect

PCR Platform Configuration Register

PID Process Identifier

PIN Personal Identification Number

PMA Protected Module Architecture

PMC Performance Monitoring Counters

PMD Page Middle Directory

PMU Performance Monitoring Units

PRM Processor Reserved Memory

PTE Page Table Entry

RAM Random-Access Memory

RNG Random Number Generator

ROM Read-Only Memory

RoT Root of Trust

SDK Software Development Kit

SDM Software Development Manual

SED Self-Encrypting Drive

SEV Secure Encrypted Virtualization

SGX Software Guard Extensions

SIMD Single Instruction Multiple Data

SLAT Second Layer Address Translation

SME Secure Memory Encryption

SM Software Module

SP Software Provider

SPT Shadow Page Tables

SSE Streaming SIMD Extensions

SSL Secure Sockets Layer

TCB Trusted Computing Base

TCG Trusted Computing Group

TLB Translation Lookaside Buffer

TLS Transport Layer Security

TOCTOU Time-of-Check Time-of-Use

TPM Trusted Platform Module

TXT Trusted Execution Technology

UART Universal Asynchronous Receiver-Transmitter

UEFI Unified Extensible Firmware Interface

USB Universal Serial Bus

VMA Virtual Memory Area

VMM Virtual Machine Monitor

VM Virtual Machine

VRAM Video RAM

XEX XOR-Encrypt-XOR

XTS XEX-based Tweaked-Codebook Mode with Ciphertext Stealing

1 Introduction

In 2015, Miller and Valasek managed to remotely access the Controller Area Network (CAN) bus of a Fiat Chrysler Jeep Cherokee by connecting their computers to the exact same wireless carrier the car was talking to [117]. Because the software of the on-board entertainment system contained a bug and was connected to the CAN bus the brake and engine controllers were connected to as well, they managed to remotely control air conditioning, door locking, windscreen wipers, brakes, and even the car's acceleration without the driver's consent. This incident is a shining example of how ubiquitous computing playing an increasing role in our daily life also increases our attack surface.

Although the manufacturer closed the vulnerability, it is generally difficult to give guarantees about whether a large software system like an on-board entertainment system is bug-free. Consequently, critical pieces of software like controllers for engines and brakes must be protected under the assumption of an untrusted environment. For example, potentially privileged software components should be assumed to be in a compromised state.

When speaking about untrusted environments, we want to consider environments which are controlled by *strong* attackers. We classify such attackers into two main categories, namely *physical attackers* and *root-level attackers*.

A physical attacker is an attacker who has physical access to a computing system either while it is running or after it has been turned off. His goal is to obtain sensitive information associated with the computing system, manipulate information processed, or both. Simple attacks would be, for example, obtaining sensitive information stored on a permanent storage device, dumping the main memory, or tapping system busses. Targets of physical attacks could be unattended devices such as notebooks on a business trip or servers within a data center.

Root-level attackers are assumed to control the whole software stack of a certain device apart from well-defined trusted modules. This includes high-privileged software such as an operating system or even a hypervisor. The attackers' goal is to extract information from these well-defined trusted modules or to influence the results of computations done within those modules. Usually trusted modules are very small compared to the overall software stack and the software running within them is checked carefully. For those modules certain security guarantees such as confidentiality and integrity can be given despite the untrusted software environment they run in. Trust is bootstrapped from some inherently trusted parts, also called Trusted Computing Base (TCB), which are typically placed in hardware.

There are plenty of examples where the party operating the trusted modules cannot rely on the environment to behave in an expected way, but needs the modules to behave with integrity. This is mostly the case when the party operating the environment is different from the party operating the trusted modules even though everything runs on the same device. An obvious application would be Digital Rights Management (DRM) where a content provider wants to restrict the ways its content is used, but the hardware where the content is displayed is under control of a non-trusted end-user. The content provider could provide a trusted module which only decrypts the content if the module behaves with integrity, i.e., it assures to the content provider that it has not been tampered with. In this scenario, the system is actually used *against* the user paying for the service as it restricts him. A typical application where such a system is used *in favour* of the user are computations within an untrusted cloud setup. Consider a user who wants to outsource computation to a cloud provider which he does not trust. In this case, the user might want to deploy trusted modules onto the provider's remote machines without having to trust the remote environment, i.e., the provider's operating system and management software.

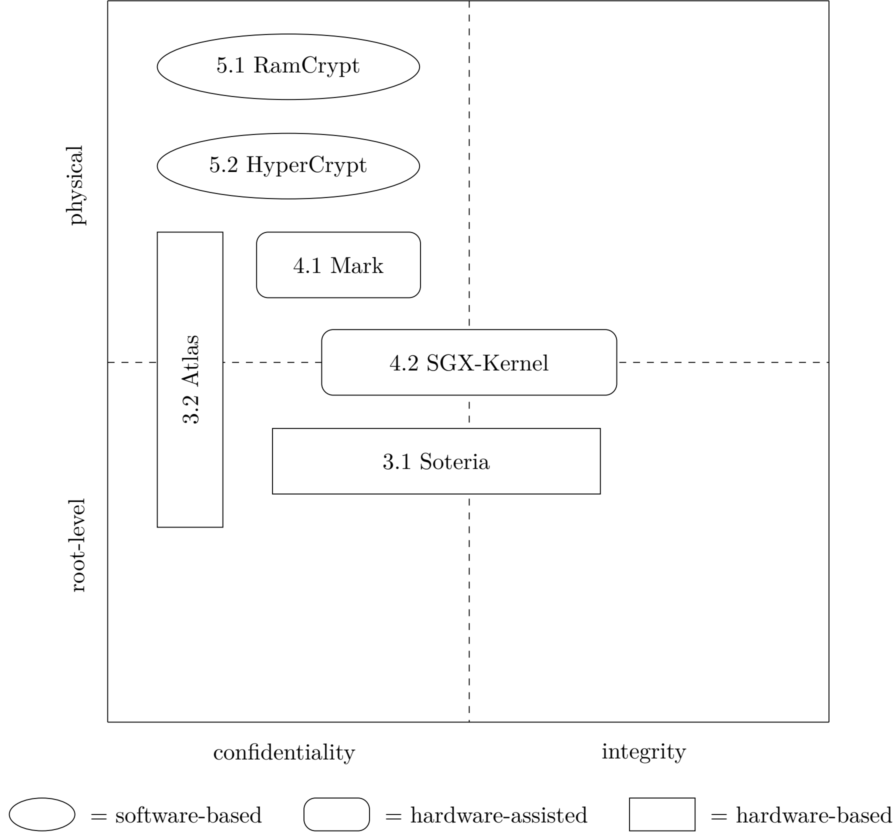

Figure 1.1: Categorization of our architectures with respect to attacker types and security properties.

Besides examples where one party runs a module in the environment of another party it does not trust, there are also scenarios where multiple parties are equally privileged but mutually not trusting each other. Embedded systems, like sensor nodes, often do not run an operating system, but multiple vendors might want to run modules which should not be able to access and modify each other's code and data. In such a scenario, the untrusted environment is composed by all modules but the current module.

Finally, even if there is only one single party operating a certain device, it might make sense to distinguish between trusted modules and an untrusted environment. Today's operating systems have huge code bases and cannot necessarily be trusted. Recent kernel exploits [6, 7] show that the highest privileged software can be attacked regularly. Thus, even with only one party involved trusted modules can shield critical applications against a compromised operating system or compromised applications. Our introductory example with the manipulated car shows that such threats are indeed practical.

The overall goal of this thesis is to make systems behave in an *expected way* even in untrusted environments, i.e., in the presence of strong attackers. To this end, we design and exploit architectures that are able to guarantee certain security properties. Figure 1.1 shows a categorization of the architectures within this thesis with respect to attacker types and security properties. We distinguish between the two attacker types *root-level* and *physical* as well as the two security properties *confidentiality* and *integrity*. Regarding physical attacks, we consider physical attackers with limited capabilities which are, for example, allowed to dump memory, but disallowed to decapsulate chips. Furthermore, while confidentiality generally refers to code and data confidentiality, integrity refers to the integrity of a trusted module's state as well as its associated code and data. Throughout this thesis our results are classified as hardware-based, hardware-assisted, or software-based architectures where hardware-assisted means software solutions built on top of existing architectures. Sect. 4.3 describes an attack against an architecture and is thus not listed in Figure 1.1.

1.1 Contributions

Our contribution to giving security guarantees in untrusted environments is threefold. In the first part of this thesis we design and extend trusted computing architectures for embedded systems. The main target of this part are small devices with a small or no operating system while the focus lies on code and data confidentiality against root-level attackers. In the second part, we exploit existing trusted computing architectures for general purpose hardware. We present non-standard applications of those architectures, but also show that no security guarantees can be given at all if the attacker assumptions are violated. Finally while in the first two parts trust is rooted in hardware, in the third and last part, we show which security guarantees can be given using pure software-based solutions with a focus on data confidentiality against physical attackers.

In detail our contributions in designing and extending trusted computing architectures are:

- We design and implement Soteria (Sect. 3.1), an architecture that provides offline software protection for low-cost embedded systems. In particular, we extend the trusted computing architecture Sancus which currently provides *integrity* and *authenticity* of software modules to also guarantee *confidentiality*.
- To this end, we design a scheme consisting of *software modules* and *loader modules*. The confidentiality of code and data of the protected software modules is guaranteed at any given point in time against all software attackers, including those who could gain system privileges. The loader modules themselves can be written in software and are not part of the minimal trusted computing base implemented in hardware.
- Based on hardware supported integrity checks, loader modules can decrypt a protected software module only if the integrity of both is not violated. The key for decryption is derived directly on the target system and hence, is not loaded from a trusted party during runtime. In particular, our solution does not have to attest its trusted execution environment to a remote party but bootstraps autonomously. In other words, Soteria is a solution for *offline* software protection.
- Because Soteria requires a static number of modules to be defined at synthesis time, we designed Atlas (Sect. 3.2), an entirely new security mechanism protecting application confidentiality with a fixed overhead that is independent of the number of applications running on the system.
- Atlas uses hardware-based memory encryption to protect application confidentiality in the event of a system compromise. As there is no need to keep track of state information, Atlas scales to an unlimited number of applications.

Our contributions in exploiting existing trusted computing architectures are:

- We leverage the Trusted Platform Module (TPM) of current commodity hardware to reveal boot process manipulations before a user enters his disk encryption password (Sect. 4.1). To this end, we introduce a *mutual authentication scheme* that proves the integrity of a computer to its users.
- Furthermore, we use an active USB drive that verifies the integrity of the boot process by utilizing sealed *nonces* to securely signal the integrity state of the PC to the user.
- While with the TPM the whole software stack is part of the trusted computing base, Intel's Software Guard Extensions (SGX) allow to dynamically establish roots of trust on general purpose hardware. We leverage SGX to shield kernel modules from the untrusted environment (Sect. 4.2). Thus, privileged malicious user space applications as well as already compromised parts of the kernel are prevented from attacking secured kernel modules.
- We practically demonstrate that SGX does not prevent secret information from being inferred from an enclave if the implementation within this enclave is vulnerable to cache attacks (Sect. 4.3). To this end, we describe a root-level cache attack relying on CPU pinning, influencing the hyperthreading affinity, and accessing Intel's Performance Monitoring Counters (PMC) against a vulnerable AES implementation running within an enclave.

Regarding software-based solutions, our contributions are:

- We propose RamCrypt (Sect. 5.1), a kernel-based encryption solution for entire address spaces of user-mode processes. Sensitive data within the address space of a process is encrypted on a per-page basis, thus data within core dumps and memory that gets accessed by debuggers, for example, are therefore protected.
- By using CPU-bound encryption, cryptographic keys are never stored in RAM, but solely inside CPU registers over the entire uptime of a system. All user-mode processes are protected with the same key, which is only visible in kernel mode.
- To be operating system independent and to protect also the address space of the kernel, we present HyperCrypt (Sect. 5.2), a hypervisor-based encryption solution for the guest's main memory. HyperCrypt is transparent for applications and the OS kernel running on top of it.

1.2 Related Work

Basic isolation concepts, like horizontal isolation of the system layer against applications, and vertical isolation of applications against each other, are available in modern operating systems since decades [149]. These basic isolation concepts are supported by hardware extensions like an MMU/MPU or CPU protection rings. To guarantee confidential execution of an application also in the presence of physical and root-level attackers, however, stronger degrees of isolation are required.

These degrees of isolation can only be provided by new hardware extensions that have an immutable trust anchor for user applications such as the TPM [157, 74] or Intel's SGX [87, 10, 115] for desktop computers and notebooks. On ARM, similar goals as those of SGX are pursued by TrustZone [13]. TrustZone allows only one enclave at a time, called the *secure world*, while the untrusted software stack is called *unsecure world*. Although TrustZone is available in ARM since years, on mass market products like iOS- or Android-driven smartphones it is mostly used during booting. In addition, it seems to be explored in academic publications [139].

The situation is slightly different for trusted execution on embedded systems without MMU support. Sancus [124, 146] enforces the integrity of software modules with the help of a dedicated program-counter based memory access logic in hardware. Other approaches for remote attestation on embedded devices are SMART [53], a recent proposal by Francillon et al. [58], and TrustLite [100]. Furthermore, researchers at IBM proposed an architecture called SecureBlue++ [168], which protects the confidentiality and integrity of an application's cache lines when they are evicted to main memory.

Besides the commercial and embedded trusted computing solutions, there are numerous other academic proposals for trusted computing architectures [147, 35, 54, 27, 42]. More details and an exhaustive comparison can be found in our recent survey [111].

Regarding attacks on SGX, Costan and Devadas [41] presented an analysis that points out weak spots of Intel SGX. The authors hypothesize that SGX does not implement protection measures against side-channel attacks. Xu et. al. [171] demonstrated controlled-channel attacks on protected applications, an attack called AsyncShock [164] targets synchronization bugs such as use-after-free and time-of-check-to-time-of-use within enclave code by manipulating the scheduler, and recently a more stealthy page table-based attack got published which avoids page faults [31]. Finally, there are independently developed cache attacks against SGX [26, 141] which are closely related.

A lot of work proposed in the recent years addresses memory disclosure problems. These related works can be categorized in three groups: (1) cryptographic processors, (2) CPU-bound encryption, and (3) full memory encryption.

Cryptographic processors try to exploit theoretic approaches such as homomorphic encryption [59, 28] or fully homomorphic encryption [61] to ensure that no sensitive data is visible at all as computations are directly performed on encrypted data. These approaches, however, are currently not yet practical. There are other theoretic proposals that suggest that only the CPU should access data in clear and that the communication to all peripheral and storage devices should be encrypted [29, 64]. Current practical solutions, however, only cover tamper-resistance and authenticity but not confidentiality of data [148].

Symmetric CPU-bound encryption schemes rage from register-based schemes as operating system patch [118, 22, 65, 143] to hypervisor-based solutions [120] and cache-based schemes [96]. There are even CPU-bound encryption schemes for asymmetric encryption algorithms as it turned out that asymmetric keys can be recovered from memory as well [84, 127]. In particular, there are CPU-bound RSA implementations that either are register based [60, 76] or based on hardware transactional memory [77]. All these solutions, however, just keep the encryption key and intermediate data out of memory but not other sensitive information because the secure storage area these solutions make use of is limited.

Full memory encryption solutions are most related to our work as a recent survey shows [86]. There are again theoretical approaches [48] but also practical implementations to encrypt, for example,

swap space [132]. Furthermore, a solution for embedded hardware [85] exists. The only solution, however, that also targets the Linux kernel [129] is not publicly available and does not store the encryption key outside RAM. Finally, AMD recently presented a security extension for their processors called Secure Memory Encryption (SME) [99] which also enables transparent memory encryption at page granularity.

1.3 Publications

The results within this thesis are exclusively taken from existing publications presented at peer-reviewed conferences and workshops as well as peer-reviewed journals. In particular, this thesis is based on the following publications:

[111] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers*, PP(99):1–1, 2017. doi: 10.1109/TC.2017.2647955. URL `https://doi.org/10.1109/TC.2017.2647955`.

[70] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix C. Freiling, and Ingrid Verbauwhede. Soteria: Offline Software Protection within Low-cost Embedded Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 241–250. ACM, 2015. doi: 10.1145/2818000.2856129. URL `http://doi.acm.org/10.1145/2818000.2856129`.

[47] Gabor Drescher, Christoph Erhardt, Felix C. Freiling, Johannes Götzfried, Daniel Lohmann, Pieter Maene, Tilo Müller, Ingrid Verbauwhede, Andreas Weichslgartner, and Stefan Wildermann. Providing security on demand using invasive computing. *it - Information Technology*, 58(6):281–295, 2016. doi: 10.1515/itit-2016-0032. URL `https://doi.org/10.1515/itit-2016-0032`.

[67] Johannes Götzfried and Tilo Müller. Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption. *ACM Trans. Inf. Syst. Secur.*, 17(2):6:1–6:23, 2014. doi: 10.1145/2663348. URL `http://doi.acm.org/10.1145/2663348`.

[134] Lars Richter, Johannes Götzfried, and Tilo Müller. Isolating Operating System Components with Intel SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution, SysTEX@Middleware 2016, Trento, Italy, December 12, 2016*, pages 8:1–8:6. ACM, 2016. doi: 10.1145/3007788.3007796. URL `http://doi.acm.org/10.1145/3007788.3007796`.

[73] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In Cristiano Giuffrida and Angelos Stavrou, editors, *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*, pages 2:1–2:6. ACM, 2017. doi: 10.1145/3065913.3065915. URL `http://doi.acm.org/10.1145/3065913.3065915`.

[72] Johannes Götzfried, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes. RamCrypt: Kernel-based Address Space Encryption for User-mode Processes. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 919–924. ACM, 2016. doi: 10.1145/2897845.2897924. URL `http://doi.acm.org/10.1145/2897845.2897924`.

[71] Johannes Götzfried, Nico Dörr, Ralph Palutke, and Tilo Müller. HyperCrypt: Hypervisor-Based Encryption of Kernel and User Space. In *11th International Conference on Availability, Reliability and Security, ARES 2016, Salzburg, Austria, August 31 - September 2, 2016*, pages 79–87. IEEE Computer Society, 2016. doi: 10.1109/ARES.2016.13. URL `https://doi.org/10.1109/ARES.2016.13`.

These publications are used throughout this thesis as follows:

- Sect. 2.1 and Sect. 2.2 are mostly based on our survey *"Hardware-Based Trusted Computing Architectures for Isolation and Attestation"* [111] which is joint work with our colleagues from KU Leuven. The architecture descriptions taken from this publication have been originally written by the author of this thesis.

- Sect. 3.1 is based on our conference publication *"Soteria: Offline Software Protection Within Low-cost Embedded Devices"* [70]. The hardware, software, and toolchain implementation as well as most parts of the paper have been written by the author of this thesis. Parts of this publication are also used for the background information about Sancus given in Sect. 2.2.1.

- Sect. 3.2 is based on currently unpublished work, but parts of the architecture description can already be found in our journal publication *"Providing security on demand using invasive computing"* [47]. For Atlas, the hardware has been developed by Pieter Maene while the software and toolchain have been developed by the author of this thesis. Consequently, the hardware description is originally mostly written by Pieter Maene while the software and toolchain descriptions are mostly written by the author of this thesis.

- Sect. 4.1 is based on our journal publication *"Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption"* [67]. Although this journal publication is a follow-up publication to previous work called Stark [121], the hardware, software, and toolchain implementation for Mark as well as most parts of the paper have been written by the author of this thesis. Parts of our journal publication are also used for the background information about full disk encryption given in Sect. 2.3.

- Sect. 4.2 is based on our workshop publication *"Isolating Operating System Components with Intel SGX"* [134]. This publication is the result of a master thesis by Lars Richter supervised by the author of this thesis.

- Sect. 4.3 is based on our workshop publication *"Cache Attacks on Intel SGX"* [73]. This publication is the result of a bachelor thesis by Moritz Eckert supervised by the author of this thesis.

- Sect. 5.1 is based on our conference publication *"RamCrypt: Kernel-based Address Space Encryption for User-mode Processes"* [72]. The implementation as well as the publication itself was mainly written by the author of this thesis. Parts of this publication are also used for the background information about CPU-bound encryption given in Sect. 2.4.

- Sect. 5.2 is based on our conference publication *"HyperCrypt: Hypervisor-based Encryption of Kernel and User Space"* [71]. While the publication was written by the author of this thesis, the implementation has been mainly done by Nico Dörr as part of his bachelor thesis and master project which both have been supervised by the author of this thesis.

Besides the core publications this thesis is based on, the author of this thesis has authored and contributed to further publications:

[66] Johannes Götzfried and Tilo Müller. Fast Software Encryption with SIMD: How to speed up symmetric block ciphers with the AVX/AVX2 instruction set. In *Proceedings of the 6th European Workshop on System Security, EUROSEC 2013, Prague, Czech Republic, April 14, 2013*. ACM, 2013. URL `http://www1.cs.fau.de/filepool/projects/avx.crypto/avxcrypto.pdf`.

[65] Johannes Götzfried and Tilo Müller. ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, pages 161–168, 2013. doi: 10.1109/ARES.2013.23. URL `https://doi.org/10.1109/ARES.2013.23`.

[68] Johannes Götzfried and Tilo Müller. Analysing Android's Full Disk Encryption Feature. *JoWUA*, 5(1):84–100, 2014. URL `http://isyou.info/jowua/papers/jowua-v5n1-4.pdf`.

[12] Maxim Anikeev, Felix C. Freiling, Johannes Götzfried, and Tilo Müller. Secure garbage collection: Preventing malicious data harvesting from deallocated Java objects inside the Dalvik VM. *J. Inf. Sec. Appl.*, 22:81–86, 2015. doi: 10.1016/j.jisa.2014.10.001. URL `https://doi.org/10.1016/j.jisa.2014.10.001`.

[69] Johannes Götzfried, Johannes Hampel, and Tilo Müller. Physically Secure Code and Data Storage in Autonomously Booting Systems. In *10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015*, pages 199–204. IEEE Computer Society, 2015. doi: 10.1109/ARES.2015.19. URL `https://doi.org/10.1109/ARES.2015.19`.

[170] Alexander Würstlein, Michael Gernoth, Johannes Götzfried, and Tilo Müller. Exzess: Hardware-Based RAM Encryption Against Physical Memory Disclosure. In Frank Hannig, João M. P. Cardoso, Thilo Pionteck, Dietmar Fey, Wolfgang Schröder-Preikschat, and Jürgen Teich, editors, *Architecture of Computing Systems - ARCS 2016 - 29th International Conference, Nuremberg, Germany, April 4-7, 2016, Proceedings*, volume 9637 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2016. doi: 10.1007/978-3-319-30695-7_5. URL `https://doi.org/10.1007/978-3-319-30695-7_5`.

[165] Andreas Weichslgartner, Stefan Wildermann, Johannes Götzfried, Felix C. Freiling, Michael Glaß, and Jürgen Teich. Design-Time/Run-Time Mapping of Security-Critical Applications in Heterogeneous MPSoCs. In Sander Stuijk, editor, *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2016, Sankt Goar, Germany, May 23-25, 2016*, pages 153–162. ACM, 2016. doi: 10.1145/2906363.2906370. URL `http://doi.acm.org/10.1145/2906363.2906370`.

[45] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. SOFIA: Software and control flow integrity architecture. *Computers & Security*, 68:16–35, 2017. doi: 10.1016/j.cose.2017.03.013. URL `https://doi.org/10.1016/j.cose.2017.03.013`.

[125] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.*, 20(3): 7:1–7:33, 2017. doi: 10.1145/3079763. URL `http://doi.acm.org/10.1145/3079763`.

[98] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, pages 2:1–2:10. ACM, 2017. doi: 10.1145/3098954.3098995. URL `http://doi.acm.org/10.1145/3098954.3098995`.

1.4 Outline

In Chapter 2, the background chapter, we first introduce the concept of trusted computing by explaining basic security properties as well as non-functional architectural features of trusted computing architectures (Sect. 2.1). We then present three hardware-based trusted computing architectures which are extended or exploited within this thesis (Sect. 2.2). Afterwards, we describe Full Disk Encryption (FDE) solutions together with current attacks (Sect. 2.3) and finally, explain the concept of CPU-bound encryption (Sect. 2.4).

In Chapter 3, we design two hardware-based trusted computing architectures for embedded devices with a focus on code and data confidentiality. We first present Soteria by extending the existing hardware-based trusted computing architecture Sancus which currently provides integrity and authenticity of software modules to also guarantee confidentiality (Sect. 3.1). Because Soteria, is restricted to a static number of software modules, we developed a second architecture, Atlas, which uses hardware-based memory encryption to protect the confidentiality of applications (Sect. 3.2).

In Chapter 4, we leverage the Trusted Platform Module (TPM) built into virtually every modern desktop PC and notebook to provide secure full disk encryption (Sect. 4.1). To this end, the integrity of the boot process is verified and can be checked through the use of an active USB drive. We then use Intel's Software Guard Extensions (SGX) to shield kernel modules from an untrusted environment including user space applications as well as potentially compromised parts of the kernel (Sect. 4.2). Finally, we show that SGX cannot protect secret information from being inferred if implementations relying on SGX are vulnerable to cache attacks and thus, particular care needs to be taken when writing enclave code (Sect. 4.3).

In Chapter 5, we describe how memory disclosure attacks can be mitigated with software-only techniques by presenting two solutions, RamCrypt (Sect. 5.1) and HyperCrypt (Sect. 5.2). RamCrypt protects sensitive information by encrypting process address spaces on a per page basis at the kernel-level while HyperCrypt works on the hypervisor-level and also encrypts kernel space.

In Chapter 6, we finally conclude this thesis.

2 Background

In this chapter, we give the background information required to understand the remaining chapters of this thesis. While specific topics are discussed in the respective sections, this chapter focuses on general information which is needed within more than one section. First, we explain the concept of trusted computing (Sect. 2.1) together with general security properties common architectures provide. Second, we introduce three hardware-based trusted computing architectures (Sect. 2.2) which are extended and exploited during this thesis. Third, we provide an explanation about full disk encryption (Sect. 2.3) together with state-of-the-art attacks against it. Finally, we describe a technique called CPU-bound encryption (Sect. 2.4) which is primarily used by our software-based memory encryption solutions.

2.1 Trusted Computing

Trusted computing has the goal to make systems behave in an expected way towards their users. In particular, certain security guarantees about the behaviour of software running on not necessarily user controlled devices are given. A device is said to be trusted *if it always behaves in the expected manner for the intended purpose* [112]. Thus, if an attacker gets access to the device, it should be impossible for him to manipulate the software, i.e., to violate the security guarantees, protected with trusted computing techniques. Of course, trusted computing cannot protect against all possible attacks, but only against a certain set of attacks which is covered by the attacker model of the specific architecture.

Trusted computing architectures can be implemented purely in software, purely in hardware, or as a combination of both. Depending on the specific implementation, different attacks can be defeated. Within this thesis, we consider two kinds of attackers, namely physical and root-level attackers. For physical attacks, countermeasures can be implemented in both, software and hardware. To protect against root-level attackers, i.e., attackers which control the whole software stack potentially including operating system and hypervisor code, trust must be rooted in hardware. While our software-only solutions (Chapter 5) have been developed from scratch, we use the hardware-based trusted computing architectures from Sect. 2.2 as a basis for our hardware (Chapter 3) and hybrid (Chapter 4) solutions.

In general, any trusted computing architecture only protects against attackers with a specific set of capabilities. For root-level attacks, the attacker is assumed to be in complete control of all software on all the devices in the system, except for the software that is part of the TCB. This means that he can tamper with the OS, or even deploy malicious software components. Some architectures use software modules that are part of the TCB, and it is assumed that the attacker cannot change these. Second, the attacker is assumed to be in control of the communication channel to the device. He is therefore capable of sniffing the network, modifying traffic, and mounting Man-in-the-Middle (MitM) attacks. These abilities are important when considering attestation. Third, the Dolev-Yao attacker model [46] is used, where the attacker is assumed to be unable to break cryptographic primitives, but can perform protocol-level attacks. Finally, giving availability guarantees is out of scope for most architectures and thus, Denial-of-Service (DoS) attacks typically cannot be prevented.

For physical attacks, the attacker does have physical access to the hardware, can probe main memory, and disconnect components. Typically, however, only physical attacks on off-chip memory are allowed, but not on any hardware components which are part of the TCB, such as the CPU

package and sometimes also buses. Also hardware side-channel attacks usually cannot be prevented. The detailed attacker model for each solution is given in the respective section.

In the following, we introduce a few basic terms regarding trusted computing which are used throughout this thesis:

- *Protected Module Architecture (PMA)*: Because software is too complex to prove that an application is bug-free and attackers are always trying to exploit those bugs, the concept of Protected Module Architectures (PMAs) [113] has been introduced. The idea is, that basic components are separated into small *Protected Modules* which can ideally be verified regarding their correctness. These modules are then isolated from other software on the system, including high privileged software such as an OS. Consequently, these modules cannot be tampered with. While PMAs are most widely known for hardware-based architectures, it has been shown that PMAs can be implemented at any level of the architecture, from the hardware to the OS kernel [146].

- *Protected Module*: Protected modules are the protected entities within PMAs. There is a different terminology for each architecture and we keep the original terminology of the respective authors throughout this thesis. Thus, protected modules will also be referred to as *Trusted Modules*, *Secure Modules*, *Software Modules (SMs)*, or *Enclaves*.

- *Trusted Computing Base (TCB)*: TCB refers to the set of hardware and software components which are critical to an architecture's security. The careful design and implementation of these components are paramount to the overall security. The components of the TCB are designed so that, when other parts of the system are exploited, they cannot make the device misbehave. Ideally, a TCB should be as small as possible in order to guarantee its correctness.

- *Measuring*: Measuring is a method to verify the integrity and authenticity of software components. This is done by calculating a hash or Message Authentication Code (MAC) of their code and data. Some designs also include other identifying information, like the memory layout. The measurement result can then be used to attest the component's state. Since a hash or MAC value for a given input is probabilistically unique, it also identifies the state of the software component at that time.

- *Trust Chain*: A chain of trust can be built by verifying each component's validity from the bottom up. Usually each component is measured in software by the previous component right before it is executed.

Each trusted computing architecture is only able to guarantee a certain subset of security properties (Sect. 2.1.1). Besides the essential security guarantees for the protected modules, trusted computing architectures also have non-functional properties or architectural features (Sect. 2.1.2). Those features do not directly affect security, but nevertheless influence the fields of application for an architecture.

2.1.1 Security Properties

We now introduce security properties which are potentially guaranteed by the three hardware-based trusted computing architectures presented in Sect. 2.2. The solutions developed throughout this thesis rely on these properties:

- *Isolation*: Isolation denotes a hardware-based architectural mechanism that provides access control for software and its associated data. By placing code and data inside a protected module, no software outside it can read or write its runtime state or modify its code. Execution of code inside such a module can only be started from a single predefined location. Such an *entry point* ensures that attackers cannot reuse the module's own code to extract secrets or implement malicious behaviour.

- *Confidentiality*: A trusted computing architecture which guarantees confidentiality ensures that sensitive code or data remains secret and cannot be obtained by untrusted hardware or software. Confidentiality is related to isolation in the sense that confidentiality of code or data can be achieved through isolation. However, confidentiality can also be enforced through encryption like, for example, in the case of Atlas (Sect. 3.2). Any architecture presented within this thesis guarantees the confidentiality of data, but not necessarily the confidentiality of code. In fact, our extension of Sancus (Sect. 2.2.1) called Soteria (Sect. 3.1) adds code confidentiality to Sancus.

- *Integrity*: Integrity means that code or data manipulations of protected modules are at least detected. This could be done, for example, by adding integrity tags to code and data stored in memory. Typically, however, integrity is provided through isolation which also has the advantage of not only being able to detect manipulations after they have happened, but rather to prevent them in the first place.

- *Attestation*: Attestation is the process of proving to an authorized party that a specific entity is in a certain state. In order to give strong security guarantees, an architecture which supports attestation should guarantee integrity of the attested state as well. Trusted computing architectures may provide *local* and *remote* attestation. The former refers to one software module attesting its state to another running on the same platform, while the latter attests to a remote party residing outside the trusted system. Local attestation is also referred to as *secure linking*. A common way to implement attestation is to measure software modules during their initialization, while preventing later modifications by means of isolation.

- *Sealing*: Sealing wraps confidential code or data in such a way that it can only be unwrapped under certain circumstances. Code or data are wrapped by binding it either to a specific device, a certain configuration of the device, the state of a software module, or a combination of these. It can then only be unwrapped when the binding conditions are met, e.g., on the same device or one which runs the same configuration. Sealing is usually based on encryption, and relies on similar mechanisms as software attestation, i.e., the key for encrypting confidential code or data is typically derived from the software module's measurement taken during initialization.

- *Root of Trust (RoT)*: In order to keep the TCB as small as possible, most trusted computing technologies build trust chains. However, these chains always need to be anchored in a component that is inherently trusted, which are referred to as RoTs. A Dynamic RoT (DRoT) is established for a software module at runtime, measuring the module's state right before execution starts. It is typically combined with isolation to protect against Time-of-Check Time-of-Use (TOCTOU) vulnerabilities as well, where an attacker changes the module's code after it has been measured.

- *Side-Channel Resistance*: A trusted computing architecture is called side-channel resistant with respect to software attackers if no software module, including privileged software like an OS, is able to deduce information about the behaviour of other modules apart from their I/O behaviour. Specifically, information flow through untrusted channels, such as caches, or information revealed by page faults cannot leak to untrusted software. An architecture with side-channel resistance should take care to flush caches during context switches. Information leakage due to page faults, for example, can be prevented by giving each software module the ability to handle its own page faults.

- *Memory Protection*: Memory protection specifically refers to protecting the integrity and authenticity of data sent over system buses or stored in external memory from physical attacks. It includes both *passive*, e.g., bus snooping, and *active*, e.g., fault injection, attacks. First, this means that data has to be encrypted to prevent sensitive data from leaking. Second, it also has to be integrity-protected, for example, using integrity tags. Third, replay attacks, where previously valid memory contents are restored, have to be prevented as well. These operations have to be performed when data is sent to or fetched from external memory.

2.1.2 Architectural Features

Besides the properties that are crucial to the security of a system, there are also a number of architectural features which can be assigned to a trusted computing architecture:

- *Lightweight*: An architecture is called lightweight when it does not use a Memory Management Unit (MMU). Lightweight embedded systems have very simple memory hierarchies, and therefore do not need complex memory management. Furthermore, they only run a limited number of applications, which share the memory space cooperatively, not requiring virtual addressing to map the memory.
- *Coprocessor*: Many trusted computing architectures require security mechanisms to be implemented in hardware. In case of a coprocessor, this functionality is added as a separate chip or module which interfaces with the main processor. Alternatively, the functionality can be integrated inside the processor. Trusted computing architectures that are integrated in a processor can typically provide stronger security guarantees than coprocessor-based designs, because data does not have to leave the CPU.
- *Hardware-Only TCB*: It is typically better for the TCB to rely on hardware, as this provides stronger security guarantees, such as protection from an untrusted OS.
- *Preemption*: When preemption is supported, the system can suspend running tasks at any time, without first obtaining permission from the task. This makes it possible to handle interrupts, but also allows preemptive scheduling of multiple protected modules. Preemption mainly impacts the context switching logic, since the architecture now has to ensure that no sensitive information can leak between modules, as this would violate the isolation primitive. Without support for preemption, applications have to run cooperatively, i.e., they need to call each other after finishing a task.
- *Dynamic Layout*: A static layout is often used when all software shares the same address space, and no MMU is present to provide virtual memory for different applications. It has the disadvantage that one trusted entity, e.g., the hardware or software manufacturer or a system integrator, needs to compile all software and fix the layout before deployment to the target device. With a dynamic layout, however, applications can be loaded to locations that do not need to be known at compile time.
- *Upgradeable TCB*: Architectures which have a HW-only TCB are not upgradeable, because their components can no longer be changed after being manufactured. However, some designs include trusted software components, typically to implement functionality which would be too expensive in hardware. These components are then protected by a hardware mechanism. This not only results in more design flexibility, but also enables upgrading the TCB at a later point in time, e.g., when a bug has been discovered or to add new functionality.
- *Backwards Compatibility*: When adding features, an important consideration is whether legacy code runs on the modified architecture without any changes, possibly after recompilation. Since these applications do not use the introduced security mechanisms, they typically do not receive any of the associated security guarantees.

2.2 Hardware-based Trusted Computing Architectures

This section gives a detailed description of three hardware-based trusted computing architectures which are extended or exploited within this thesis. First, we present Sancus (Sect. 2.2.1) which is the basis of our newly designed trusted computing architecture Soteria (Sect. 3.1). Sancus is a protected module architecture targeting small embedded devices with no operating system and a statically defined number of software modules. Second, we introduce the Trusted Platform Module (TPM) (Sect. 2.2.2), which is built into virtually every modern desktop computer or notebook, together with an extension called Trusted Execution Technology (TXT). The TPM is a basic building block for our secure full disk encryption solution Mark (Sect. 4.1). Finally, we describe Intel's Software Guard Extensions (SGX) (Sect. 2.2.3) which enable establishing dynamic RoTs inside regular applications. We leverage SGX to isolate kernel components from each other (Sect. 4.2) and show that SGX enclaves are generally vulnerable against cache attacks (Sect. 4.3).

2.2.1 Sancus

Sancus [124] is a security architecture for networked embedded devices that supports third-party software extensions. It enables software from different mutually untrusted parties to run on the same device while providing strong guarantees that software remains untampered. This includes support for isolating different software modules, remotely attesting software modules to detect tampered software, and authentication of messages to software providers.

Sancus has a small hardware-only trusted computing base, and uses a minimal amount of hardware features. Its attacker model assumes the attacker to be in complete control of all software, as no software is part of the TCB. The results received from a module can always be validated to see if they are genuine. The system, however, makes no guarantees on availability and confidentiality for code.

The Sancus project consists of a hardware description of a Sancus-enabled openMSP430 core [62], as well as a C compiler for Sancus-enabled devices. The compiler supports annotations in the source code that allow developers to easily develop code for this architecture. The full project is provided as open source.

The system model of Sancus is as follows. An Infrastructure Provider (*IP*) owns a set of networked nodes (N_i). Each node consists of a low-end microcontroller, in our case an MSP430, with a single address space for instructions and data. Different Software Providers (SP_j) make use of the infrastructure provided by *IP*. These *SP*s can make software available on the nodes of the infrastructure by compiling their software into a software module ($SM_{j,k}$). An *SM* consists of a binary file which consists of a text and data section, as well as header information that specifies which regions of memory are protected/unprotected. An *SM* can only be loaded onto a node that is part of the infrastructure on behalf of a software provider.

Sancus provides the following security properties: First, software modules are isolated from each other. This ensures that other *SM*s cannot read or modify each other's state. Second, the hardware TCB performs remote attestation, allowing the system to make strong guarantees on the integrity and authenticity about the running software on the system. Third, a software provider can verify any software module loaded on *IP*. Fourth, tamperproof communication is provided between modules with integrity and authenticity guarantees. Fifth, software can securely link to each other, meaning that software modules can call each other with the assurance that the intended module is being called.

Sancus Module Layout Sancus isolates software modules using program-counter based memory access control. An overview of the resulting memory layout is shown in Figure 2.1. Two types of memory can be distinguished: There is *protected memory* assigned to a specific module, and everything else referred to as *unprotected memory*. The protected memory region is divided into two sections, one for code and constants and one for protected data. The boundaries of these regions

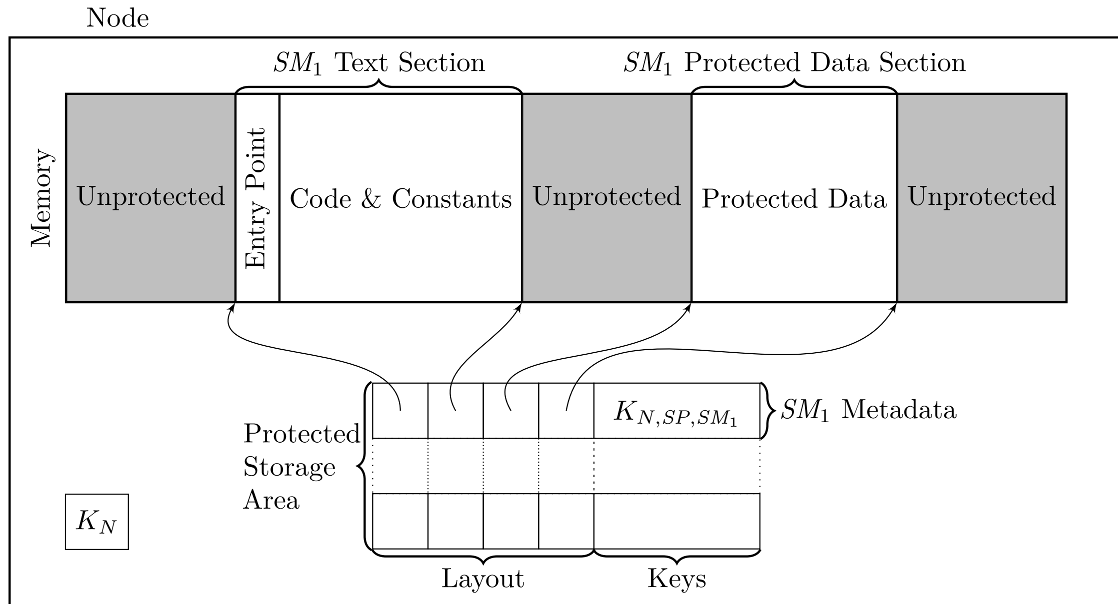

Figure 2.1: The layout of a Sancus software module in memory [124].

are stored in dedicated registers which are added to the processor architecture. These registers are used as inputs to the memory access logic which compares them to the current program-counter to enforce the access rights.

The code in the text section can *only be executed* when the program-counter is either at the entry point or the module is already executing. Note, however, that it is possible to *read the code* from anywhere. Contrary to that, the module's data can only be read or written when the program-counter is in the module's text section.

Sancus adds two custom instructions which can be used by application developers to isolate their application, namely `protect` and `unprotect`. When `protect` is called, the implications are threefold: The memory boundaries are checked for overlap with existing ones and are written to their respective dedicated registers, enabling access control. The last step is deriving the module's key and storing it in its dedicated register. Executing `unprotect` clears the boundary registers, thereby deactivating the memory protection.

Sancus Key Derivation Sancus introduces three types of keys to the system. First, the node key K_N is bound to the hardware and only known by the *IP*. $K_{N,SP}$ is used by the *SP* to deploy modules to a specific node. It is derived from K_N together with the provider's unique public ID using a key derivation function *kdf* as follows:

$$K_{N,SP} = kdf(K_N, SP)$$

Third, $K_{N,SP,SM}$ is shared between the *SP* and *SM*. It is obtained from $K_{N,SP}$ and the identity of *SM* as follows:

$$K_{N,SP,SM} = kdf(K_{N,SP}, SM)$$

Figure 2.1 shows that this key is stored in the protected storage area alongside the memory boundaries.

2.2.2 Trusted Platform Module

The Trusted Platform Module (TPM) version 1.2 [157, 158, 159] was specified by the Trusted Computing Group (TCG) in 2011. It is a coprocessor on the motherboard, which is capable of

storing keys and performing attestation. It is a passive piece of hardware, meaning that software can interact with the TPM, but needs to do so explicitly. To give a local or remote party guarantees, any software that runs on the target device, i.e., the boot loader, Operating System (OS), and applications, needs to be measured successively by the TPM, and is consequently part of the TCB. Code manipulation is only detected during measurement, and all parts of the software are considered trusted after loading. The TPM only provides limited protection against physical attacks, because not only the CPU package, but the TPM chip and all connecting buses are part of the TCB as well. For instance, a physical attacker can compromise software integrity by tapping the Low Pin Count (LPC) bus between the TPM and CPU [103].

The design principles published by the TCG in version 1.2 specify the minimum functionality and cryptographic primitives that are required for TPMs, but a TPM manufacturer is allowed to extend the hardware module with additional functionality. Each TPM has to be equipped with a Random Number Generator (RNG) as a randomness source and an RSA implementation with at least 2048-bit keys. The minimum requirement for software measurement is the SHA-1 hash algorithm. At manufacturing time, the Endorsement Key (EK) is generated and written to persistent memory within the TPM. The EK is unique to every TPM chip, not known to the user, and serves as master key for all operations provided by the TPM. In addition to the EK, Attestation Identity Keys (AIKs) and storage keys can be generated on the fly and stored in volatile memory within the TPM. AIKs are used for all operations directly involving digital signatures, whereas the storage keys are used for encryption and decryption of data. Finally, a TPM contains a certain number of Platform Configuration Registers (PCRs), which are capable of storing successive hash values for code or data that is sent to the TPM and are important for remote attestation. The TCG also specified TPM version 2.0 [160] in 2014, supporting a larger variety of cryptographic algorithms, multiple banks of PCRs, and three hierarchies instead of only one.

With the minimum functionality required by the TCG, each TPM is able to perform at least three different operations. First, it is able to bind code or data to a given device by encrypting it with one of the storage keys. The encrypted code or data can then only be decrypted on the same platform, because the particular storage key cannot be extracted from volatile memory within the TPM.

Second, a TPM is able to attest to another party that the given device is currently running a certain software configuration. To this end, code or data sent to the TPM is hashed together with values of specific PCRs, and the result is again stored in the same PCRs. This way, each software component running on the device is able to successively extend a measurement over all components running on the device. The PCRs are initially set to a fixed value before starting the system, and because the hash function is irreversible, it is not possible to set the PCRs to a user-defined value. The resulting hash values from the PCRs can then be cryptographically signed by an AIK, and this signature can later be verified by a second party. When this party receives a correctly signed value, it can be sure that the system runs a certain software configuration, because this signed message could not have been created without going through the software measuring process.

Third, code or data can be sealed to a given device in a certain software configuration. In general, sealing works similar to binding, i.e., code or data is encrypted and decrypted, but it is additionally ensured that sealed code or data is only decrypted if the platform configuration has not changed in between. To check against changes of the platform configuration, the PCR values are saved together with the sealed code or data and checked against the current PCR values during unsealing.

Attestation and sealing only behave as intended if the platform configuration is measured from the earliest boot step, up to the currently running software component, because otherwise malicious software could potentially exclude itself from the measurement. This restriction is the biggest disadvantage of the standalone TPM over other solutions that support DRoTs.

To overcome the restriction of all software having to be part of the TCB, Intel introduced its Trusted Execution Technology (TXT) [74], which also uses the TPM chip, but allows dynamically establishing a new RoT for software running in a virtualized environment besides the usual software stack. TXT ensures that the virtualized software has exclusive control over the device by suspending all other running software, i.e., the OS and all applications. When switching to trusted TXT software, the CPU essentially performs a warm reset and initializes a certain subset of PCRs with

a new value. The TXT software can then extend this measurement and attest to a second party that it has not been modified before being loaded. Since it monopolizes all resources once it has been loaded, the integrity of the TXT software is guaranteed over its entire runtime.

Although TXT can be used to overcome the restriction of all software having to be part of the TCB, it still has some issues. Suspending all other applications on the device for the TXT software to run negatively impacts performance, or might even lead to losing interrupts depending on its size. Since the TXT software has to run exclusively, it cannot easily use functionality of untrusted software and needs to perform expensive context switches. Finally, all physical attacks that succeed for a standalone TPM, e.g., LPC bus tapping, also succeed for TXT. Fides [145], Flicker [113], and TrustVisor [114] are examples of architectures which build on the functionality offered by TXT.

2.2.3 Software Guard Extensions

In 2013, Intel announced SGX [115], which enables establishing dynamic RoTs inside regular applications, without monopolizing the system. SGX supports protecting an application's code as well as data [87], is able to guarantee integrity, and provides local and remote attestation [10]. In addition, SGX includes physical attacks on communication channels and main memory in the attacker model.

In SGX, the protected parts of an application are placed within so-called *enclaves*. An enclave can be seen as a protected module within the address space of a given process, and enclave accesses obey the same address translation rules as those to usual process memory, i.e., the OS and the hypervisor are still in charge of managing the page tables. This has the advantage that SGX is fully compatible with existing memory layouts, usually configured and managed by an MMU, and also works well in a multi-core environment. Although an enclave resides in the usual process address space, there are certain restrictions in enclave mode. For example, system calls and instructions that would cause a trap into the OS or hypervisor, such as `cpuid`, are not allowed and it is necessary to leave enclave mode before dispatching them. Furthermore, this mode can only be entered from user mode, which essentially means enclaves can only be used within applications, but not the OS [88].

An SGX-enabled CPU ensures in hardware that non-enclave code, including the OS and potentially the hypervisor, cannot access enclave pages. Specifically, a region called the Processor Reserved Memory (PRM), which contains the Enclave Page Cache (EPC) and the Enclave Page Cache Map (EPCM), is protected by the CPU against all non-enclave accesses. The EPC stores enclave pages, i.e., enclave code and data, while the EPCM stores state information about the pages currently held within the EPC. The state information consists of the enclave page access rights and the page's virtual address when the enclave was created, amongst others. For each (non-cached) access of an EPC page, the current access rights and virtual address are checked against the state stored within the EPCM, and if a mismatch is detected, access is denied. The caching of state information is necessary, because all software, including at system level, is considered untrusted, and therefore attacks such as enclave layout changes through remapping have to be prevented directly in hardware. If the capacity of the EPC is exceeded, enclave pages might be written out to a memory region outside the PRM by the OS, but are then transparently encrypted with the help of a hardware Memory Encryption Engine (MEE), which is inside the CPU package.

Before an enclave can be used, it has to be created and initialized by untrusted software. The hardware ensures that an enclave's pages can only be modified before initialization is finished. All page contents, including code and static data, are measured during initialization. As this measurement depends on all contents of the enclave, and later modifications are prevented, it can be used as a basis for local or remote attestation. All operations involved in the management of an enclave, e.g., enclave creation, initialization, and destruction, are performed by system software (ring zero), while entering and leaving the enclave is done by the application software (ring three). The latter is implemented similarly to system calls, i.e., an enclave has its own execution context, there is a single entry point into the enclave, and dedicated instructions need to be called. Upon leaving the enclave, the context of the current thread is saved within an EPC page and all registers are cleared. The appropriate context is loaded again when the enclave is entered. If an interrupt occurs

during enclave execution, an Asynchronous Enclave Exit (AEX) is performed by the CPU, which also saves the current enclave execution context and ensures that no data leaks to the untrusted system software handling the interrupt.

Within an enclave, other features are provided in addition to confidentiality and integrity of code and data. One enclave can attest to another that it has been loaded as intended by sending a *report*. The report includes information about the enclave (the measurement) or enclave author. This process is called *local attestation*. With the help of a trusted *quoting enclave* provided by Intel, the report can be wrapped into a *quote*, converting the local attestation to a remote attestation by signing the quote with an asymmetric attestation key, which is part of Intel's Enhanced Privacy Identifier (EPID) group signature scheme [95]. The quote can be verified by a remote party with the corresponding verification key provided by Intel. Besides local and remote attestation, data produced within an enclave can also be *sealed* to the enclave and, for example, written to memory outside the PRM. Sealed data can serve as permanent storage and retains information during different runs of an enclave. Local attestation, remote attestation, and sealing all rely on the non-forgeability of the initial enclave measurement.

Intel uses dedicated enclaves for complex functionality which would be expensive to implement in hardware, like the asymmetric cryptography needed for remote attestation. It also provides a *launch enclave* required to launch any other enclave, a *provisioning enclave* to initially provision asymmetric keys for attestation to end-user devices, and the previously mentioned *quoting enclave* to cryptographically sign the attestation quotes. The downside of this approach is that Intel has a de facto monopoly regarding enclave signing and remote attestation, as Intel decides which enclaves are allowed to run and everybody who wants to verify quotes needs to have an agreement with Intel.

More details about SGX in general can be found in an exhaustive academic summary [41]. There are two academic solutions which use SGX in an untrusted cloud context, namely Haven [18] and VC3 [140]. However, neither solution used real hardware, but relied on an emulator instead. Finally, AMD recently presented security extensions for their processors called Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) [99]. The former adds memory encryption at page granularity to protect data in memory. The latter relies on this unit to isolate virtual machines from each other as well as the hypervisor.

2.3 Full Disk Encryption

Full Disk Encryption (FDE) protects sensitive data against unauthorized access in the event of a device being physically lost or stolen. The risk of data exposure is reduced by rendering disks unreadable to unauthorized users through encryption technologies like AES [55]. Unlike file encryption, FDE encrypts whole disks transparently on the operating system level, without the need to take action for single files. Therefore, FDE is more user-friendly than file encryption, and consequently, it can be considered best practice for securing sensitive data against accidental loss and theft.

With Mark (Sect. 4.1), we designed a secure FDE solution to protect against attacks with repeated physical access. Thus, in this section, we argue why current FDE solutions are insufficient to protect against such attacks. We explain traditional simple cold boot and evil maid attacks (Sect. 2.3.1) followed by complex bootkit attacks (Sect. 2.3.2) which are able to break even the most sophisticated FDE solutions.

2.3.1 Cold Boot and Evil Maid Attacks

FDE cannot protect sensitive data when a user logs into the system and leaves it unattended. Likewise, FDE does not protect against system subversion through malware. In case of malware infiltration, adversaries can access data remotely over a network connection with user privileges after the user logged in. Hence, hard disk encryption is only intended for scenarios in which an adversary gains *physical access* to a target.

Given physical access, two kinds of attacks must be distinguished: opportunistic and targeted attacks. In *opportunistic* attacks, the adversary steals a computer and immediately tries to retrieve its data. This is the attack scenario which is withstood by all widespread FDE solutions, including BitLocker [116] and TrueCrypt [156]. If an attacker simply grabs a computer, data decryption without having the key or password is impossible with modern crypto primitives like AES. However, a more careful adversary can carry out a *targeted* attack on the key, password, or access control management. For this *cold boot attacks* [81, 82, 75] and *DMA attacks* over FireWire [23], PCIe [34] or Thunderbolt [135] can be used. However, keylogging attacks via *bootkits* [106] are also a notable threat.

Software-based FDE needs to modify the Master Boot Record (MBR) of a hard drive in order to present a pre-boot environment for user authentication. Commonly, pre-boot screens ask users for credentials in form of secret passwords or passphrases, but they can also ask for credentials such as smart cards and tokens. Only after a user is authenticated, the operating system is decrypted and prepared to take over system control.

However the MBR of an encrypted hard drive can easily be manipulated because it is necessarily left unencrypted for bootstrapping since CPUs can interpret unencrypted instructions only. As a consequence, bootkits can always be placed in the MBR to subvert the original bootloader with software-based keylogging. Such attacks are referred to as *evil maid attacks* [91] and typically require physical access to the target machine twice. Let the victim be a traveling salesman who left his laptop in a hotel room and goes out for dinner. An “evil maid” can now gain physical access to his target system unsuspiciously. The evil maid replaces the original MBR with a malicious bootloader that performs keylogging and, later on, the unaware salesman boots up his machine and enters his password. On the next event, the evil maid accesses the laptop a second time and reads out the logged passphrase.

As shown by a study [119] on the security of *hardware*-based FDE, so-called Self-Encrypting Drives (SEDs), evil maid attacks are generally *not* defeated by SEDs although these drives encrypt the MBR. The reason is that evil maids can alternatively replace the entire disk, plug in a tiny bootable USB drive, flash the BIOS image [138, 94] or UEFI image [108, 11], or even replace the entire machine with an identical model.

2.3.2 Bootkit Attacks

The original evil maid attack was implemented against TrueCrypt in 2009 [91]. Earlier that year, another bootkit, called the *Stoned Bootkit* [128], could circumvent TrueCrypt as well. Until it has been discontinued in 2014, TrueCrypt was vulnerable to these attacks. Microsoft's BitLocker, on the other hand, defeats software keyloggers up to a certain degree as it verifies the integrity of the boot process by means of the Trusted Platform Module (TPM). BitLocker's decryption key can only be derived if the TPM PCRs are consistent with a reference configuration from system setup. Otherwise, users cannot decrypt their data and hopefully become suspicious that somebody manipulated their machine.

However, at the end of 2009, even BitLocker was successfully compromised by bootkit attacks. Türpe et al. [162] practically performed *tamper-and-revert* attacks that (1) tamper with the bootloader for introducing keylogging functionality, (2) let the victim enter his password into a forged text-mode prompt, (3) revert to the original bootloader, and (4) reboot. The victim may wonder about the reboot, but most likely will enter his password again and proceed as usual – unaware of the fact that his password was logged in the meantime.

Attempts for Countermeasures against Bootkit Attacks As shown by tamper-and-revert attacks, TPMs alone are *not* suitable to guarantee a trusted password prompt. Consequently, other countermeasures had been taken into consideration, as well. The first countermeasures were *external bootloaders* such as the Anti-Bootkit Project [52] from 2010. But even if the integrity of external USB bootloaders can be guaranteed as they are never left unattended, this measure is insufficient for several reasons. First, it remains unclear which USB port is preferred for booting. Thus, an attacker could plug in his own, tiny USB flash drive at the back of the machine. Second, BIOS passwords can often be reset by removing the onboard battery. This allows an attacker to reconfigure the boot sequence at his convenience, e.g., to boot from his own MBR. Third, the BIOS or UEFI may be manipulated and display a fake password prompt. And fourth, the entire target machine can simply be replaced with a manipulated model.

Another countermeasure named Anti Evil Maid [92] was introduced in 2011. This project mentions a *mutual authentication* scheme in the context of hard disk encryption for the first time. Only if the boot process behaves with integrity, a secret (user-defined) message can be unsealed with the help of the trusted platform module. This message must be shown to the user before he enters his password. If the secret message cannot be shown, the user is strongly advised against entering his password because the boot process is likely to be compromised.

Possible Attacks against Anti Evil Maid Considering travelers who boot their laptops at public places like airports, conferences and meeting halls, it is unlikely that the authentication message from Anti Evil Maid remains confidential for a long time. It may be present on surveillance cameras, can be obtained by shoulder surfing, and can perfectly be reconstructed as it is simply ASCII text.

Even worse, the authentication scheme of Anti Evil Maid is not secure if we take the confidentiality of the authentication message during usual usage for granted. It is also not secure if we additionally assume that the sealed authentication message is placed on an external USB drive. Anti Evil Maid only raises the number of required physical accesses from two to three, because the tamper-and-revert attack by Türpe et al. [162] against BitLocker can be extended as follows:

1. On the first physical access, the evil maid manipulates the bootloader in a way that it clones the entire USB drive on booting, e.g., into the very last sectors of the hard drive or into another USB drive. Then he lets the victim plug in his USB drive and, after it has been cloned, the bootloader automatically reverts to its original state and reboots. The reboot occurs quickly after the copy procedure, and thus an unaware user may not even recognize any suspicious behavior.

2. On the second physical access, the evil maid boots up his target with the recently cloned USB drive and notes down the appearing authentication message. He builds his own bootloader with this message and overwrites the original bootloader a second time. The remaining attack equals the procedure of tamper-and-revert: The evil maid lets his victim enter the password and after it has been logged, the bootloader automatically reverts to its original state and reboots again. This time, the victim may wonder about the reboot, but most likely he will re-enter his password and proceed as usual.
3. On the third physical access, the evil maid can read out the logged password and thus gain full access to the encrypted data.

If a second password, e.g., a PIN for the TPM, is used instead of an external USB drive, as alternatively proposed by Anti Evil Maid as well, the attack works analogously with a second password logger instead of cloning the USB drive. If, instead of simple text messages, images are used as additionally suggested by Anti Evil Maid, the attack still works because appearing images can be photographed or digitally retrieved through cold boot attacks [81, 82].

2.4 CPU-bound Encryption

CPU-bound encryption is a technique to implement a certain encryption algorithm in a way that the key and any intermediate state of the cipher is stored and processed solely within the CPU package. Typically CPU registers are either permanently or temporarily occupied by relevant secret information needed by the cipher. In particular, no main memory is used which makes retrieving the key by physically obtaining a dump of RAM impossible.

CPU-bound encryption is often used together with Full Disk Encryption (FDE) to provide a cipher which protects the disk encryption key, for example, against cold boot attacks. Consequently, we use CPU-bound encryption for our secure full disk encryption solution Mark (Sect. 4.1). However, FDE is only one application for CPU-bound encryption. In addition, we also use CPU-bound encryption for our software-based memory encryption solutions RamCrypt (Sect. 5.1) and HyperCrypt (Sect. 5.2) to protect the RAM encryption key from being retrieved by physical memory disclosure attacks.

In this section, we present the two CPU-bound encryption schemes that are used within this thesis. TRESOR [118] is a CPU-bound implementation of AES within the Linux kernel and is used by our kernel-based memory encryption solution RamCrypt. TreVisor [120] implements a CPU-bound version of AES on top of the BitVisor hypervisor [142] and is used by our secure full disk encryption solution Mark as well as our hypervisor-based memory encryption solution HyperCrypt.

CPU-bound encryption/decryption prevents cryptographic keys and any intermediate state from ever being stored in RAM [118, 120, 143, 65]. Both implementations, TRESOR and TreVisor, only use CPU registers to store the encryption key as well as any intermediate information like the AES key schedule. In fact, no cache or RAM is used to store any part of the key or any intermediate state of the AES computation. To prevent intermediate states of AES entering RAM due to context switching, TRESOR is executed inside an atomic section in kernel mode. Inside those atomic sections, interrupts are disabled. According to the authors of TRESOR, however, the *interbench* test suite has shown that interrupts are disabled only briefly such that system interactivity and reaction times are not affected. For TreVisor, the AES cipher is executed inside an atomic section within the hypervisor, i.e., interrupts are disabled and the operating system is paused while the cipher runs on a particular CPU.

TRESOR and TreVisor store the encryption key inside the four x86 breakpoint registers and prevent access to these registers for any other purpose than AES at all times. These four breakpoint registers are normally used to set up hardware breakpoints. Debuggers like GDB, however, use software breakpoints by default, meaning that they set breakpoints by overwriting code with a special instruction which has the same effect in the majority of cases. TreVisor prevents access to these registers by intercepting every access from the guest within the hypervisor. For write accesses, the value to be set is copied to a shadow area in memory and for read accesses, this value is returned back. To the guest operating system it seems the registers are functional, but the processor is not able to use real hardware breakpoints as the registers are not set to the value stored within the shadow area. Because most debuggers use software breakpoints by default, this is mostly a limitation for watchpoints.

If an input block has to be encrypted, the AES computation is done within the SSE registers of the x86 architecture [153] inside an atomic section. Before the atomic section is left, all SSE and general purpose registers are cleared and thus, any critical state of AES is wiped. As the debug registers are only accessible from kernel space or the hypervisor, no user mode program is able to read out or manipulate the key. Attackers who gained kernel or hypervisor privileges, however, can do so. To thwart attacks against intermediate states of AES, TRESOR and TreVisor recalculate the key schedule for every input block which is, however, acceptable in terms of performance due to the use of the AESNI [78] instruction set.

To bootstrap TRESOR or TreVisor, the user has to type a password at an early stage during start-up. The password is used to derive the key which then is stored within the debug registers. We patched TRESOR and TreVisor in a way that also a random key can be generated during start-up. This key is protected against physical attacks like cold boot [81] because debug registers

are cleared after a CPU reset. An attacker would require logical access to a machine and must be able to execute kernel or hypervisor code in order to read out the key.

There exists also a CPU-bound encryption solution for ARM-based devices developed by the author of this thesis called ARMORED [65]. ARMORED is a CPU-bound implementation of the AES cipher similar to TRESOR, but uses the ARM NEON instruction set instead of the SSE registers as intermediate storage area. It was primarily developed to provide the ability to use a cold boot resistant cipher for full disk encryption on recent smartphones, but could also be leveraged against physical memory disclosure attacks and, for example, be combined with RamCrypt.

3 Designing Trusted Computing Architectures

Although many software protection techniques such as crypters and packers were pioneered by malware, there always have been *legitimate* interests in hiding the internals of a program, such as the protection of intellectual property of software creators, hardening software against vulnerability detection, and protecting cryptographic keys of communication protocols. The possibilities to reverse engineer software, however, are steadily increasing as witnessed, for example, by the famous IDA Pro disassembler that fully supports full-scale Intel and AMD assembly as well as the instruction sets of embedded devices such as the TI MSP430 on which Soteria is based on.

Hence, software protection remains a problem of large significance today, particularly for embedded devices, because attacks against them have a clear economic motivation. Potential attacks against smart meters for electricity and heat, for example, range from end-user customers who want to tamper with the amount of consumed energy reported back to their supplier, to competing industrial entities who want to analyze a piece of software for the purpose of re-engineering [37].

In this chapter, we focus on the design of trusted computing architectures for embedded devices. We present two architectures, namely Soteria and Atlas, which are both designed against root-level attackers, i.e., we consider the whole software stack being untrusted, and mainly focus on the code and data confidentiality of an application. While there are trusted computing architectures which also cover code confidentiality, they are usually designed for general purpose devices and rather heavyweight. In contrast to those architectures, our proposed architectures are lightweight and only rely on symmetric cryptographic primitives.

The main application scenario for our architectures are platforms where multiple vendors want to deploy software but are mutually distrusting each other. Those vendors, however, have equal privileges regarding the platform, i.e., either there is no privileged operating system at all and the different software modules of those vendors are cooperatively calling each other or the privileged operating system is untrusted by each vendor as well. In the first case, we want to make sure that the confidentiality of code and data can be maintained even if a malicious module tries to read or write the memory of another module. In the second case, we in addition need to ensure that a potentially compromised or malicious operating system is not able to extract code and data from a less privileged software module.

Because all software, potentially including an operating system, is considered untrusted, our trust is rooted in hardware. Our first architecture, Soteria which is described in Sect. 3.1, is an extension of the trusted computing architecture Sancus (Sect. 2.2.1) and thus, separates software modules with the help of boundary registers. The number of available boundary registers, however, is fixed at synthesis time and one pair of boundary registers is needed per software module. Consequently, while Soteria is well suited for small constraint applications like smart meters or sensor nodes, it does not scale to applications that need a large number of protected software modules.

This is why we propose our second architecture, Atlas which is presented in Sect. 3.2. Instead of relying on boundary registers for the separation of software modules, Atlas protects modules with the help of an encryption unit placed between cache and main memory. While Atlas alone cannot guarantee integrity, it is meant to be combined with a traditional Memory Protection Unit (MPU) which usually guarantees confidentiality and integrity. In the case of a compromised operating system, the MPU can no longer be trusted, but Atlas can still guarantee the confidentiality of all protected software modules.

3.1 Offline Software Protection within Low-cost Embedded Devices

Protecting the intellectual property of software that is distributed to third-party devices which are not under full control of the software author is a difficult problem. Modern techniques of reverse engineering such as static and dynamic program analysis with system privileges are increasingly powerful, and despite possibilities of encryption, software eventually needs to be processed in clear by the CPU. To anyhow be able to protect software on small embedded devices, a part of the hardware must be considered trusted.

We present a lightweight software protection scheme that is designed for low-cost embedded devices such as the TI MSP430. Our solution is called "Soteria" after the *ancient Greek personification of safety, preservation and deliverance from harm.* At its heart, Soteria is a program-counter based memory access control extension for the TI MSP430 microprocessor. Based on our open implementation of Soteria as an openMSP430 extension, and our FPGA-based evaluation, we show that the proposed solution has a minimal performance, size and cost overhead while effectively protecting the confidentiality and integrity of an application's code against all kinds of software attacks including attacks from the system level. Soteria builds upon Sancus [124] which serves as the basis for our work regarding its zero-software TCB for attestation and integrity checking.

Contribution Our contribution is the design and implementation of a system that provides offline software protection for low-cost embedded systems as an addition to the existing remote attestation and integrity checking capabilities of Sancus [124]. In detail, our contributions are as follows:

- While Sancus currently provides the *integrity* and *authenticity* of software modules, we add the capability to also guarantee *confidentiality.*
- We designed a scheme consisting of *software modules* and *loader modules.* The confidentiality of code and data of the protected software modules is guaranteed at any given point in time against all software attackers, including those who could gain system privileges. The loader modules themselves can be written in software and are not part of the minimal trusted computing base implemented in hardware. For bootstrapping reasons, however, the code of the loader modules is unprotected against reverse engineering.
- Based on hardware supported integrity checks, loader modules can decrypt a protected software module only if the integrity of both is not violated. The key for decryption is derived directly on the target system and hence, is not loaded from a trusted party during runtime. In particular, our solution does not have to attest its trusted execution environment to a remote party but bootstraps autonomously. In other words, Soteria is a solution for *offline* software protection.
- We implemented our approach by patching the openMSP430 core from OpenCores and provide a modified toolchain including the compiler and linker. For example, to get RAM executable and to disallow read access between different modules, we mainly modified its memory access logic. According to our design, Soteria software modules are fully backwards compatible with the original Sancus environment.
- Based on our FPGA-based evaluation, we show that Soteria has practically no runtime overhead but only a small loadtime overhead. We also show that Soteria's costs in terms of size and power consumption are minimal. For example, the power consumption raised by only 0.2% when compared to Sancus.

Since we built upon Sancus, we refer the reader to the description of Sancus within the background chapter of this thesis (Sect. 2.2.1). It provides a brief overview of Sancus including its design goals and security properties. For more detailed information about Sancus, please refer to its original publication [124].

3.1.1 Attacker Model

In our model, we assume that an attacker wishes to violate confidentiality of the code and data of an arbitrary *SM*. To achieve this, an attacker can mount all kinds of *software* attacks but *no hardware* attacks. To be more precise, an attacker is allowed to control all peripheral components, including tampering with the communication to other devices. Furthermore, each piece of software, also privileged software like an operating system is allowed to be under the control of attackers, because access to *SM*s is solely prevented by hardware.

However, hardware attacks are excluded from our attacker model, in particular attacks like RAM dumping, chip probing and fault injection are excluded. If hardware attacks are considered at all, only ROM dumping is allowed as it does not harm our solution. Also note that, as the attacker is able to control privileged software, he can easily restrict availability of the overall system. Denial-of-Service (DoS) attacks are excluded from our attacker model as they have no impact on the confidentiality of code and data.

3.1.2 Soteria Architecture

In this section, the architecture of Soteria is explained in detail. Our contribution is maintaining the confidentiality of code and data in a low-cost embedded system without the need to trust any software component.

For the Sancus-based design of Soteria we thought of two possible concepts: (1) Putting a loader stub together with the code that should be encrypted into one module and decrypting it in-place, and (2) designing a separate loader module. In the first case, the loader stub has to reside within RAM in addition to or instead of ROM and at least parts of the loader code would be duplicated for each encrypted module wasting space. Therefore, we decided to use the second concept, i.e., to design a separate loader module. With this concept, a new loader software module SM_L provided by SP_L is responsible for decrypting and protecting another module SM_E provided by SP_E. The decryption key for SM_E is derived from the loader key K_{N,SP_L,SM_L} and the unique identifier $\widetilde{SM_E}$ of SM_E which consists of the name and the current version of SM_E. The decryption and protection of SM_E within SM_L must run atomically to guarantee the confidentiality of code and data of SM_E at any point in time.

Soteria maintains the confidentiality of code and data based on a zero-software TCB with two different mechanisms. First, before loading an encrypted module, the code resides encrypted within ROM or RAM such that no other module is able to read it. Second, after an encrypted module has been loaded, a program-counter based memory access logic ensures that no other module can access code or data of the decrypted module. The remainder of this section gives an overview of how loading encrypted modules works.

In Figure 3.1 and Figure 3.2, the loading process of a module within Soteria is illustrated: At first, only the loader is protected and active. Then, SM_L derives E_{SM_E}. Next, SM_E is checked for integrity, gets decrypted and protected, and finally SM_L is able to unprotect itself. The detailed loading process is as follows:

1. SM_L is loaded like an ordinary Sancus module and typically has a code section within ROM and a data section within RAM.
2. If SM_L is instructed to load another encrypted module, it first derives the decryption key E_{SM_E} from its own module key K_{N,SP_L,SM_L} and the unique identifier $\widetilde{SM_E}$ of the encrypted module SM_E it is about to decrypt:

$$E_{SM_E} := K_{N,SP_L,SM_L,SM_E} = kdf\left(K_{N,SP_L,SM_L}, \widetilde{SM_E}\right)$$

3. The module SM_E is decrypted and checked for integrity simultaneously by using authenticated decryption with the key E_{SM_E}. If the integrity property is violated, all intermediate data is wiped and the loading process is aborted.

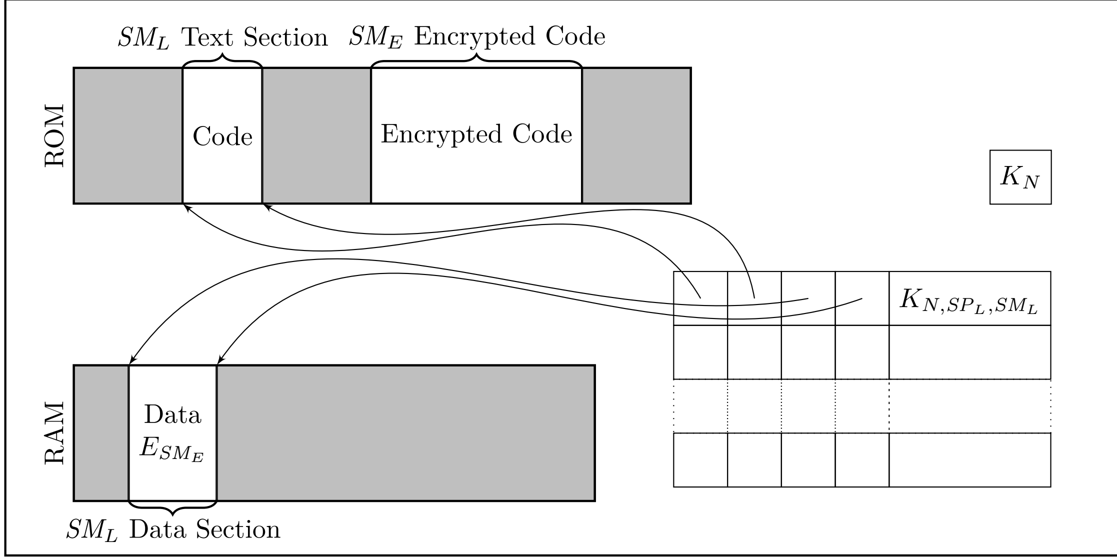

Figure 3.1: Loading Steps 1+2: SM_L is active, protected, and has derived E_{SM_E}.

4. The module SM_E gets protected and implicitly derives the key K_{N,SP_E,SM_E}.
5. The loading process is finished and SM_L is now able to load the next encrypted module or to unprotect itself.

Note that steps 3 and 4 need to be performed atomically by SM_L. Hence, it has to be ensured that no other module runs between these three steps, e.g., by disabling interrupts globally. All other steps do not need to be performed atomically as E_{SM_E} is securely stored within the protected data section of SM_L and therefore cannot be read out by any other module even if it runs while E_{SM_E} has already been derived. For encryption and integrity checking, we use AES-128 [55] in CCM mode of operation [57] which provides authenticated encryption, i.e., encryption and integrity checking is done at the same time with the same key. We chose CCM over other authenticated encryption modes such as GCM [50] or OCB [136] because of its simplicity and the fact that CCM is not patent encumbered. Another advantage of CCM is that only the encryption routine of AES-128 is needed which is more efficient in terms of runtime compared to the decryption routine. For the detailed CCM parameters see Sect. 3.1.6.

Although the code for the loader and the encrypted code for the module usually reside within ROM, it is possible to place it in RAM as well. In any case, it is cryptographically guaranteed that no tampered module is loaded, and that the valid module is never decrypted by a tampered loader. For more details, see Sect. 3.1.5.

3.1.3 System Deployment

Sancus specifies an Infrastructure Provider (*IP*), Software Provider (*SP*) and Software Modules (*SM*s). In practice, a software provider SP_j receives K_{N_i,SP_j} once from the *IP* for each node N_i that should run software of SP_j. For each $SM_{j,k}$, the software provider can then derive $K_{N_i,SP_j,SM_{j,k}}$ on its own, which can without the knowledge of K_{N_i,SP_j} only be derived from node N_i in hardware. Thus, for each $SM_{j,k}$ a secret shared key $K_{N_i,SP_j,SM_{j,k}}$ between N_i and SP_j exists which is used to prove its integrity and authenticity.

To the above scenario, Soteria adds SP_L, the software provider of the loader, and SM_L, the loader module which is responsible for loading encrypted modules $SM_{E,1}, \ldots, SM_{E,n}$, as shown in Figure 3.1. For the deployment of our system there are basically two possibilities: (1) $SP_L = SP_E$, i.e., the software provider of the encrypted modules is also the software provider of the loader

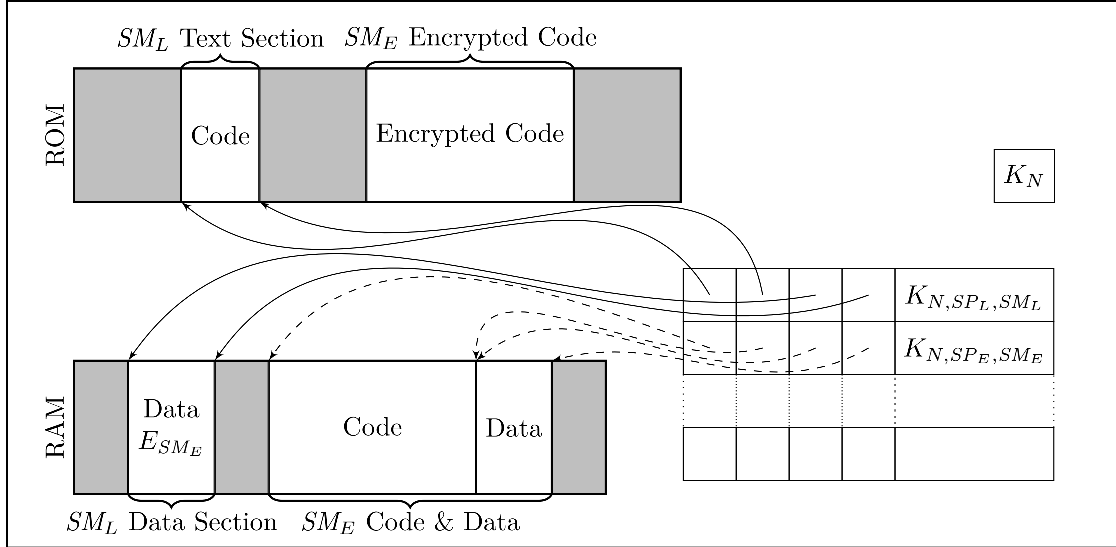

Figure 3.2: Situation after Step 4: SM_E is decrypted, protected, and has derived K_{N,SP_E,SM_E}.

module, and (2) $SP_L \neq SP_E$, i.e., the software provider of the encrypted modules is different from the software provider of the loader.

Encrypting modules, i.e., preparing the ROM image for a target device or preparing software updates can be done on the host of SP_E with knowledge of a previously exchanged encryption key, but has to be done differently in both cases. The key exchange is trivial for (1), because the software provider has direct access to the key K_{N,SP_L,SM_L} and can therefore derive $E_{SM_{E,1}}, \dots, E_{SM_{E,n}}$ (Sect. 3.1.2) and encrypt the modules $SM_{E,1}, \dots, SM_{E,n}$. For case (2), however, SP_L and SP_E have to cooperate for the key management, meaning that SP_E has to get $E_{SM_{E,k}}$ for each module $SM_{E,k}$ by telling SP_L the unique identifier $\widetilde{SM_{E,k}}$ of $SM_{E,k}$. Although SP_E does not need to give code to SP_L but only the unique identifier of the module, SP_E needs to trust SP_L. In particular, SP_E needs to rely on the fact that SP_L only provides a loader for the module on a specific target device but keeps the derived decryption key secret.

Similar to *IP* being the trusted party for remote attestation, SP_L is the trusted party regarding confidentiality. If only confidentiality is needed but no attestation is required, it is sufficient for SP_E to communicate with SP_L if $SP_L \neq SP_E$. Note that *IP*, which is the root of trust in the overall system, is of course able to violate confidentiality at any time. This is not a drawback to our proposed system, as the hardware is considered trusted and all parties need to trust *IP* as in Sancus.

3.1.4 Access Control

Assuming an encrypted module SM_E has been loaded correctly, its code now resides within RAM and needs to be protected by the program-counter based memory access logic. In theory, reading the text section only has to be disabled for modules that were encrypted, as the others are stored in cleartext within ROM, but we decided to disable read access from anywhere for all modules.

Table 3.1 shows the different access rights for the whole address space. The only part of a module that remains readable and executable is the entry point. All other areas of a module, i.e., the text section and the data or protected section, are not read- or writeable from any other module or unprotected code, which is ensured by adding new checks to the memory access logic of Sancus. There is a new kind of memory violation that is triggered if another module or unprotected code tries to read the text section of a module. All violation signals are combined into a single signal that resets the processor immediately when triggered. Although it would be possible to return

From/To	Entry	Text	Protected	Unprotected
Entry	r-x	r-x	rw-	rwx
Text	r-x	r-x	rw-	rwx
Unprotected/ Other SM	r-x	---	---	rwx

Table 3.1: Access rights from/to a module with prohibited read access to code [124].

NULL on read violations and to ignore write violations, there is no other reasonable choice for handling execute violations than triggering a reset because system level attacks are included in our attacker model and thus no appropriate handler routine can be called. Furthermore, we cannot fully guarantee availability due to attacks from the system level such that not triggering a reset would not improve availability.

In addition to an extension of the memory access logic, we take care that RAM is wiped completely in hardware after a reset has been triggered. Consequently, no fragments of code or data of a previously encrypted module can be found after a reset. This is particularly useful if a violation is triggered, because a malicious module could otherwise trigger a reset intentionally, and, when the module is reloaded after the reset, read out portions of RAM which still contain cleartext of encrypted modules.

3.1.5 Security Properties

After explaining the architecture of Soteria in the previous sections, we briefly describe its security properties now. In detail, Soteria can guarantee the following security properties:

- *Isolation*: Every module is completely isolated from other modules regardless of their privilege level. No other module or unprotected code can read or write from or to the code and data section of a given module. This property is partly inherited from Sancus, but was extended by the capability to reject read-access to the code section of a module.
- *Remote Attestation*: A remote party can cryptographically verify that a specific module has been loaded on a given device with a previously defined state. Based on remote attestation, tamperproof communication is possible as well, because messages can be combined with authenticity and integrity information before they are sent. This property is most widely inherited from Sancus, but the host tools for HMAC generation had to be modified slightly.
- *Secure Linking*: A software module on a given node can call a function of another module on the same node while guaranteeing that it is calling the intended function of the intended destination module. This property is inherited from Sancus.
- *Confidentiality*: Confidentiality of code and data for encrypted modules can be guaranteed at any given point in time, i.e., before modules are loaded as well as afterwards. Furthermore, confidentiality can be ensured *offline* due to mutual integrity checks between the loader and the module that is about to be decrypted, without the need for remote attestation. This property is not present in Sancus but a novel contribution of Soteria.
- *Integrity*: While for Sancus the software provider is able to ensure the authenticity and integrity of a module through remote attestation only after communicating with it, Soteria is able to guarantee the integrity of encrypted modules *offline*, meaning that manipulations are already detected at load time rather than communication time. This property is a novel contribution of Soteria.

The confidentiality property is guaranteed by two mechanisms. Before load time, modules are encrypted and thus considered confidential. After load time, modules are protected by the program-counter based memory access logic and are therefore considered confidential as well. The only possibility of attackers to violate the confidentiality is by compromising the loading process, which

is prevented by our design as follows: If the loading position or protected sections of the loader are tampered with when the module SM_E is about to be loaded, the key E_{SM_E} is derived incorrectly because K_{N,SP_L,SM_L} is derived wrongly. The authenticated decryption then fails and the loading process immediately aborts. If the encrypted module SM_E is tampered with before loading, the authenticated decryption fails as well and loading aborts again. Tampering is not possible while the loading takes place because authenticated decryption and protecting SM_E is performed atomically.

In summary, confidentiality is always preserved and any attempts to tamper with encrypted modules are detected immediately. The only property an attacker is able to affect is availability, but as explained in Sect. 3.1.1, we exclude denial of service from our attacker model. Our solution provides no DoS protection and DoS is out of scope for Soteria. For example, an attacker could repeatedly trigger resets or monopolize the CPU with an endless loop to impact availability.

3.1.6 Hardware Implementation

We now describe the implementation details of Soteria. First, we describe the changes we made to the hardware in order to guarantee the security properties mentioned in Sect. 3.1.5. Second, in Sect. 3.1.7, we describe the software we developed for the target device, i.e., for the modified openMSP430. Finally, in Sect. 3.1.8, the toolchain that is necessary to deploy our system is explained. All software belonging to the toolchain runs on the host, meaning that it does not increase the trusted computing base.

Our implementation is based on the openMSP430 [62], an open-source implementation of the MSP430 from Texas Instruments [152] in Verilog provided by OpenCores. Sancus is built on the openMSP430 and was the starting point of our implementation. The MSP430 is a 16-bit processor with a von-Neumann architecture. It has support for 8-bit and 16-bit peripheral components, such as a timer, UART, and GPIO. During our development, we simulated the modified openMSP430 using Icarus Verilog directly on the host system. For the evaluations, we ported our implementation to an FPGA and used the TimerA component, UART, and GPIO. For more information on the evaluation, see Sect. 3.1.9.

First of all, we introduced a new violation that is triggered if a module tries to read the code section of any other module. The address of the memory access is checked against the boundaries of the code section of every protected module and is only allowed if the program-counter is within the code section of the module currently being executed. Otherwise a violation is triggered, which is OR'ed with the other memory access violations.

While implementing our concept, we came across a fundamental limitation of the openMSP430. Although it has a von-Neumann architecture, it distinguishes between ROM and RAM by mapping them to specific locations within the 16-bit address space. As a result, it turned out that RAM was not executable, which is also mentioned as a drawback on OpenCores. To support our proposed design, we need to decrypt a module to RAM, protect and execute it. We therefore needed to patch the openMSP430 to make RAM executable. Our current implementation of the openMSP430 has executable RAM and the ability to switch execution freely between code from any region, so that unprotected code that resides, for example, within ROM, can directly jump to previously encrypted modules which reside within RAM after decryption.

To prevent the exploitation of deliberately triggered resets (Sect. 3.1.4), we developed a memory wiping component directly in hardware. The component is inserted between the openMSP430 itself and the RAM. When a reset is triggered, it starts wiping the RAM word by word for all 16-bit words in memory. While this takes place, the memory wiping component holds the reset line, simulating to the openMSP430 that the reset is still not released. The openMSP430 therefore remains paused while the entire memory is being zeroed out from start to end. After wiping is finished, the reset line is released and the openMSP430 dispatches the first instruction which is referenced by the reset vector. This procedure ensures that RAM is completely wiped before the very first instruction is dispatched after a reset.

3.1.7 Software Implementation

In addition to hardware modifications, we have written software that runs on the target device, namely the openMSP430 processor. As we want to protect third-party software, we needed to provide two components: (1) a library which supports encrypted modules while being fully compatible with non-encrypted legacy modules of Sancus, and (2) the loader module itself, which is capable of integrity checking, decrypting and protecting an encrypted module.

In Sancus, modules are represented by a structure consisting of a unique ID, the software provider ID, the module name and the boundaries for the code and data section. This structure contains all values necessary for traditional modules to be loaded, and the boundaries are those that are considered by the memory access logic at module runtime. For Soteria, we extended the structure with new boundaries for the code section of the encrypted module, i.e., the boundaries are valid only for the encrypted code but not for the code section of a running module. In a typical scenario, the boundaries for the encrypted code point to ROM, while all others, i.e., those for the code and the data section, point to RAM. The authentication tag for encrypted modules is stored directly after the encrypted code section such that the upper boundary for the encrypted code section points directly to the integrity information. Our new library is fully compatible with existing non-encrypted legacy modules. On the one hand, new encrypted modules can of course be used in conjunction with our loader, but on the other hand, they can also be directly passed to existing functions of the old Sancus library. Our library also includes a function to destroy a module. When it is called, all the module's protected data and code within RAM is first wiped and then the `unprotect` function of the Sancus library is invoked. It is basically a convenience routine to allow the programmer to destroy a module with only a single call.

We also provide an open implementation of the loader for the openMSP430 such that it can be used by every software provider SP_L. The loader accepts encrypted modules and basically provides two functions to the user: The load routine which takes an encrypted module, derives the decryption key for the destination module, performs integrity checking, decrypts the module to RAM and finally protects the just decrypted module. As already explained, these steps have to be performed atomically. The second routine that is provided by the loader is a routine which causes the loader to wipe its protected data section and afterwards unprotect itself. This routine is used to destroy the loader.

The key derivation is done in hardware, as explained in Sect. 3.1.2, using the HMAC functionality provided by Sancus which is based on *Spongent-128* [24]. To this end, the unique identifier $\widetilde{SM_E}$ which is a concatenation of the module name and the module version number is passed to the HMAC function which implicitly uses K_{N,SP_L,SM_L} as key.

The decryption is implemented in software using AES-128 [55] in CCM mode of operation [57]. We implemented the CCM mode according to RFC 3610 [166] with an authentication tag length of sixteen bytes, a two byte length field and no associated data, i.e., data that just needs to be authenticated. With this choice we are able to decrypt software modules with a maximum length of 64 kilobytes which is also the maximum addressable size on the openMSP430 processor. In addition to the decryption key and the encrypted module, CCM with the given parameters also requires a thirteen byte nonce which does not need to be secret but must be different for each CCM en- or decryption with the same key. We simply use the unique identifier $\widetilde{SM_E}$ (padded with zeros) as nonce because it is changed for every new version of the module SM_E and thus satisfies the nonce properties. If the module name concatenated with the module version number exceeds thirteen bytes, a cryptographic hash of $\widetilde{SM_E}$ is calculated and truncated to thirteen bytes. The calculation of the hash then only needs to be done by the toolchain and not on the target device because the nonce can securely be stored along with the authentication tag.

We built our implementation onto the *tinyAES* implementation [101] combined with the CCM implementation of mbed TLS [14]. Consequently, our authenticated decryption routine is implemented entirely in C and it has been optimized for size. The overall size of the key schedule, the encryption routine and the CCM implementation is about two kilobytes. Memory protection is enabled by calling the special instruction `protect`.

3.1.8 Toolchain Implementation

Aside from hardware and target device software, we had to implement toolchain components in order to successfully deploy our system. The whole toolchain is based on LLVM [105], msp430-gcc [151] and pyelftools [20]. The toolchain consists of three Python tools that need to be called after successfully compiling different source files with LLVM:

1. `ld`: The linker which produces a single ELF file from the different object files and generates separate sections for each software module. Two sections are generated for encrypted software modules. One is placed in the RAM address range and contains all cleartext code like it would be after the decryption during the loading process. The other is placed in the ROM address range and is 16 bytes larger than the section within RAM to reserve space for the integrity information that is produced by the CCM authenticated encryption. This second section is just filled with zeros by the linker and will later be overwritten by our encryption script.
2. `hmac`: This script, which was inherited from Sancus and only needed to be modified slightly, adds HMACs to the ELF file for secure linking.
3. `crypt`: This script performs authenticated encryption of the cleartext code and places the result within the section in the ROM address range. It implicitly computes integrity information in CCM mode and stores it directly after the encrypted code. To this end, the script has to derive the encryption key from the key of the loader and the unique identifier of the encrypted module. The derivation is fully automated and only the vendor ID, node key and name of the loader module need to be supplied as arguments. After the code has been encrypted, the original section within RAM is stripped from the ELF file so that the resulting image can be flashed to the target device or distributed securely, as it no longer contains any unencrypted data.

All described scripts work directly on ELF files to avoid using different intermediate formats. The toolchain is easy to use because very few information needs to be provided. Modules that need to be encrypted are identified by their name: If a module starts with `crypt_`, the scripts transparently add new sections for this module, encrypt it, add integrity information and reserve space in RAM to store the plaintext at runtime. Of course, all components of the toolchain run on the host instead of the target device, are therefore not part of the device trusted computing base, and consequently do not consume space on the target device.

3.1.9 Performance, Area, and Power Analysis

In this section, we evaluate Soteria regarding its performance, impact on chip size and power consumption. All data in this section was produced using a Xilinx XC6VLX240T Virtex-6 FPGA, with the core running at 20 MHz.

Performance When evaluating the performance of Soteria, we have to distinguish between the runtime overhead and the time a module needs to be loaded. First of all, there is a constant overhead for resetting the processor as all data memory is wiped in hardware before execution resumes. Since the MSP430 has no caches and memory can be written directly, wiping takes exactly $2 + \text{DMEM_SIZE}/2$ cycles, because it is wiped by writing successive 16-bit words which corresponds to two bytes. Our reference configuration, for example, has a program memory of 48 kilobytes and a data memory of 10 kilobytes, so 5,122 cycles are needed for wiping.

Once an application is running, our solution imposes no additional overhead compared to plain Sancus, meaning that the additional program-counter based memory access checks do not have a performance impact on the critical path. During our experiments, we also verified, using the TimerA component of the MSP430, that routines of modules built with our solution execute in the same amount of cycles as those built with plain Sancus. This applies to unencrypted legacy modules as well as to encrypted modules after they have been loaded.

Size	Key Derivation	Authenticated Decryption	Protection	Total
208	13,504 (0.675)	383,344 (19.167)	26,984 (1.349)	424,312 (21.216)
256	13,504 (0.675)	463,088 (23.154)	30,464 (1.523)	507,536 (25.377)
512	13,504 (0.675)	888,456 (44.423)	49,024 (2.451)	951,464 (47.573)
768	13,504 (0.675)	1,313,816 (65.691)	67,584 (3.379)	1,395,384 (69.769)
1024	13,504 (0.675)	1,739,176 (86.959)	86,144 (4.307)	1,839,304 (91.965)

Table 3.2: Number of cycles (ms) needed for loading modules with different sizes.

The main performance overhead of our solution is incurred by loading of an encrypted module. Protecting the loader imposes a constant overhead on all encrypted modules combined. This process takes 72,976 cycles, while destroying the loader (i.e., freeing all resources, wiping keys and calling `unprotect`) needs only 800 cycles. The loading overhead for an encrypted module depends on the size of the module and is dominated by the time needed for the authenticated decryption routine. In Table 3.2, the number of cycles needed to load modules with different sizes are shown. First, the key for authenticated decryption of the encrypted module is derived. This is done in hardware and the number of required cycles is therefore rather small. Furthermore, the unique identifier had the same length for all modules tested in Table 3.2 and thus the number of cycles needed for the key derivation is independent from the size of the encrypted module. The module then has to be decrypted and checked for integrity, which takes up most of the cycles because authenticated decryption is done in software using AES-128 in CCM mode of operation. Finally, the decrypted module needs to be protected to deny other modules access to the decrypted code. The protection process is dominated by the key derivation for the new module and therefore the execution time again depends on the size of the module that is about to be protected. Note that the total number of cycles to load an encrypted module is more than just the sum of key derivation, authenticated decryption and protection, because of additional boundary checks and a length calculation of the unique identifier. These additional computations, however, only depend on the unique identifier of the encrypted module and as it was identical for all of our tests, the additional overhead is constant.

The smallest module that was evaluated had a size of only 208 bytes. This is the smallest size possible for a module with at least one entry point and no additional data. The minimum size results from the stubs that are included by the Sancus compiler to be able to call the entry point. Other module sizes that have been tested range from 256 bytes up to one kilobyte. Considering the clock rate of 20 MHz, we need about 92 milliseconds to load an encrypted module of one kilobyte and 25 milliseconds for an encrypted module of 256 bytes size. While this might be some overhead, the primary reason for which is running AES in software, one has to consider that modules are only loaded when the system is started. Since no runtime overhead is imposed on the modules, short loading times below 100 milliseconds seem acceptable. Furthermore, minimum code size instead of maximum loading performance was the main focus. The implementation of AES-128 in CCM mode of operation only needs about 2 kilobytes of ROM and circa 200 bytes of RAM.

To be able to produce these performance evaluations, a special version of Soteria was synthesized which does not trigger a reset if a memory violation occurs because interrupts, for example, can trigger an execute violation. The TimerA component of the MSP430 was used with the main system clock of 20 MHz in divide-by-eight mode. As this component only supports a 16-bit counter, it overflows for large measurements and an additional counter for the higher 16 bits needed to be implemented in software. This counter is incremented in the interrupt handler called for the overflow. Although only a single operation, i.e., the increment of the counter for the higher 16 bits, is executed within the interrupt handler, measurements which take more than 65,536 cycles are not perfectly cycle accurate, but instead a higher number of cycles is measured than what is actually needed for the pure computation. This is due to the fact that entering and leaving the interrupt handler takes cycles, and those cycles are included in our measurements. Consequently, we give upper bounds for the actual performance results which are expected to be even less than those shown in Table 3.2.

	Sancus		Soteria		Overhead	
	REGs	LUTs	REGs	LUTs	REGs	LUTs
1 SM	1,897	3,686	1,938	3,894	41	208
2 SMs	2,110	4,100	2,150	4,322	40	222
3 SMs	2,323	4,378	2,363	4,620	40	242
4 SMs	2,536	4,778	2,576	5,034	40	256

Table 3.3: Number of slice registers and slice LUTs for Soteria compared to Sancus.

Area We measured the area overhead of Soteria by considering the required slice registers and slice LUTs. The plain openMSP430 core needs 1,146 slice registers and 2,520 LUTs when synthesized for our FPGA. In Table 3.3, the number of slice registers and slice LUTs for Soteria in comparison to plain Sancus are shown in different configurations, i.e., for a specific number of supported modules. The slice register overhead Soteria incurs over Sancus is almost constant and equal to about 40. The number of additional slice LUTs depends on how many modules are supported, but it is small compared to the overall number of slice LUTs. The main overhead comes from the memory wiping logic and the additional read access prevention which is added for each module. Since the decryption routines are implemented in software rather than hardware, our solution imposes very little area overhead.

Power We have analyzed the power consumption with the static power analysis tool *Xilinx XPower Analyzer*. Soteria had an overhead of about 0.2% compared to Sancus, regardless of the number of supported modules. We also tried to measure the power consumption experimentally, but could not find any difference between Soteria and Sancus running on our FPGA. The overall power consumption of the FPGA reported by *Xilinx XPower Analyzer* was about 3.537 W for Soteria and about 3.530 W for Sancus.

3.1.10 Discussion

Soteria has been designed to protect the intellectual property of code and data against root-level attackers, including those who could gain system privileges. Our attacker model, however, excludes all kinds of hardware attacks such that attackers with physical access are potentially able to acquire sensitive data while circumventing the protection of Soteria, e.g., by chip probing or fault injection.

The primary performance bottleneck of Soteria is the decryption routine which is implemented in pure software. If special hardware instructions are available, cryptographic primitives can be considerably accelerated like we have shown in previous work [66]. The openMSP430, however, does not offer such instructions as it is a small embedded core which aims to be lightweight. Consequently, those software acceleration techniques cannot be applied to Soteria.

The ideas from Soteria and other improvements have been backported to Sancus and are available to the public as a new version Sancus 2.0 [125]. In Sancus 2.0, the encryption and decryption routines are implemented in hardware and thus, run considerably faster. Also, since the program-counter based memory access logic prevents all jumps into module code different to the entry point from other modules or unprotected code, Sancus and Soteria do not support interrupts. Sancus 2.0, however, has been leveraged for interrupt handling by adopting separately developed interrupt handling proposals [44].

3.2 Application Confidentiality in Compromised Embedded Systems

Due to their increased complexity, modern embedded systems run many different applications by potentially different vendors. We focus on protecting the confidentiality of code and data against system-level attackers through transparent memory encryption. Our solution is designed to be complementary to traditional MPUs, which rely on the OS to be configured. However, when the OS has been compromised, security, and especially confidentiality of code and data, can no longer be guaranteed.

With our solution, called Atlas, we ensure confidentiality even in the event of a system compromise. Atlas relies on its zero-software TCB to protect against system-level attackers. In addition, compared to other trusted computing mechanisms for lightweight processors, e.g., based on boundary registers like Sancus [124] or Soteria (Sect. 3.1), Atlas has lower area overhead, which is fixed for any number of applications.

Contribution We introduce Atlas, a hardware-based security mechanism protecting application confidentiality against system-level attackers, with a fixed overhead that is independent of the number of applications running on the system. Furthermore, Atlas enables the use of shared memory as a lightweight and easy-to-use secure communication channel. In detail, our contributions are:

- We propose the use of hardware-based memory encryption to protect application confidentiality in embedded systems. In particular, our solution protects confidentiality in the event of system compromise, including a potentially compromised OS.
- We ensure that neither code nor data leaks to any other application or the OS, relying on a zero-software TCB. Since there is no need to keep track of state information per application, our solution scales to an unlimited number of applications.
- We provide confidential shared memory, which can be used as a communication channel between multiple applications, without the need for a dynamic key exchange.
- We designed and implemented Atlas by extending the open source LEON3 processor. This includes a host toolchain to compile C programs for our architecture.
- We evaluated the software and hardware implementation of Atlas regarding performance and area. Atlas has 0.031% cycle overhead compared to an unmodified binary for a real-world signing application, at the cost of a four times slower maximal clock and 46.595% area increase.

3.2.1 Attacker Model

In our model, we assume the attacker wants to extract confidential intellectual property, e.g., proprietary algorithms, from the application's code. Furthermore, he is also looking to obtain confidential data processed by it, which was either statically compiled or dynamically calculated at runtime. The attacker has system-level privileges, i.e., he can exploit any piece of software running on the device, including the OS. As long as the OS has not been compromised, an MPU ensures that applications only access their own memory. When an attacker has obtained system-level privileges, however, he can read from and write to any memory location. DoS attacks are considered to be out of scope. Following the Dolev-Yao model [46], the cryptographic primitives used in our scheme cannot be broken, but protocol-level attacks are allowed.

In addition to controlling any software, the attacker can physically probe main memory. However, we assume he does not have access to the CPU's registers or caches. Invasive attacks where the chip is decapsulated are therefore excluded. This is a reasonable assumption, since such attacks require a high level of technical skill, expensive equipment, and take a long time to plan and execute. For example, Tarnovsky's attack on the Infineon SLE 66 microcontroller took six months from planning to execution [150].

3.2.2 System Model

Encrypting memory transparently under a single key is not sufficient to protect against a system-level attacker, as such a system could not track ownership and would return any requested data in plaintext. Therefore, the device's system model has to meet two requirements. First, all calls to any confidential application have to pass through its entry point, and applications therefore need to know each other's location. Second, an application should not be able to relocate itself to the entry point of another confidential application, as this would give it access to that application's confidential code and data. The entry point corresponds to the first instruction being executed when an application is called. Atlas satisfies the first constraint by creating a static layout of all applications running on a single device. Since decryption will fail when an attacker moves his application, and because it is hard for him to generate a correctly encrypted binary himself, the code encryption mitigates the second issue. Note that applications are expected to yield control when finished, as there is no preemption.

In addition to the device key K_D, the current implementation of Atlas also uses a tweak key F (Sect. 3.2.4). Both keys are unique for each device, and generated by the system integrator. They are hardwired in the silicon, e.g., by blowing fuses of the manufactured device.

The secure shared memory feature relies on pre-shared secrets. Because a confidential application's static data is encrypted, the communication keys can be stored securely in memory and decrypted when necessary. Generating these keys, defining the regions where the applications can read and write shared data, and updating the binary with these parameters is also done by the integrator.

3.2.3 Atlas Architecture

Atlas' encryption unit protects the confidentiality of applications sharing the same address space. Once memory protection mechanisms relying on software have been compromised, applications can read from or write to any given address. However, the entry point is used as a unique Initialization Vector (IV), binding dynamically encrypted data to its application. Although a system-level attacker has the ability to read any location, he will be unable to recover the correct plaintext when trying to access protected memory.

When the OS has been compromised, the MPU can no longer be trusted to protect against an attacker modifying memory. As shown in Sect. 3.2.2, code encryption prevents an attacker from relocating himself to another application's entry point. Although an attacker can now write to any memory location, data encryption cannot be configured independently and thus, code needs to be encrypted as well. This increases the attack complexity, as any instruction manipulating memory needs to be encrypted. Since the attacker does not know the encryption key, it is hard for him to obtain the instruction's ciphertext. Consequently, Atlas protects the confidentiality of code and data against all software attacks including relocation attacks.

Encryption Unit Properties The encryption unit is considered to have the following properties: First, in order to protect the confidentiality of different applications, it is able to identify the application to which the current memory bus request belongs. Second, as one of the design goals is to build a scalable architecture, it has to be stateless. Finally, to support secure shared memory, it should be possible to dynamically reconfigure the symmetric key used for data encryption to one that is shared among the communicating applications.

Hardware Architecture The encryption unit is inserted between the cache and main memory. Once it is turned on, confidential instructions will be automatically decrypted when read, and data will be decrypted and encrypted transparently when entering or leaving the cache. Remember that it is assumed to be impossible for attackers to read the processor's caches or internal registers (Sect. 3.2.1). To prevent leakage, our hardware and toolchain respectively take care of flushing both caches, and clearing all registers when the encryption unit mode is changed.

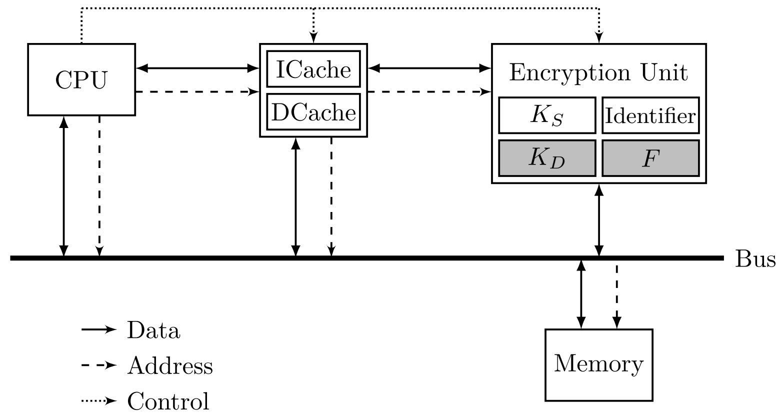

Figure 3.3: Encryption unit which has been added between cache and main memory.

The encryption unit is controlled through custom instructions that were added to the Instruction Set Architecture (ISA). They can be used to turn encryption on or off, e.g., when an application does not need confidentiality or in case it wants to access unprotected memory. Additional instructions are available to configure and use secure shared memory.

Software Architecture In order to decrypt encrypted code and dynamically protect data, the currently executing application has to be identifiable. An entry point is therefore created for each application, which is the very first instruction that has to be called when execution of an application is started, and takes care of setting the application's identity and switching the encryption context. Since all local and global functions are encrypted, as well as its static data, they will not be decrypted correctly unless the application was called through its entry point. During secure execution, an application can turn off data encryption, e.g., to write out a final result, but code encryption remains switched on until the application exits. Furthermore, applications are able to call unprotected code, but then any affected data will be processed in clear.

Applications are not tied to a specific region, but instead code and data of each application can be spread over the entire address space. In particular, the stack is shared between applications, and the registers of each application are saved to and restored from this single stack. Due to encryption context switching, stack data, including saved registers, is encrypted with a different IV for each application.

3.2.4 Hardware Implementation

We implemented Atlas by modifying the LEON3 processor from Gaisler, a 32-bit SPARCv8 architecture with a seven-stage pipeline and instruction and data caches. The hardware implementation of Atlas consists of two main parts: First, a newly designed encryption unit with the properties described in Sect. 3.2.3 and second, custom instructions were added to the integer unit to configure and control memory encryption. Figure 3.3 shows how the LEON3 architecture was modified.

Encryption Unit So far, the encryption unit was described as a building block which satisfies three properties: It can identify the currently running application, encrypts data without storing state, and has a reconfigurable key. Our implementation stores the identifier of the active application in a dedicated register, which can only be updated through a custom instruction. The device key K_D is always used, except when the encryption unit is configured to secure shared memory. In this case, the unit switches to the secure shared memory key K_S, which is stored in a dynamically configurable dedicated register. Note that this key is only used to encrypt and decrypt shared data, with K_D still being used to decrypt protected code.

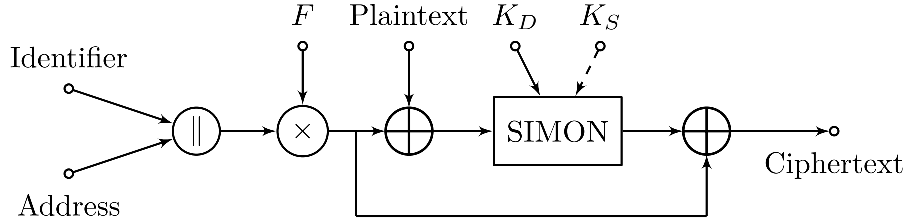

Figure 3.4: Encryption unit with SIMON 32/64 in the LRW tweakable mode of operation.

Figure 3.4 shows a diagram of the encryption unit. The LRW tweakable mode of operation [107] is used to realize stateless encryption of a single 32-bit word. The tweak ensures that every message is unique. In this mode, the ciphertext C is calculated as follows:

$$C = E_K(P \oplus X) \oplus X$$
$$X = F \otimes I$$

where P is the plaintext, X the tweak, E_K encryption with the key K, F the tweak key, and I the IV. Atlas uses the concatenation of the application identifier and the memory address that is being read from or written to as the IV. Both values are 32-bit, so therefore the tweak key F also has to be 64 bits long and the finite field used for the multiplication is GF(2^{64}). X is then truncated to 32 bits before XOR'ing it with the plaintext and output of the cipher respectively.

Since any block cipher can be used in this mode of operation, the choice of algorithm is determined by the word size of the CPU architecture. The LEON3 is a 32-bit architecture where values are read from and written to memory at word granularity. In order to reduce the complexity of the memory controller, a 32-bit block cipher was selected. Additionally, a low-latency single-cycle implementation was used to ensure there is no additional cycle overhead for memory accesses, and to keep the critical path as short as possible. SIMON 32/64 [19] was shown to be the fastest and smallest algorithm with 32-bit blocks [110]. Currently, none of the alternatives with longer keys have low latencies, e.g., KATAN supports 80-bit keys, but has a two times longer critical path. Although 64-bit keys offer short term protection against small organizations, we recommend using PRINCE [25] in the case of a 64-bit architecture. PRINCE has 64-bit blocks and 128-bit keys, and is the fastest single-cycle cipher currently available, with very competitive area [110].

LRW is a tweakable mode of operation, like XTS, which is now widely used to encrypt block devices like hard disks. The reason for choosing LRW over XTS was that the latter passes through the block cipher twice for each block, which would result in a longer critical path. LRW has a known weakness when the plaintext contains the tweak key F. Since the tweak key register is not accessible directly from software, this is not an issue in our design.

Custom Instructions Atlas extends the LEON's integer unit with eight new instructions to give software developers access to the new security features:

ENCENTER stores the current value of the program counter in the identifier register and turns on encryption. It is the first instruction that has to be called at the entry point of any confidential application.

ENCEXIT clears all registers of the encryption unit and turns off encryption. It has to be called whenever there is an exit from a confidential application.

ENCPAUSE turns *data* encryption off without clearing any registers. An application which wants to write to unprotected memory needs to call this instruction first.

ENCRESUME turns *data* encryption on with the currently saved settings, usually resuming confidential execution of the currently running confidential application.

ENCSHMON turns on shared memory encryption. This instruction switches the data encryption key to K_S and uses zeros instead of the application identifier.

ENCSHMOFF turns off shared memory encryption without clearing K_S and resumes isolated execution by switching back to K_D.

ENCSETKEY `ENCSETEKEY` and `ENCSETDKEY` are used to set the encryption and decryption key, i.e., the last subkey of the key schedule, for the SIMON cipher used in secure shared memory. The full 64-bit key is passed within two general purpose registers.

To prevent data leakage, the hardware ensures that the instruction and data cache are always flushed when encryption is enabled or disabled, i.e., when `ENCENTER` and `ENCEXIT` are dispatched. The data cache is not flushed during `ENCPAUSE`, `ENCRESUME`, `ENCSHMON`, or `ENCSHMOFF`, as they are executed by protected code which can be assumed to not leak confidential information. Finally, this also means that except for `ENCENTER`, these instructions will always be encrypted in the binary.

3.2.5 Software Implementation

In order to use Atlas' features, the new instructions need to be dispatched at some point. To this end, we developed a toolchain to expose the functionality of Atlas to programmers as transparently as possible. With our toolchain, usual C programs can be compiled and linked for the modified core, while the programmer only needs to properly separate the functionality into confidential and unprotected code. On a high level, we use ELF rewriting with relocatable object files and executable files, i.e., no compiler patch is needed. It can therefore be easily combined with other existing toolchains.

Confidential Applications Code and data of a confidential application are transparently protected by the encryption unit. With our toolchain, the programmer can define which files constitute a confidential application. The remaining functionality of all other source files is considered to be unprotected. Each application can be written in standard C code, and programmers have the ability to annotate their code with macros. These enable them to call into other confidential applications without data leakage besides the supplied parameters.

Control Flow Rewriting After each confidential application has been compiled, our toolchain parses all relocatable object files and identifies inter-application calls from unprotected code to a confidential application, or vice versa. These calls are then rewritten to go through entry and exit routines, which take care of switching the encryption context. Identifiers for the target function as well as the originating function and application are passed in registers, preserving the original control flow.

The entire context of a confidential application, i.e., all callee-saved registers, is saved and cleared before the context switch, and restored afterwards. Caller-saved registers, which are not used for passing arguments for the current call, are cleared to ensure that no data leaks.

Encryption Since our toolchain supports standard C code, we also provide built-in support for encrypting confidential applications. The code and static data of each application are both placed in separate text and data sections, except for the entry and exit stubs. After the linking step, our toolchain parses the executable file for both sections and transparently encrypts them. Furthermore, it locates all generated stubs belonging to the application in the main text section, and also encrypts those.

Atlas Library While most of our software implementation is part of the toolchain, we also provide a library for programmers. Besides macros for annotation, we provide library functions for copying data between confidential applications and unprotected code, and template functions for opening and accessing secure shared memory sections between different applications. Helper functions are provided to set a shared precomputed key and to copy from and to these sections. Furthermore, we provide a generator to create these routines for an arbitrary number of applications.

3.2.6 Performance and Area Analysis

In this section, Atlas is evaluated regarding performance and area. We obtained results for the Digilent Atlys and Xilinx ML605 development boards, which have Xilinx Spartan 6 and Virtex 6 FPGAs respectively. Xilinx ISE 14.7 was used for synthesis, place, and route.

Critical Path Single-cycle implementations of encryption algorithms result in very long combinational circuits which impact the critical path. Since the memory hierarchy is part of a processor's critical path, the maximum clock frequency of our design is reduced compared to the original design. On the Atlys, the original design can run at a maximum frequency of 78.57 MHz, whereas Atlas can be clocked at 19.05 MHz. We saw similar results on the ML605, where the original maximum frequency of 109.09 MHz was reduced to 31.58 MHz. Embedded systems, however, are typically designed for low power, and therefore not clocked at the maximum possible frequency [36]. Consequently, the actual overhead depends on the application. If the maximum possible frequency of the current design would not be sufficient, the cipher could be serialized to improve performance, trading latency for delay on memory operations.

Microbenchmark Two microbenchmarks have been run on our evaluation platform to measure the performance impact of our toolchain. The first is an application which invokes a confidential one that simply returns. To show the overhead between entering a confidential application and a regular call, we compiled this application with a vanilla GCC toolchain as well as our modified one. The former finishes in 87 cycles, while the latter executes in 227 cycles. The secure context switch and cache flush, which ensure that no confidential data will leak, are responsible for this overhead.

The second benchmark copies 1 KB of data from a confidential application to unprotected memory. This requires encryption to be switched off and on repeatedly, as each data element needs to be loaded into a register while encryption is enabled and written back to memory after encryption has been disabled. This operation is 4.557 times slower than `memcpy`, which is again caused by the cache flushes.

Macrobenchmark To demonstrate the overhead Atlas imposes on real world applications, we wrote an example signing application, which consists of a confidential application with static encrypted data and unprotected code. A message is passed from unprotected code to the confidential application, where it is signed with an asymmetric private key stored securely in the static data section. The signed message is then passed back to unprotected code, where the signature is verified with the corresponding public key. In addition to the overhead imposed by the confidential application call, the message has to be copied from unprotected memory to protected and vice versa. The TweetNaCl [21] library is used to generate and verify the signature.

We compiled this application with an unmodified GCC toolchain and our modified one. When the LEON3 issues partial writes, only the modified bytes are sent over the bus, breaking encryption which requires the full word. Therefore, `stb` or `sth` cannot be used in our current prototype. Consequently, the benchmark was run with data encryption disabled. However, since all other modifications to the core remained in place, e.g., cache flushes, and as the cipher implementation is single-cycle with the design clocked at the same frequency, the performance results are not affected. For both binaries, the execution time was measured with and without copying to and from protected memory. The binary which has all Atlas features enabled imposes an overall overhead of 0.031% compared to the GCC-compiled binary without any secure copies. When secure copies are disabled in the binary compiled with our toolchain, execution takes on average 449 cycles longer than the 1,625,595 cycles of the reference binary. When compiled with GCC and secure copies enabled, the overhead is equal to 0.019%. Recall that both caches are flushed during the execution of `ENCENTER` and `ENCEXIT`, which contribute significantly to the reported overhead. For comparison, when the toolchain-compiled binary compiled with copies enabled is executed on an Atlas core where these flushes were removed, the overhead drops to 0.021%.

	Unmodified	Atlas	Overhead
Digilent Atlys			
Slices	2,496	3,659	46.6%
Registers	3,070	3,333	8.6%
LUTs	6,261	9,726	55.3%
Xilinx ML605			
Slices	5,519	7,970	44.4%
Registers	11,021	12,046	9.3%
LUTs	13,070	18,482	41.4%

Table 3.4: Area in terms of registers, Look-Up Tables (LUTs) and occupied slices.

Area The area usage of Atlas was measured after Xilinx ISE finished place and route. An unmodified LEON3 synthesized with the same settings occupies 2,496 slices on the Atlys. Atlas occupies 3,659 slices, resulting in an overhead of 46.595% (Table 3.4). To reduce the number of required gates, the same cipher core is reused for encryption and decryption. Due to its Feistel structure, both are very similar for SIMON, the biggest difference being the key schedule reversal. Although SIMON is the smallest cipher currently available, cryptography remains expensive in terms of area, especially in case of single-cycle implementations. As mentioned earlier, a serialized implementation could also further improve the area requirements.

3.2.7 Discussion

Atlas does not support SPARC register windows, requiring software to be compiled flatly. The reason is that overflowing or underflowing register windows triggers an interrupt, which currently cannot be handled by Atlas. Enabling interrupts in the current design would violate our security policy, as they circumvent the encryption context switch.

Furthermore, function pointers cannot be used for calls between applications. Since it is impossible to reliably determine the destination address of calculated calls at compile or link time, control flow cannot be rewritten by our toolchain.

Encrypting on word granularity leads to small block sizes of 32 bits, which would change when porting our design to a 64-bit architecture, allowing stronger algorithms to be used, e.g., PRINCE [25]. Alternatively, encryption could be done on cache line granularity.

As was mentioned before, serializing the cipher would improve the clock frequency overhead and further reduce the area requirements of the design. This would come at a cycle cost for each memory access, because the processor would have to wait for the encryption unit to finish. Thus, a good tradeoff would have to be found.

3.3 Summary

In this chapter, we presented Soteria, a lightweight software protection solution with a zero-software TCB. Only the hardware, i.e., a modified variant of the openMSP430, needs to be considered trusted to be able to guarantee confidentiality and integrity of code and data. To the best of our knowledge, this is a novel design and the confidentiality of code against reverse-engineering has not been provided by means of a program-counter based memory access logic before.

Furthermore, we presented Atlas, a scalable security architecture which provides code and data confidentiality for applications through hardware-based memory encryption. Atlas protects intellectual property against system-level attackers in the event of a complete system compromise. It has a zero-software TCB and also protects against physical attacks on main memory. While Atlas is only able to guarantee confidentiality, it scales to a large number of applications.

Based on software protection, all kind of intellectual property for digital contents can be protected. Soteria and Atlas are therefore adaptable to new protection scenarios beyond software protection. We are aware of the dark side of our proposed technology ranging from divisive applications such as DRM to the threat of unanalyzable malware and backdoors. Nevertheless, software protection has many legitimate use cases that must be solved such as the protection of intellectual property, hardening software against vulnerability detection, and protecting cryptographic keys, to name but a few.

4 Exploiting Existing Trusted Computing Architectures

While for small embedded devices there was a lack of trusted computing architectures which also provide code confidentiality, the situation is completely different for general purpose devices. Heavyweight trusted computing architectures have long been available and in addition to code confidentiality also provide sealing and remote attestation capabilities. In this chapter, we want to exploit those existing trusted computing architectures and use them in non-standard ways to demonstrate the possibilities such architectures provide.

The term *exploiting* has a double meaning within this chapter. On the one hand, we show systems which are built on top of trusted computing architectures, i.e., use primitives that are provided by those architectures. On the other hand, we also demonstrate how such an architecture can be broken and how confidential information can be inferred if the attacker assumptions of that architecture are violated. While within the last chapter we mainly focused on root-level attackers, in this chapter we also consider physical attackers with limited capabilities. Such physical attackers are, for example, allowed to probe main memory or to tamper with the master boot record of a physical disk.

The Trusted Platform Module (TPM) is built into virtually every modern notebook or desktop computer available and thus an interesting target. In Sect. 4.1, we leverage the TPM to support a secure boot process for consumer devices in order to provide secure Full Disk Encryption (FDE). In more detail, we implement an authentication protocol which fully defends against traditional evil maid attacks with repeated physical access. Trust is bootstrapped from an external active USB drive which verifies the boot process by utilizing sealed nonces. The underlying principle is that the user (through the active USB drive) and the computer are mutually authenticating each other.

One reason why the TPM never got widely adopted besides during boot up, is that it provides a static Root of Trust (RoT). All software from the very first BIOS settings, to the boot loader, the operating system, and finally the targeted application, contributes to the software measurement and is used, for example, to derive sealing keys. A software update for any of those components would change the measurement and would make unsealing impossible. Consequently, using the TPM to attest the state of a given application to an external party is difficult because, for example, the operating system cannot be changed. To overcome those limitations, Intel introduced the Software Guard Extensions (SGX) which allow to dynamically establish RoTs after the system has been booted by running critical software in so-called *enclaves*. Only software within those enclaves now contributes to the measurement used for attestation and sealing.

While SGX has been designed to protect user level applications, we leverage SGX to also protect kernel functionality (Sect. 4.2). Today's kernels typically have huge code bases and a compromise of one part of the kernel usually leads to a compromise of the whole system. With our approach we are able to protect selected operating system components even in the event of a full system compromise also including a kernel compromise.

SGX aims to shield enclaves against all software attackers including those who gained system level privileges. However, in Sect. 4.3, we show that SGX is generally vulnerable to cache attacks and thus, particular care needs to be taken when writing enclave code. Although Intel excludes cache attacks from the SGX attacker model, they have to be considered as they are especially powerful in a root-level attacker scenario. For example, only root has access to the Performance Monitoring Counters (PMC) which amongst others give detailed information about cache misses but are usually restricted to kernel code.

4.1 Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption

The weakest link in software-based full disk encryption is the authentication procedure. Since the master boot record must be present unencrypted in order to launch the decryption of remaining system parts, it can easily be manipulated and infiltrated by bootkits that perform keystroke logging. Consequently, password-based authentication schemes become attackable as we have shown in Sect. 2.3.1 and Sect. 2.3.2.

We present the design and implementation of Stark, an authentication protocol that, for the first time, fully defends against traditional evil maid attacks with repeated physical access. The goal of Stark is to reveal boot process manipulations before a user enters his password. To this end, we introduce a *mutual authentication scheme* based on *trust bootstrapping* that proves the integrity of a computer to its users.

Contribution Unlike Anti Evil Maid [92], whose authentication scheme is based on a sealed authentication message stored on a passive USB drive, we use an active USB drive that verifies the integrity of the boot process by utilizing sealed *nonces* to securely signal the integrity state of the PC to the user. Roughly speaking, the authentication sequence of Stark works as follows:

- To prevent replay attacks as in Anti Evil Maid (Sect. 2.3.2), we use *nonces*: numbers that are used only once. Each time a number has been verified, a new one is generated by our system. All nonces are sealed using the Trusted Platform Module (TPM) and are stored on an external USB drive.
- Upon boot, a previously generated nonce is verified with an active USB drive. A nonce is sealed by the TPM and can only be unsealed if the boot process behaves with integrity. If the verification fails, the user must not enter his password. The verification status is indicated to the user by an LED on the USB drive itself to defeat PC manipulations.
- The USB drive must be handled like a physical key and must never be left unattended. The trustworthiness of our authentication scheme depends on the trust we have in the USB drive (*trust bootstrapping*).
- Additionally, we store a sealed token value on the USB drive, to bind the drive necessarily to the decryption process (*two-factor authentication*). Thus, for successfully decrypting the hard disk, both the USB drive and the user password are needed such that sniffing the password with a hardware keylogger is not sufficient for decrypting the hard disk.

In short, the trustworthiness of a machine state is authenticated to the user by verifying whether a sealed nonce can be unsealed and by indicating the verification status in the form of an external LED to the user. Replay attacks fail because an attacker can only retrieve nonces that have already been consumed.

To summarize, our contribution is threefold:

1. With Stark, we present a *mutual authentication* protocol between users and computers that is more secure than existing solutions while still remaining user-friendly (Sect. 4.1.1).
2. We provide a practical implementation of Stark, for which we applied our protocol to an active Teensy USB drive and a hypervisor-based FDE solution to support both Linux and Windows (Sect. 4.1.5 and Sect. 4.1.6).
3. We give a formal argument for the security of Stark (Sect. 4.1.7).

Our implementation is open source (under GNU GPL v2). It has been improved significantly compared to the implementation in our previous publication [121] that did not support active USB drives. In addition to our authentication scheme, our implementation is cold boot resistant and offers plausible deniability.

4.1.1 The Stark Protocol

We now describe the actual Stark protocol. Stark is a protocol for mutual authentication between users and computers by means of a trusted external USB device. So there are three parties that follow the Stark protocol: The user $\mathcal{U}$, the computer $\mathcal{C}$ and the trusted device $\mathcal{D}$. Computer $\mathcal{C}$ must contain a TPM, because Stark requires its sealing capabilities to attest to $\mathcal{D}$ that it behaves with integrity. Device $\mathcal{D}$ needs some computing capability to verify the integrity of $\mathcal{C}$ and to indicate the integrity state of $\mathcal{C}$ to $\mathcal{U}$. To attest to $\mathcal{C}$ that $\mathcal{U}$ is who he claims to be, traditional passwords are used.

Note that the protocol is different from the one we described in our previous publication [121]. The protocol explained here uses three parties instead of two and is both more secure and more user-friendly, because the user does not need to remember one-time messages but reusing the same message is prevented. We formulate Stark in the standard notation of authentication protocols. Stark requires a bootstrapping phase, called session 0. Authentication sessions are numbered consecutively starting with session 1.

Bootstrapping Phase In the bootstrapping phase (Figure 4.1), $\mathcal{U}$ defines an initial password p and gives it to $\mathcal{C}$. Computer $\mathcal{C}$ chooses t and n_0 arbitrarily but randomly and sends the initial nonce n_0 to device $\mathcal{D}$. $\mathcal{D}$ must store n_0 securely, i.e., not directly accessible for $\mathcal{C}$. In addition, $\mathcal{D}$ receives $\widetilde{n_0}$ and $\widetilde{t}$, which are sealed with the platform configuration of $\mathcal{C}$; t is a token value that binds the USB drive to the authentication process. Both sealed values $\widetilde{n_0}$ and $\widetilde{t}$ can safely be stored in cleartext. Note that for a practical implementation, some additional values must be created and stored by $\mathcal{D}$. We will explain implementation details in Sect. 4.1.4. After the setup phase, $\mathcal{U}$ and $\mathcal{C}$ can engage in an infinite sequence of authentication sessions.

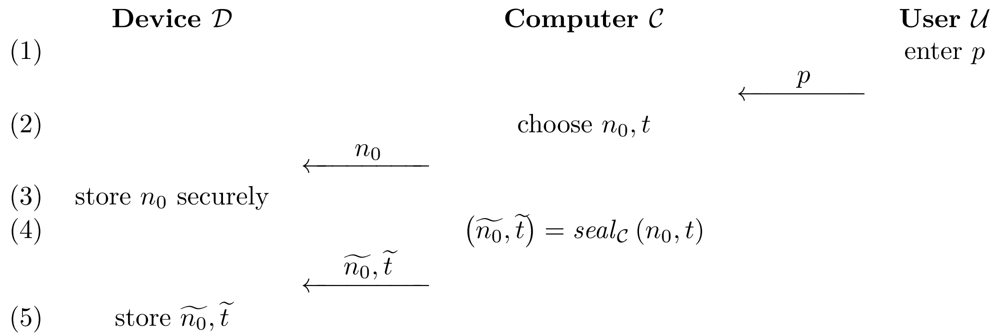

Figure 4.1: Session 0 (bootstrapping phase).

Authentication Sessions In authentication session i, $\mathcal{U}$ plugs in his USB drive and boots up $\mathcal{C}$. With the help of device $\mathcal{D}$, the TPM and the previously stored nonce, the integrity of the boot process can now be guaranteed towards $\mathcal{U}$. In Figure 4.2, the details of an authentication session are listed, given that every step succeeds. In detail, the authentication scheme listed in Figure 4.2 works as follows:

(1) At first, user $\mathcal{U}$ plugs in his trusted device $\mathcal{D}$.

(2) $\mathcal{C}$ now unseals the sealed nonce $\widetilde{n_{i-1}}$ and the sealed token $\widetilde{t}$ from the previous authentication session (or the bootstrapping phase). This step only succeeds if the platform configuration has not changed, e.g., if the BIOS has not been tampered. If unsealing does not succeed, the protocol is aborted.

(3) Device $\mathcal{D}$ authenticates $\mathcal{C}$ by comparing n_{i-1} with the securely stored nonce from the previous authentication session. Whether the authentication succeeded or not is indicated to $\mathcal{U}$ over a separate channel not involving $\mathcal{C}$, e.g., a small LED on $\mathcal{D}$.

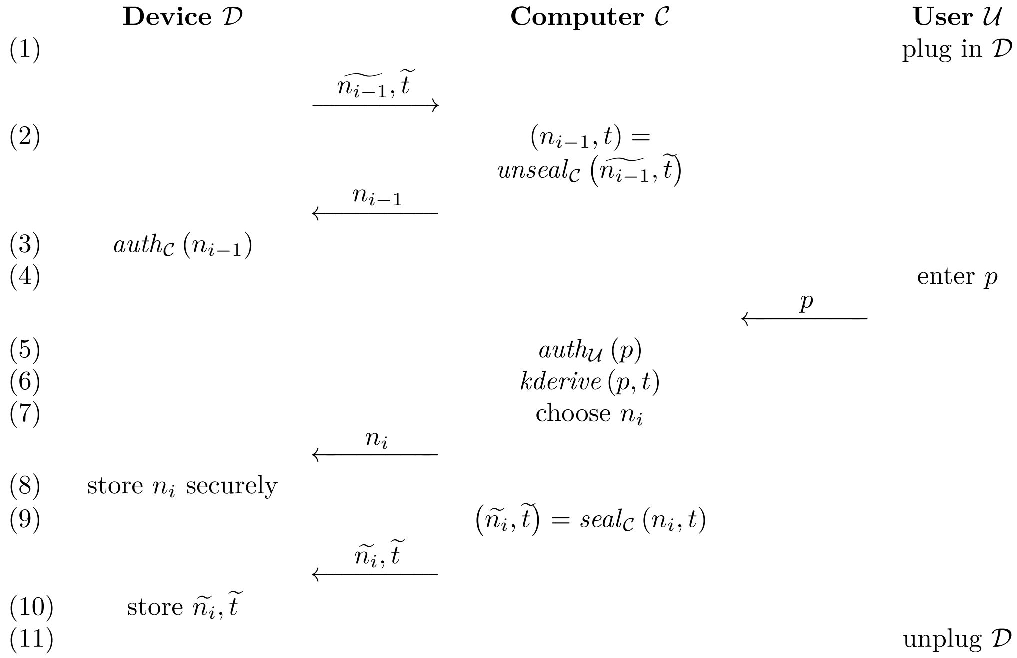

Figure 4.2: Session i, $i \geq 1$ (authentication sessions).

(4) User $\mathcal{U}$ should only enter his password if $\mathcal{D}$ has successfully verified the integrity of $\mathcal{C}$, otherwise the protocol must be aborted.

(5) Computer $\mathcal{C}$ authenticates $\mathcal{U}$ by checking the password p with $auth_{\mathcal{U}}(p)$. If $\mathcal{C}$ fails to authenticate $\mathcal{U}$, the protocol jumps back to step (4).

(6) From password p and token t, a Key Encryption Key (KEK) is derived. This key is used for en- and decrypting the Data Encryption Key (DEK). However, the protocol does not specify key derivation details which are implementation dependent.

(7) Computer $\mathcal{C}$ chooses the new nonce n_i for the next authentication session randomly.

(8) Device $\mathcal{D}$ stores n_i such that it is not accessible for $\mathcal{C}$ anymore. This operation only succeeds if step (3) was previously successful.

(9) The new nonce n_i and the token t are sealed by the TPM of $\mathcal{C}$.

(10) Device $\mathcal{D}$ stores $\widetilde{n_i}$ and $\widetilde{t}$ such that both values are accessible by $\mathcal{C}$. They are needed for the next authentication session.

(11) User $\mathcal{U}$ unplugs device $\mathcal{D}$ and authentication session i has been completed.

This mutual authentication protocol based on trust bootstrapping is the core of our FDE implementation. Some details which are not specified here, for example how $auth_{\mathcal{C}}(n_{i-1})$, $auth_{\mathcal{U}}(p)$ and $kderive(p, t)$ are exactly defined, are specified for our implementation in Sect. 4.1.4. The basic protocol might be used by different implementations in a different way.

4.1.2 Security Argument

Also in traditional cryptographic protocols, replay attacks are oftentimes defeated by *nonces*. The central idea behind the security of Stark is that nonces disallow replay attacks on the verification process between $\mathcal{C}$ and $\mathcal{D}$. For each boot, a different nonce needs to be verified by $\mathcal{D}$ and therefore replay attacks are prevented. In contrast to our previous publication, the security of the scheme does not depend on the user interacting with the system, but is transparently provided by device $\mathcal{D}$. We replace the single authentication messages known from Anti Evil Maid with a series of continuously

alternating numbers that are now verified by device $\mathcal{D}$ (and not by the user $\mathcal{U}$). As most humans have limited computational capabilities, it is beneficial to let the device do the verification. This way the nonces can be chosen randomly by the computer $\mathcal{C}$, which is significantly more secure than letting $\mathcal{U}$ choose messages manually, as in our previous publication.

The problem with letting $\mathcal{U}$ choose the message is twofold. On the one hand, the message might be read by an adversary when the user enters the message or the message might be easy to guess. On the other hand, a user probably notices if a completely different message is shown to him, but he might not notice if an older message is shown again. Both scenarios are critical to the security of our protocol and by using $\mathcal{D}$ as a trusted device, they can both be eliminated. Hence, using the device $\mathcal{D}$ gives a significant gain in security.

To sum up, we use the sealing capabilities of the TPM to authenticate the computer towards an external device which indicates the PC state to the user. After checking the device, the user authenticates towards the computer with a traditional password. So upon each boot, the user only needs to give one input, namely his password. Behind the scenes, a new authentication number n_i is generated and stored on the device. We give a formal correctness argument for Stark in form of an inductive security argument in Sect. 4.1.7.

4.1.3 Design Components of Stark

We now describe the design choices that influenced Stark. There were two questions we had to answer when designing Stark. The first question is: What is the best way to achieve mutual authentication? Since users authenticate using a password, the problem of mutual authentication is the computer having to prove its integrity first. We do this using a novel combination of the TPM and an active USB drive verifying sealed nonces.

The second question is: How can we best protect the necessary authentication nonces while still having a user-friendly protocol? Here we use the USB drive in two different ways. The sealed nonces are stored publicly accessible on the drive and the plaintext nonces themselves are stored such that they cannot be read out by the computer, but just verified by the external drive. Another design decision, not directly related to the evil maid scenario, is the use of token values, which are also sealed and placed on the USB drive. This implements two-factor authentication to protect users who lose the hardware device or choose weak passwords.

Mutual Authentication by Means of the Trusted Platform Module In practice, all implementations to date based on TPMs fail to attest the trustworthiness of boot parameters in a tamper-proof manner. An example for this are tamper-and-revert attacks against BitLocker [116] as described in Sect. 2.3.2. Even though BitLocker leverages TPMs, users can easily be tricked into bogus password prompts because BitLocker looks exactly the same on every machine. An attacker can reproduce the original BitLocker prompt perfectly in his own bootloader and victims have no chance to distinguish malicious from benign behavior. TPMs themselves are neither insecure nor useless, however, they must be integrated carefully into protocols. In Stark, the TPM is used to seal nonces and hence, to bind them to the configuration of a computer.

Bootstrapping Trust from an Active USB Drive We store sealed nonces on external USB drives in a way that makes them accessible by any computer. The cleartext nonces, however, are not directly accessible without additional physical interaction, e.g., only by pressing a button on the external drive. This way, we ensure that only the sealed nonces can be accessed by the computer. If the nonces were stored on the local hard drive, Stark would be insecure to attackers capable of booting the machine and intercepting the nonce after it is unsealed. Such attackers could build their own bootloaders with the exactly same nonce. If merely the computer is compromised, nonces remain confidential. That means we can leave PCs unattended when we ensure the trustworthiness of the USB drive. In this sense, Stark is a protocol that performs *trust bootstrapping* from a trusted USB drive.

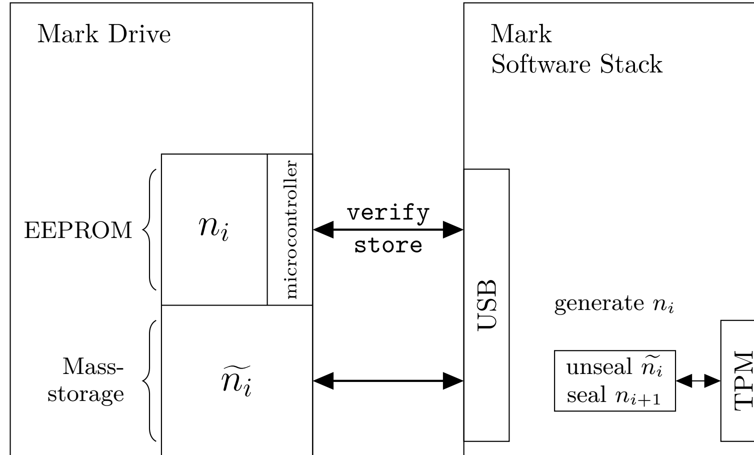

Figure 4.3: Schematic diagram of Mark.

Two-Factor Authentication with Passwords and Token Values The FDE decryption key is in addition to the machine state and the user password based on a token value stored on the external device. Similar to nonces, the token is sealed by the TPM and can only be unsealed if the machine is consistent with its reference configuration. While all components introduced above defeat bootkit attacks, token values prevent dictionary attacks against weak passwords. Without a token value an adversary would be able to boot the target system and try multiple passwords one after another.

Tokens are also a protection in scenarios where the user disclosed his password but did not lose the external drive. Such a scenario can be caused by social engineering [154], skimming and shoulder surfing [102, 15], reusing the password in a different system [90], or writing it down. Random token values prevent such attacks because they bind the external drive compulsorily to the decryption process. The use of a token value is a strong security feature within the Stark protocol because hardware keylogging or even capturing the password with the help of a camera does not allow an attacker to decrypt the hard disk without possessing the external device. Thus, assuming the trustworthiness of the USB drive, Stark renders hardware keylogging and surveillance alone useless for getting access to the hard disk.

4.1.4 Mark: A Practical Implementation of Stark

We now present Mark, our practical implementation of the authentication protocol given above. Note that Mark is a new implementation which is different from our previous implementation named Potts. Mark is an acronym for *Mutual Authentication to Resist Keylogging* and implements the Stark protocol with three parties. The implementation is operating system independent, resistant towards cold boot attacks and provides plausible deniability. In Figure 4.3, a schematic diagram of Mark is shown. The details of this diagram are explained in the next sections.

In the presentation of Stark, we focused on the authentication procedure between $\mathcal{U}$, $\mathcal{C}$ and $\mathcal{D}$. However, we ignored some real-world aspects, like error recovery and usability, to make the protocol more amendable for its security analysis in Sect. 4.1.7. We now make up for those aspects and present a concrete implementation of our protocol. We first present the features provided by Mark and afterwards we show in detail how we have instantiated the implementation specific parts of our protocol.

Features Besides the features that immediately follow from the Stark protocol, like resistance towards evil maid attacks and tamper-and-revert attacks, Mark provides some additional features:

- *Independence of Operating Systems*: Mark is a hypervisor-based solution and supports basically all x86 operating systems. It has been tested with Windows and Linux.

- *Cold Boot Resistance*: Mark utilizes TRESOR (Sect. 2.4) which is an implementation of the AES cipher that stores sensitive information solely in CPU registers, i.e., outside RAM to defeat cold boot attacks.
- *Targeted Manipulations*: Although Mark does not verify the integrity of the whole disk, but only of the boot process, targeted manipulations of the disk are impossible as we are using the XTS mode of operation [51]. An attacker can destroy sectors of the disk but not inject code.
- *Plausible Deniability*: Mark encrypts the whole hard disk including the master boot record such that an encrypted disk looks like a disk with random data. Hence, an adversary cannot decide whether encrypted data is stored on the disk or not.
- *User-Friendliness*: Mark was designed to be as user-friendly as possible. We provide an easy-to-use installer, recovery tools and a self-explanatory graphical user interface as well as user and developer manuals.

The details on how we achieved these features are presented in Sect. 4.1.6. We now want to give a higher level view of the design choices we made to implement Mark.

Key Derivation Function In the protocol specification of Stark, we stated that there is a key derivation function $kderive\,(p, t)$ such that k is derived from the user password p and the token value t. We now specify the key derivation function used in Mark. We do not use a single SHA-256 checksum over p but use the Password-based Key Derivation Function 2 (PBKDF2) which is recommended by NIST [97, 161]. PBKDF2 applies a cryptographic hash to the password along with a salt and repeats the procedure several times to derive the final key (*key stretching*). In Mark, we have chosen the variant

$$kderive\,(p, t) := pbkdf2(\text{HMAC-SHA-256}, p, t, 4096, 256)$$

to derive KEK k from password p. We use the token value t as salt, perform 4096 iterations of SHA-256, and get a final key size of 256 bits. At this point we only have the Key Encryption Key (KEK) k, but to actually encrypt the disk we use a Data Encryption Key (DEK) d. Initially the DEK d is either entered by the user or it is randomly generated by the computer. The DEK d is stored as an encrypted variant D on our device. For this encryption step, AES-256 in CBC mode is used. Thus, the decrypted DEK variant d is derived from its encrypted variant D via

$$d = decrypt(\text{AES-256}, \text{CBC}, iv, k, D)$$

meaning that the DEK d is encrypted with k and AES-256. For the CBC mode of operation we need an initialization vector iv which is stored on our device, too. Decrypted DEK d is a 256-bit value and used to encrypt the disk. The lower 128 bits from d are used as the actual key for disk encryption and the upper 128 bits are used as tweak key for the XTS mode of operation. Taking everything together we are encrypting the data on the disk with AES-128 in XTS mode.

Authentication Functions The remaining functions we did not specify yet are $auth_{\mathcal{C}}\,(n_{i-1})$ and $auth_{\mathcal{U}}\,(p)$. The first function $auth_{\mathcal{C}}\,(n_{i-1})$ is used to authenticate the computer $\mathcal{C}$ to the device $\mathcal{D}$. This function is rather simple: $\mathcal{D}$ receives the nonce from a previous authentication session $i-1$ and compares it to the internally stored nonce $\widehat{n_{i-1}}$ of that session. Thus, $auth_{\mathcal{C}}\,(n_{i-1})$ can be defined as

$$auth_{\mathcal{C}}\,(n_{i-1}) := n_{i-1} \equiv \widehat{n_{i-1}}$$

which represents a simple comparison.

The authentication of the user $\mathcal{U}$ by the computer $\mathcal{C}$ is done via salted hashing again. Therefore, a salt s and the correct password hash h is stored on our device and used by $\mathcal{C}$. The function $auth_{\mathcal{U}}\,(p)$ is finally defined as

$$auth_{\mathcal{U}}\,(p) := pbkdf2(\text{HMAC-SHA-256}, p, s, 4096, 256) \equiv h$$

where the same variant of PBKDF2 as for the key derivation defined above is used. The authentication of the user by the computer in this way is not crucial to guarantee the integrity of the boot process by the protocol. However, it is more convenient to tell the user directly that he has entered the wrong password instead of just deriving the wrong KEK, getting the wrong DEK and consequently fail to boot.

Data Rescue and Recovery Mechanism To recover from protocol failures, from hardware configuration changes, and from system compromise, we use two different keys in Mark: a Key Encryption Key (KEK) and a Data Encryption Key (DEK). KEK k is derived from token value t and the user password p. KEK k decrypts D to DEK d which is encrypted using AES and therefore is safe to be stored without special treatment. DEK d is used to decrypt user data.

While k may change frequently, e.g., through password changes, d is designed to be constant. So d prevents cumbersome re-encryption of the disk in the case of password changes since only d must be re-encrypted. Furthermore, d allows for data recovery in the case of password loss, and in case the USB drive is lost or TPM hardware failures occur. For data recovery, users must store d in plaintext at a physically secure location, e.g., in a vault at home. This allows for migration of the hard disk into other machines and can be consulted in the case of hardware configuration changes that lead to different TPM states.

Not all protocol inconsistencies must necessarily point to a system compromise but can arise from technical problems, too. However, this is hard to differentiate in practice, and thus we advise users to act carefully: In practice, data can be recovered with d by using our recovery procedure to recover a potentially compromised HDD on a second, trusted computer. After data rescue, the HDD in question must be formatted, including the master boot record, and the PC must be reset to factory settings (including the BIOS or UEFI image). Only then, Mark's bootstrapping phase can be rerun securely to set up a new system. Overall, Mark enables users to *identify* a potential system compromise, but we do not strictly regulate which actions to take if such a compromise is detected in Mark.

Usability vs. Security After we specified the instantiation of Stark in detail, the question arises how usable Mark actually is. Concerning the tradeoff between security and usability, we managed to get considerable results as compared to our previous implementation. Figure 4.2 shows that the user is not required to take additional actions. His only tasks are plugging in and plugging out the device as well as entering a password which would be required by other authentication methods as well.

However, a reasonable argument against the practical applicability of Mark might be the additional overhead we demand from users by checking the indication of the device LED on every boot. If a user does not check the device LED but just enters his password, the authentication scheme of Mark falls back to that of BitLocker. We therefore advice the user to always check the indication of the device LED when booting as it is crucial for the security of Mark. We believe that the overhead is acceptable compared to the gain in security provided by our implementation.

4.1.5 Hardware Implementation

In this section we give some information about the hardware we used to implement Mark. The Stark protocol requires that the hardware must satisfy certain properties to be suitable as a trusted device. First of all, the device must be capable to store sealed data as well as a pre-boot environment for booting from the device. For these requirements a USB thumb drive would be sufficient. However, in addition the device must provide the ability to securely store the nonce of the current authentication session and there needs to be a way to verify the nonce of the last authentication session and to indicate the verification status to the user.

It is clear that a passive thumb drive is not sufficient for this task, but of course there is more than one solution to comply with these requirements. In the rest of this section, we show an exemplary hardware solution that fulfills all requirements while still being easy to build.

Hardware Components We decided to use *Teensy USB* [130] as an active programmable USB drive for indication and verification purposes. Teensy USB is a development board with an AVR processor supporting USB directly. It can be used in many different configurations depending on the way it has been programmed. For example, keyboard and mouse behavior can be simulated. As our case is simple, we use the Teensy USB as a serial device and define specific commands for the communication with the PC. For indicating the verification status to the user, we use a single LED on the Teensy USB, but of course every kind of feedback mechanism would be equally suitable. However, it is crucial to the protocol that the indication takes place without involving the connected computer, i.e., over a separate channel, because otherwise the verification status might get tampered.

To fulfill the second requirement, i.e., to store sealed data and to be able to act as bootable USB drive, we use a common USB thumb drive as mass storage and connect both components, the Teensy USB and the thumb drive, via a USB hub (Figure 4.3). This setup is good enough to act as a prototype and to test our implementation. Of course, for a production-ready implementation, both components should be integrated into one case, but from a technical point of view there is no difference.

To make sure that the securely stored nonce cannot be accessed by the computer while the Teensy USB is connected, we burn the nonce into the EEPROM of the AVR at every boot. To access data in the EEPROM directly, the computer would need to put the Teensy USB into *flash mode* which is only possible if the user presses a physical button on the Teensy USB. We therefore recommend the user to never touch the button unless he wants to install a new system or needs to recover from system failure. To verify the integrity of the boot process utilizing the securely stored nonce, we defined a communication protocol consisting of two serial commands.

Hardware Communication Most steps of the Stark protocol are software-based and a detailed description of the Mark boot process is given in Sect. 4.1.6. To verify the previous nonce and to indicate it to the user, however, our external device needs to communicate with the computer. To this end, we define two commands that the computer is allowed to send to the device over a simulated serial connection (Figure 4.3):

- `verify`: With the help of this command, the computer sends the unsealed nonce of a previous authentication session to the device. The device compares the nonce with the one stored in its EEPROM and if they are equal it switches on an LED. If they are different, the LED blinks three times and the user is strongly advised against entering his password because the system might be compromised.
- `store`: This command is used to store the newly generated nonce of an authentication session in the EEPROM of the device. This command only succeeds if the previously defined `verify` command has been successful.

Other communications over the serial connection to the Teensy USB are ignored. As the computer is unable to catch a nonce from a previous authentication session without unsealing it, it cannot forge the authentication status that is shown by the device.

4.1.6 Software Implementation

Using TreVisor as basis for our Stark implementation Mark was clearly motivated by the fact that we wanted to build an FDE system which is more secure than common FDE solutions in use today. Cold boot attacks do not fall into the defined threat model, but combining the concepts of Stark with that of TreVisor seemed reasonable.

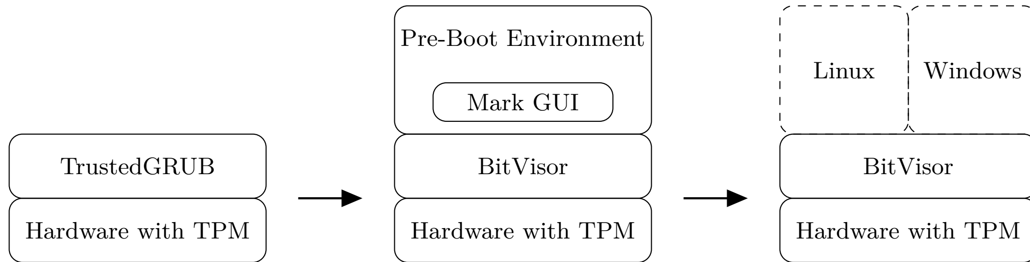

Figure 4.4: Mark software stack.

Figure 4.4 visualizes the software stack of Mark during a boot process. First, the TrustedGRUB bootloader runs directly on top of the hardware. Then, a Linux-based pre-boot environment that displays the Mark GUI, runs on top of a patched BitVisor hypervisor. Finally, the target operating system, e.g., a Linux or Windows takes over system control. All components apart from the target operating system, which is stored on the encrypted disk, are loaded from the USB thumb drive.

Combining Stark with TreVisor When an attacker gains physical access to a machine while it is *running* or in *standby mode*, it is not protected by Stark but must be protected by separate mechanisms. Therefore we used TreVisor (Sect. 2.4) as basis for Mark because TreVisor already defeats another kind of physical access attacks, namely cold boot attacks.

Cold boot attacks exploit the *remanence effect* of DRAM [80], meaning that RAM contents fade away gradually over time. Due to the remanence effect, encryption keys can be restored from memory through rebooting a system with a malicious USB drive, or by replugging RAM chips physically into another machine. This makes cold boot attacks rather generic and therefore they constitute a threat for *all* software-based FDE technologies to date.

Software Components We now give more information about the implementation of Mark on the PC side. Specifically we list the different software components Mark is composed of.

- Mark is based on BitVisor [142], a thin hypervisor that is capable of intercepting accesses to the hard disk and performing actions like encryption transparently to the operating system. We patched it to support the TRESOR encryption routine in XTS mode of operation. Moreover, we modified the hypervisor to boot from any hard disk that is found on a system and not just the first one. Because we use a hypervisor, our FDE solution is operating system independent.
- To authenticate the computer with our external device, we use a Linux-based pre-boot environment. We compiled a minimal Linux kernel capable to load a specifically designed ramdisk. Our verification and bootstrapping software runs within this ramdisk. The reason behind this design decision is that TPM toolchains like TrouSerS [5] are available for user mode only, and that talking to the TPM on a lower level turned out to be difficult. Another advantage of using a Linux-based environment is its driver support for storage devices like hard drives.
- We designed a graphical user interface to access the functionality that our software provides. After booting our FDE solution, the user can choose between booting an existing system, installing a new system, and recovery from system failure. For the installation of a new system, no special knowledge is required but users must simply follow the on-screen instructions.

Booting the System To explain how Mark works, we show an exemplary boot process. We assume that everything was set up and that no errors occurred. In a nutshell, the boot process of Mark follows the following sequence (Figure 4.4):

1. Users plug in the Mark USB drive and boot the PC from it. With the help of *TrustedGRUB*, a patched version of the BitVisor hypervisor is loaded. TrustedGRUB is an extended version of the well-known GRUB offering TPM support. A secure bootstrap architecture is possible because TrustedGRUB measures the whole boot process and makes it verifiable for system integrity checks.
2. The hypervisor then loads a minimal Linux kernel together with a ramdisk. The ramdisk consists basically of a minimal Debian system.
3. Our graphical user interface starts within this ramdisk and the user has the choice to install a new system or to boot into an existing system. In the following we assume booting into an existing system is selected.
4. The nonce of the last boot is unsealed. This step only succeeds if the platform configuration is unchanged. An attacker has no possibility to tamper the graphical user interface and to get the previous nonce at the same time.
5. The nonce is sent to the device and an LED on the device signals whether the verification succeeded. For the PC, it is not possible to switch the LED on without knowing the previous nonce.
6. The user should enter his passphrase if and only if the LED on the device is switched on. With the passphrase, the KEK can be derived and finally the DEK can be loaded.
7. A new nonce is generated and stored into the EEPROM of the drive. This step is only possible if the verification of the previous nonce succeeded.
8. With the DEK, the target system can now be booted. Therefore a special `vmcall` instruction is executed via `kexec` which passes the DEK to the hypervisor and initiates the boot process.
9. The hypervisor transfers the DEK to the debug registers and wipes the whole memory to make sure that no key fragments remain in RAM.
10. The hypervisor hooks all disk operations and transparently decrypts everything with the DEK, including disk operations that involve the MBR. It is perfectly possible to encrypt the whole hard disk this way. Finally, the hypervisor loads the first sector of the hard disk and jumps to it.
11. The target operating system takes over control and the boot process of the target system continues like a normal system boot.

While this implementation might seem complex, it follows a clean design and not much interaction is needed by the user. Furthermore, the design choice provides us with a flexible system for future enhancements. Since the user password is not entered in a pre-boot environment but in a full Linux system, we can easily add functionality to bootstrap trust from different external devices in addition to the active USB drive.

Graphical User Interface We now give a brief overview of our graphical user interface illustrated by the two screenshots in Figure 4.5 and 4.6. The graphical user interface automatically starts up after the hypervisor has loaded our Linux-based pre-boot environment. After the booting has completed, the user is presented with the main menu shown in Figure 4.5. From this menu the user can choose whether he wants to boot an existing system, install a new system, perform configuration or recovery or reboot his machine. If the user chooses to boot an existing system, he has to check the LED on the external drive, enter his password and remove the USB drive.

Figure 4.6 shows an example dialog during the initial configuration. In the background, every step that needs to be performed is listed and different colors indicate which steps are already completed, still open or failed. Every time the user needs to take action a dialog pops up and shows instructions. In this example the user needs to press a button on the drive because an initial image needs to be flashed to the drive. User interaction is needed here to prevent an attacker from flashing the drive

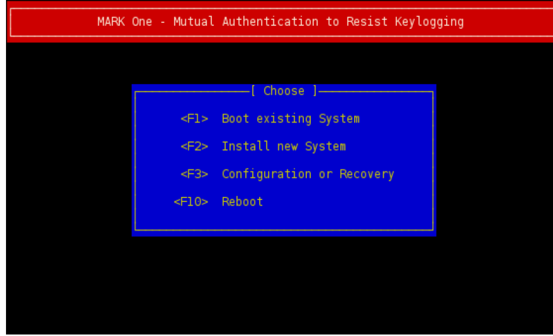

Figure 4.5: Mark main menu.

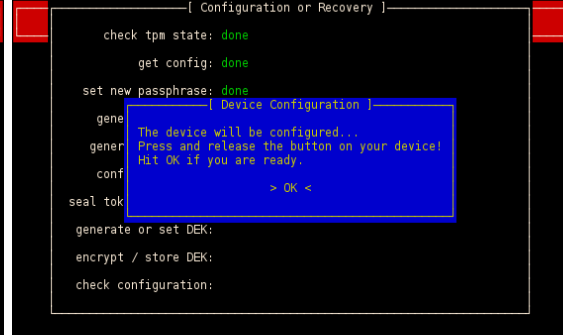

Figure 4.6: Mark setup phase.

with his own firmware or forging the verification status. If anything fails the user can safely abort the current procedure and restart. No expert knowledge is required for using or installing Mark.

4.1.7 Formal Security Analysis

In this section, we formalize Stark as a security protocol within an abstract system model. To this end, we add a fourth player: Attacker $\mathcal{A}$ who wants to break the authentication scheme between $\mathcal{U}$, $\mathcal{C}$ and $\mathcal{D}$. Parties $\mathcal{U}$, $\mathcal{C}$ and $\mathcal{D}$ follow the protocol given above (Sect. 4.1.1) and additional rules given below. Attacker $\mathcal{A}$ can act arbitrarily under the restrictions described below.

In the formal model of Stark, parties communicate by exchanging messages in an abstract sense. This can correspond to typing text on the keyboard, displaying a message on the screen, or attaching a USB drive. Due to the immediate geographical vicinity of all parties, we assume that message exchange is reliable and synchronous, meaning that if a party sends message m, the other party receives m within a short delay.

The senders of individual messages cannot be authenticated, i.e., the receiver of a message does not necessarily know who sent the message. For example, $\mathcal{C}$ cannot distinguish text typed by $\mathcal{A}$ or $\mathcal{U}$. Similarly, a nonce verification request to $\mathcal{D}$ may come from $\mathcal{C}$ (using the original boot loader) or from $\mathcal{A}$ (using a forged boot loader).

User Model We now give the rules that $\mathcal{U}$ has to conform to. $\mathcal{U}$ corresponds to a human user sitting in front of the keyboard and wishes to authenticate securely to $\mathcal{C}$. The security of Stark relies on the following rules for $\mathcal{U}$:

- *Completeness*: $\mathcal{U}$ must complete the protocol and not abort any step as soon as he starts session i.
- *Coherence*: $\mathcal{U}$ must not leave the computer during the authentication phase but execute all steps consecutively.
- *Correctness*: If $\mathcal{D}$ fails to authenticate $\mathcal{C}$ and indicates this to $\mathcal{U}$, $\mathcal{U}$ must abort the protocol entirely and never engage in the protocol with the same participants again.

If the completeness rule is violated it might happen that the new nonce n_i has not already been written to the secure storage of $\mathcal{D}$. Furthermore it is possible that $\mathcal{A}$ has captured n_{i-1} and thus is able to forge the verification indication during the next boot. Coherence is crucial because otherwise $\mathcal{A}$ is able to simply manipulate $\mathcal{D}$ directly. If the correctness property is violated $\mathcal{U}$ does not check the integrity of $\mathcal{C}$ at all and therefore $\mathcal{A}$ is able to perform arbitrary manipulations.

Computer Model We now define the possibilities for $\mathcal{C}$. Unlike $\mathcal{U}$, $\mathcal{C}$ is an electronic system that has powerful computation capabilities, including crucial cryptographic primitives. Stark relies on the following properties of $\mathcal{C}$:

- *Integrity*: $\mathcal{C}$ must not be compromised initially during setup phase.
- *Cryptographic Capability*: We assume that $\mathcal{C}$ can *seal* information as follows: Given a precise software configuration c of $\mathcal{C}$, $\mathcal{C}$ can encrypt and sign a message m such that $\mathcal{C}$ itself can decrypt it only when being in c again. It is unfeasible to seal or unseal information for any party besides $\mathcal{C}$ as well as for $\mathcal{C}$ itself if it is not in c.
- *Singularity*: Every nonce n_i chosen by $\mathcal{C}$ must be unpredictable and used only once.
- *Reliability*: $\mathcal{C}$ waits indefinitely for password p to be entered without shutting down. Reading from and writing to a USB thumb drive does not fail. Neither does communication with $\mathcal{D}$ over the serial line or any other electronic operation fail.

If the integrity property is violated the whole Stark protocol is useless as it would guarantee the integrity of a potentially initially compromised system. Intuitively, the cryptographic capability of $\mathcal{C}$ corresponds to the sealing capabilities of TPMs, meaning that $\mathcal{C}$ is required to have a TPM. Most notably, configuration c encompasses the BIOS settings as well as the whole software on our external drive. If the cryptographic capability is violated the Stark authentication scheme might be broken, e.g., because $\mathcal{A}$ might be able to unseal nonces and the token or $\mathcal{C}$ might unseal information when not being in c. In both cases the verification indication for $\mathcal{U}$ can be forged and thus $\mathcal{U}$ would enter his password into a potentially forged prompt.

The singularity property is not allowed to be violated because $\mathcal{A}$ might have recorded all previous nonces and could try to let them verify by $\mathcal{D}$ within a forged bootloader. If a nonce is reused by $\mathcal{C}$ at any point or is somehow predictable, the verification might succeed although the system is compromised. Note, however, that this kind of attack is also prevented by our implementation Mark. Firstly $\mathcal{D}$ blinks on any wrong verification attempt which should be noticed by $\mathcal{U}$. Secondly $\mathcal{D}$ has to be replugged after the third wrong attempt. In practice the singularity property is fulfilled by using a proper random number generator.

Violating the reliability property of $\mathcal{C}$ is similar to violating the correctness property of $\mathcal{U}$. $\mathcal{C}$ needs to wait indefinitely for the password as it has to be ensured that the new nonce is written to $\mathcal{D}$. For the same reason communication, reading and writing to the thumb drive is not allowed to fail. If any of these rules is violated $\mathcal{A}$ might forge the verification indication during the next boot. The reliability of $\mathcal{C}$ is an interesting property, both in practice as well as in theory, because it is generally hard to differentiate between malicious and faulty behaviors. This makes secure recovery mechanisms in the case of accidental data corruption difficult.

Device Model We now define the rules for $\mathcal{D}$. Unlike $\mathcal{U}$, $\mathcal{D}$ is an electronic system, but unlike $\mathcal{C}$ it does not have powerful computation capabilities. Most notable it does not have enough entropy to generate keys or numbers with sufficient randomness. This is the reason why $\mathcal{C}$ chooses the nonces instead of $\mathcal{D}$. Stark relies on the following properties of $\mathcal{D}$:

- *Correctness*: The firmware of $\mathcal{D}$ must be correct in the sense that it is impossible for $\mathcal{C}$ to read out the stored nonce n_{i-1} of a previous authentication session. Also, storing a new nonce n_i must be forbidden for $\mathcal{C}$ unless the verification of n_{i-1} succeeded.
- *Confidentiality*: $\mathcal{D}$ must remain confidential at any point in time.
- *Reliability*: $\mathcal{D}$ waits indefinitely for nonce n_{i-1} to be sent for verification and for nonce n_i to be sent for storage. In the meantime $\mathcal{D}$ does not lose its internal or the verification state.

If correctness is violated, $\mathcal{A}$ is either able to read out n_{i-1} of a previous session or is able to store an arbitrary new nonce to $\mathcal{D}$ without the need of having verified n_{i-1} before. In both cases he now knows the nonce that is stored securely by $\mathcal{D}$ and thus is able to send this nonce to $\mathcal{D}$ for verification while displaying a forged password prompt. Confidentiality must not be violated because otherwise $\mathcal{A}$ is either able to manipulate the firmware of $\mathcal{D}$ or might be able to read out the secure storage of $\mathcal{D}$ which is not accessible by $\mathcal{C}$ without interaction of $\mathcal{U}$. Confidentiality is crucial and has to be ensured by $\mathcal{U}$.

Reliability is a crucial property for $\mathcal{D}$, because $\mathcal{D}$ is not allowed to lose its state. It is possible that $\mathcal{C}$ cuts the power for $\mathcal{D}$ because $\mathcal{A}$ forged the bootloader. In that case, however, $\mathcal{U}$ would notice the fraud since successful verification cannot be indicated by $\mathcal{D}$ even if $\mathcal{C}$ tries. For this reason the reliability property is not considered violated because the power cut was initiated by $\mathcal{C}$. If, however, reliability is violated because, for example, $\mathcal{D}$ does not wait indefinitely for nonce n_i to be sent for storage, the new nonce might not be stored correctly and thus $\mathcal{A}$ could indicate successful verification from compromised $\mathcal{C}$ with the previous nonce n_{i-1}.

Attacker Model Attacker $\mathcal{A}$ corresponds to an evil maid who may additionally be equipped with electronic devices. So $\mathcal{A}$ can perform both human actions as well as computationally complex calculations. Overall, $\mathcal{A}$ acts arbitrarily under the following restrictions:

- Attacker $\mathcal{A}$ can let $\mathcal{C}$ unseal any data (when plugging in a USB drive and booting $\mathcal{C}$), but cannot break cryptography, i.e., he cannot unseal data himself. As stated above, this is ensured by the cryptographic capability of $\mathcal{C}$, i.e., by the TPM.
- Attacker $\mathcal{A}$ can inject or replay arbitrary messages before and after the authentication process of Stark, and he can start an authentication process himself. But he cannot send messages to $\mathcal{U}$, $\mathcal{C}$ or $\mathcal{D}$ within complete protocol parts, i.e., he is not allowed to interrupt the authentication process of $\mathcal{U}$. This corresponds to the event that $\mathcal{A}$ interferes with $\mathcal{U}$ while he boots up $\mathcal{C}$, which we exclude. This is ensured by the completeness and coherence properties of $\mathcal{U}$.
- Attacker $\mathcal{A}$ is allowed to send arbitrary messages to $\mathcal{D}$ via the serial line, but he is not allowed to perform actions which would require physical interaction by $\mathcal{U}$, e.g., set $\mathcal{D}$ to flash mode. This is ensured by the confidentiality property of $\mathcal{D}$.
- Attacker $\mathcal{A}$ cannot get physical access to $\mathcal{D}$. Nevertheless, he can make a one-to-one copy of it after manipulating the MBR (as sketched in the attacks from Sect. 2.3.2); so he may know all nonces up to n_{i-2}. The copy, however, does not include the secure storage of $\mathcal{D}$ which is not directly accessible by $\mathcal{C}$ according to the correctness property of $\mathcal{D}$.
- Attacker $\mathcal{A}$ is allowed to gather all kind of information about the protocol participants, but he is not allowed to sniff the traffic between $\mathcal{C}$ and $\mathcal{D}$, e.g., on the USB bus. Note that this alone would not compromise security entirely, but as our protocol currently specifies cleartext communication this restriction has to be made.

Security Argument The following security argument explains the achieved security of Stark. We argue that over an infinite sequence of sessions, the following invariant is maintained at the beginning of every session i (the security of Stark follows from item 5 of the invariant):

1. User $\mathcal{U}$ knows p.
2. Device $\mathcal{D}$ *securely* stores n_{i-1}.
3. Device $\mathcal{D}$ stores $\widetilde{n_{i-1}}$.
4. Attacker $\mathcal{A}$ does not know n_{i-1}.
5. Attacker $\mathcal{A}$ does not know p.

Note that token t does not play a role in our security model, because t does not add security to the evil maid scenario, but in other scenarios as described in Sect. 4.1.3.

We now show that Stark maintains the invariant by induction over the number of sessions. The base case (session 0) is straightforward and follows from integrity property of $\mathcal{C}$ and the confidentiality property of $\mathcal{D}$. In the setup phase it holds: $\mathcal{D}$ stores n_0 securely and $\widetilde{n_0}$ in cleartext; the attacker $\mathcal{A}$ can neither have the password p nor the nonce n_0 because of the initial integrity of $\mathcal{C}$ and the confidentiality of $\mathcal{D}$.

For the induction step, assume that the invariant holds at the beginning of session i. There are two possibilities: Either $\mathcal{A}$ manipulates $\mathcal{C}$ and tries to inject messages, or $\mathcal{U}$ starts the protocol with

regular behavior of $\mathcal{C}$, possibly after a reboot, and possibly after his USB drive was cloned. We discuss both cases in turn:

- Case 1 ($\mathcal{A}$ manipulates $\mathcal{C}$ and tries to inject messages): Since $\mathcal{A}$ does not know n_{i-1} (from invariant) and n_{i-1} can neither be guessed (singularity property of $\mathcal{C}$) nor be retrieved from $\mathcal{D}$ (correctness property of $\mathcal{D}$), any value injected by $\mathcal{A}$ towards $\mathcal{C}$ results in a wrong nonce sent for verification to $\mathcal{D}$. $\mathcal{C}$ cannot unseal $\widetilde{n_{i-1}}$ because it is not consistent with its reference configuration (cryptographic capability of $\mathcal{C}$). Consequently $\mathcal{D}$ does not indicate the successful verification of $\mathcal{C}$ to $\mathcal{U}$ and the protocol is aborted and never restarted with the engaged parties (correctness property of $\mathcal{U}$).
- Case 2 ($\mathcal{U}$ starts the protocol with regular behavior of $\mathcal{C}$, possibly after a reboot): $\mathcal{A}$ may know $\widetilde{n_{i-1}}$, and hence, $\mathcal{A}$ can learn n_{i-1} from $\mathcal{C}$. However, $\mathcal{A}$ cannot inject any messages because $\mathcal{U}$ started the protocol (coherence property of $\mathcal{U}$). $\mathcal{U}$ successfully authenticates towards $\mathcal{C}$ using his password p, and completes the protocol (completeness property of $\mathcal{U}$). During this process, $\mathcal{C}$ has chosen n_i and n_i is stored securely by $\mathcal{D}$; $\widetilde{n_i}$ is stored by $\mathcal{D}$ accessible for $\mathcal{C}$ (reliability property of $\mathcal{C}$ and $\mathcal{D}$). Furthermore $\mathcal{A}$ is not able to sniff n_i while it is transferred from $\mathcal{C}$ to $\mathcal{D}$ (restriction on $\mathcal{A}$).

In both cases, session i completes and session $i+1$ commences. Before the new session starts, observe the following points:

1. Of course, $\mathcal{U}$ still knows p.
2. From session i, $\mathcal{D}$ stores n_i securely.
3. Furthermore, $\mathcal{D}$ stores a sealed version $\widetilde{n_i}$ of it.
4. Attacker $\mathcal{A}$ does not know n_i. Indeed, $\mathcal{A}$ may know n_{i-1}, but $\mathcal{A}$ cannot exploit it to fool $\mathcal{U}$ because $\mathcal{D}$ would fail to verify n_{i-1} and only succeed to verify n_i.
5. Attacker $\mathcal{A}$ does not know password p.

Overall the invariant still holds at the beginning of session $i+1$, concluding the security argument. Crucial for our argument is that *complete protocol phases* restrict the behavior of adversary $\mathcal{A}$. As explained above, we assume that $\mathcal{U}$ completes session i, meaning that when $\mathcal{U}$ executes the first step of a session, $\mathcal{U}$ does not abort but runs the protocol to completion. Without the completeness property, Stark would be insecure.

4.1.8 Discussion

While implementations of the Stark protocol like Mark guarantee the integrity of the boot process they do not guarantee the integrity of the entire hard disk. Thus, they are able to prevent malicious bootloaders but they cannot prevent an attacker from manipulating sectors of the disk. However, because we are using AES in XTS mode, at least targeted manipulations become impossible. An attacker is able to destroy entire sectors of the disk, but he is not able to inject malicious data or code, which might then be executed by the operating system. Like we have shown in separate work [69], providing integrity for disk data requires changing the visible sector size due to the additional space needed for integrity tags. Although this is perfectly fine for systems developed from scratch, it would break compatibility with existing operating systems such as Windows and Linux.

Mark defeats traditional evil maid attacks that require physical access to the machine. In practice, however, the borderline from software-only attacks to more complex, hardware-based attacks is blurred. For example, hardware-based attacks can install hidden USB or PS/2 keyloggers. Hardware keyloggers cannot be detected by Mark, but nevertheless it is not possible for an attacker to decrypt the hard disk after having captured the user password with a keylogger. Unfortunately Mark cannot protect other sensitive information from keylogging, too, e.g., passwords used in online forms.

A type of hardware attacks which are not necessarily detected by Mark are *bus sniffing attacks*. For example an attacker could try to sniff physically on the USB bus or the LPC bus that is connected with the TPM [169]. If an attacker manages to sniff on the USB bus, for example by monitoring USB downstream traffic with a passive USB device on a different port connected to the same root hub, he might intercept the newly generated nonce n_i that is written to the external drive. Afterwards a modified tamper-and-revert attack could be used to display a forged password prompt to the user and finally get the password. Although this attack is difficult in practice, it might work without the user noticing it. However the attacker is not able to get the token t, which is needed to derive the KEK, and thus still not able to decrypt the hard disk.

To sum up, sophisticated hardware-based attacks cannot be counteracted well by software solutions. In the abstract model of Stark, attackers can arbitrarily manipulate the software configuration, including the BIOS and the MBR, but manipulating hardware that is not measured by the TPM might remain undetected.

4.2 Isolating Operating System Components with Intel SGX

The protection of computer systems against malicious applications and the isolation of software components is still a difficult task for software developers, software architects and researchers. If one software component within a given system is compromised, it is often easy to compromise the other components with the help of privilege escalation. This is especially true for compromised software components *within* the operating system as one compromised operating system component can easily compromise other components of the kernel without the need for further vulnerabilities.

The security model of Intel SGX states that only the CPU needs to be trusted and all software including applications and the operating system is considered untrusted. Secured containers (trusted enclaves) are protected by hardware mechanisms and no malicious application, operating system or virtual machine monitor can access the information in the secured container. Furthermore, the memory belonging to these containers is transparently encrypted by the CPU to defend against physical hardware attacks.

Given the security model of SGX with its strong security guarantees and protection mechanisms against potentially malicious high-privileged system software, SGX looks like a promising hardware extension to shield kernel components against each other and enforce isolation of specific parts within an operating system. However, SGX enclaves can only be entered from ring three (user mode) and not ring zero (kernel mode). Consequently, it is not directly possible to put kernel functionality within SGX enclaves and to use this functionality from non-protected parts of the kernel. To still be able to provide isolation for operating system components, we developed a solution for moving parts of an operating system kernel to user space and protecting those parts with the security features provided by Intel's SGX.

Contribution We present a generic concept to provide SGX security guarantees for operating system components. Our concept provides tamper resistance for kernel functionality and is able to strictly isolate operating system components from each other. To the best of our knowledge, we are the first showing that Intel SGX can be used to secure kernel functionality. In detail our contributions are:

- Instead of shielding user mode applications from other user mode applications, we leverage SGX to secure Linux kernel modules. Similar to a microkernel, kernel components are isolated from each other such that a vulnerability in one kernel module cannot escalate into compromising the kernel. Furthermore, our solution prevents privileged malicious user space applications from attacking secured kernel modules.
- Because SGX has not been designed to work in kernel mode, we present a generic concept to move parts of the kernel to the user space and then protect the given functionality by wrapping it in SGX enclaves.
- We provide a proof-of-concept implementation of our concept which protects one kernel function, namely full disk encryption, using an Intel SGX enclave in user space. Our implementation is seamlessly integrated by using the Linux Crypto API. The integrity of the disk encryption as well as the confidentiality of the encryption key is protected against all software level attackers. In addition, the disk encryption key is also protected against physical attacks.
- Compared to usual disk encryption solutions, our prototype derives the encryption key not only from a user password but also from a second factor bound to the machine. This factor is stored sealed by the platform such that it can only be used indirectly by a software attacker on that same platform. Retrieving the user password and stealing the hard drive is insufficient for breaking disk encryption.
- We evaluated our prototype regarding correctness, security, and performance. Although our solution is a step forward in terms of security, it imposes a notable performance overhead up to factor 100.

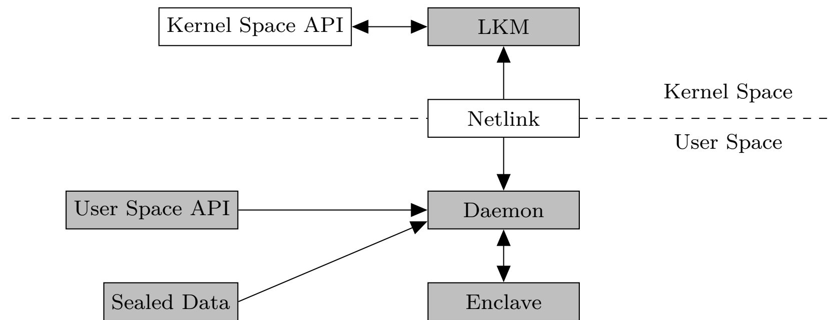

Figure 4.7: A Loadable Kernel Module (LKM) using functionality wrapped by an enclave.

4.2.1 Architecture

As described above, it is not possible to enter an enclave from kernel space. An enclave's code has always to be executed in ring three with a reduced set of allowed instructions and a limited amount of available memory. Furthermore, it is not possible to initiate an enclave on its own but instead the Intel *launch enclave* must be used to generate the appropriate launch token.

To overcome these major limitations of SGX, we decided to build an architecture which moves part of the kernel functionality to user space such that the core functionality can then be wrapped by an enclave. This enclave is implemented by a user space service or daemon which calls the launch enclave for initialization. Once the enclave is running, functionality within the enclave can be used by the daemon. Consequently, the kernel first has to communicate with the daemon which then passes the request to the enclave.

Figure 4.7 shows the architecture which enables Linux kernel modules to use functionality wrapped by SGX enclaves. On the one hand, the LKM uses the API of the Linux kernel to provide a certain functionality within the kernel. On the other hand, the LKM communicates with a user space daemon via a Netlink interface to send certain requests or receive responses. The daemon just forwards the requests and responses to and from the enclave and thus, the enclave is able to provide functionality to the LKM which the LKM itself provides to the kernel. In addition, the daemon is allowed to interact with other user space applications if necessary and stores sealed data of the enclave. Consequently, the enclave and thus the LKM is able to securely store persistent data across reboots.

To start and initialize the daemon, the LKM uses the user mode helper API of the Linux kernel. After the daemon has been started, it informs the kernel about its state and then data can be transferred.

Communication During our work, we tested different communication protocols for exchanging data between kernel and user space. Netlink interfaces have the advantage that they can be easily used by kernel modules and no patching of the kernel itself is required. Furthermore, it is possible to implement callback functions on incoming Netlink messages in kernel and user space, thus avoiding polling. A drawback of the Netlink communication is reduced throughput. Nevertheless, Netlink interfaces are used in this work because a reliably bidirectional connection is more important than maximal performance.

4.2.2 Prototype Implementation

We provide a prototype implementation of our architecture and use it in the scope of full disk encryption. Figure 4.8 shows a specialized version of Figure 4.7. Our LKM registers a new cipher within the crypto API of the Linux kernel which can then be used by the device mapper *dm-crypt*. The encryption algorithm used for full disk encryption is implemented within an enclave, and thus it

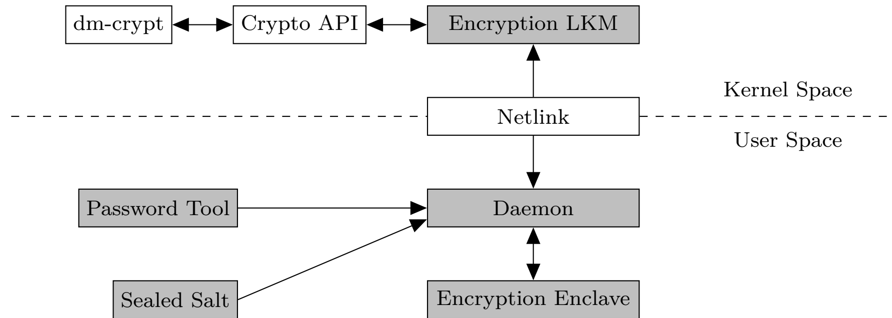

Figure 4.8: Our generic architecture used in the scope of full disk encryption.

is guaranteed that the implementation cannot be tampered with. The key used for disk encryption is securely derived within the enclave from a password chosen by the user and a device specific salt. The user password can be entered with the help of a tool which communicates with the daemon directly in user mode and the salt is stored sealed to the enclave identity. Consequently, it cannot be unsealed on a different device.

Full disk encryption is a well-known procedure to guarantee the confidentiality of sensitive data even in the event of attackers having physical access to a computer system. When the device is turned off, the data is at rest and no cryptographic keys are stored on the device. When it is turned on, however, the user must insert a passphrase to initialize the cryptographic cipher. Standard full disk encryption solutions store the password and the derived key in clear within main memory whereas our implementation stores the key securely within an enclave and protects it by logical and physical means through security guarantees provided by SGX.

Protection against Physical Attacks Besides isolation, we also provide protection against physical attacks which is especially useful in the context of full disk encryption. Currently, there are only very few disk encryption solutions which protect the disk encryption key from physical attacks on main memory. TRESOR and ARMORED (Sect. 2.4) use the debug registers of Intel and ARM processors instead of RAM to store the encryption key and all intermediate states of the encryption algorithm. Consequently, sensitive key-related data is not stored in RAM and cannot be retrieved with physical hardware attacks.

However, multiple attacks on sensitive data in main memory exist and some can be used against TRESOR and ARMORED. Cold boot attacks on RAM allow the retrieval of encryption keys after the system is turned off [82]. Sensitive data from main memory can also be obtained using DMA attacks, for example, via Firewire [23] or USB OTG [93]. TRESOR and ARMORED generally prevent against cold boot attacks, but not DMA attacks, because the control flow of the kernel can be changed to extract keys [22].

In contrast to TRESOR and ARMORED, our SGX-based solution not only protects against cold boot attacks, but also against DMA attacks, because SGX ensures that no peripheral device is able to read enclave memory. Furthermore, the use of a sealed salt adds another factor to the encryption key generation. Even if the user password could be retrieved from main memory, it is still not sufficient to derive the actual encryption key because the salt as well as the derived key are never stored outside the enclave in clear.

Detailed Workflow The overall functionality of our implementation is spread between the LKM and the user space daemon. In the following, the initialization sequence, the key setting process, and the encryption and decryption functionality will be described.

- *Initializing LKM and Daemon*: When the kernel module is initialized, it registers a Netlink family to communicate with the daemon. Once the Netlink socket is created, it starts the daemon via the user mode helper API. The daemon then creates and starts the enclave.

- *Setting Password using the Daemon*: Because using the key setting functionality of the crypto API would leak the key or password to main memory, we provide a possibility to directly set the password using only the daemon. To this end, the daemon opens a named pipe and waits until the password is written to that pipe.
- *Deriving Disk Encryption Key*: After the password has been read from the user, the daemon loads a predefined file from disk which contains the sealed salt. The enclave checks if the sealed salt is valid and unseals it. If the sealed data is not valid, it will generate a new salt and seal it. PBKDF2 is used to finally derive the disk encryption key from the user password and the salt.
- *Establish Netlink Communication*: The daemon creates the same Netlink interface as the kernel module and sends an *initialization succeeded* message to the kernel. The kernel receives the message and registers the new cipher at the crypto API.
- *Data Encryption and Decryption*: After initialization, the encryption and decryption process is straight forward. The encrypt and decrypt callback functions of our LKM are called by the user of the crypto API. The LKM then sends a Netlink message to the daemon, which calls the encrypt and decrypt functions of the enclave. The enclave performs the requested cryptographic operation and returns the encrypted or decrypted block which is passed back to the kernel via Netlink. Finally, the kernel module copies the block to the destination given by the caller of the crypto API and returns.

Enclave Cryptography To register a new cipher within the crypto API, at least routines for encryption and decryption need to be provided. Usually also a key setting routine is required, but because we set the key using our daemon directly, we do not need to implement this routine for the crypto API.

The easiest way to provide a new cipher within the crypto API is to register a bare block cipher such as AES, because it allows the crypto API to combine the bare block cipher with all modes of operation that are already implemented in the kernel. Intel provides a cryptography library together with the SGX SDK, but it is only possible to use AES together with preselected modes of operation. We therefore decided to use a small Intel AESNI library which provides a standalone bare block cipher implementation of AES with multiple key sizes and uses AESNI, Intel's hardware AES instructions [78].

4.2.3 Correctness, Performance, and Security Analysis

In this section, our prototype implementation is evaluated regarding correctness, performance, and security.

Correctness The correctness of our implementation regarding the encryption and decryption functionality has been tested in different ways. The crypto API of the kernel provides a test manager which initializes a given cipher, sets a predefined key, and then checks encrypted and decrypted blocks with different sizes against predefined test vectors. Only if the resulting blocks are equal to the ones stored within the test vectors, the implementation is correct.

When using PBKDF2 with the user password and the salt as input, the data will be encrypted with a different key compared to the user password being directly passed to the crypto API. To still be able to test the functionality of our implementation, the enclave was temporarily modified to print the PBKDF2 derived key to the host application. Once this key was fed into the crypto API directly, we could verify that encrypted and decrypted blocks are equal and thus our implementation is correct.

Furthermore, we were able to encrypt and decrypt partitions over multiple reboots and thus verified that the key can always be derived on the computer using the same user password and sealed salt.

Test	Plain	AES	SGX
dd 100 MB block write	107.0	104.5	1.1
hdparm uncached read	110.1	113.7	1.1
hdparm cached read	13,289.5	12,004.3	1,576.7

Table 4.1: Reading and writing speeds in MB/s of different operations with plain disk access, the default AES implementation, and our implementation.

Performance We compared our implementation to the standard AES implementation of the Linux kernel in a full disk encryption scenario. In addition, we compared both variants to non-encrypted disk access. All tests have been performed on a Dell Inspiron 7559 notebook with an Intel i7-6700HQ CPU, sixteen gigabytes of main memory, and a Seagate ST1000LM024 hard drive. We used a vanilla version of the Ubuntu 15.10 distribution running Linux kernel version 4.4.7.

During the design, it became clear that the multi-layered approach will result in many context switches which decrease performance. Our performance tests show that this assumption is true. In Table 4.1, three different performance tests with our implementation, the default Linux AES implementation and unencrypted disk access is shown. Three different partitions were mounted on the same hard disk for this evaluation and 24 tests have been executed before calculating the median of the results. The Linux command line tool `dd` was used to analyze the write speeds and `hdparm` for measuring the read speeds.

In detail, `dd` was executed with the parameters `fdatasync` and `notrunc` using a block size of 100 megabytes. This results in a physical non-truncated write of one file with the size of exactly 100 megabytes. After `dd` has completed its operation, the average write speed for that operation is printed.

The tool `hdparm` was executed with both, the `-t` and `-T` option respectively. The first option performs uncached disk reads and the buffer cache is cleared before performing the read. The second option performs cached reads and displays the reading speed from the Linux buffer cache without forced disk access.

Our implementation reaches about 1% of the read and write performance with forced disk access and about 10% of the read performance using the buffer cache. To write one encrypted block to disk, the kernel must send the data block over the Netlink bus to the daemon. The daemon must enter the enclave, which is another context switch, and then the enclave is able to encrypt the block using AESNI instructions. In return, the enclave sends the encrypted block back over the Netlink bus via the daemon to the LKM. Our implementation is meant to be a proof-of-concept for demonstrating that isolating kernel components using SGX is possible and improving the performance is part of future work.

Security Our design prevents wrapped kernel components from being tampered with. Functionality moved to user space and wrapped by an enclave as shown in Sect. 4.2.1 is protected against all software attackers due to the strong security guarantees of Intel SGX. Even if the kernel is compromised, the kernel component within the enclave cannot be tampered with because accesses are prevented by protection mechanisms in hardware. Even though SGX does not support entering enclaves in ring zero, we found a way to protect ring zero functionality with SGX.

Our prototype implementation derives the disk encryption key within an enclave. At no point during the whole enclave lifecycle, the key is passed to the outside world. Furthermore, the salt is generated randomly and unknown outside the enclave as well. Consequently, any attempt to retrieve the key or the salt by physically acquiring main memory will fail.

Of course, it is still possible to physically attack a system running our prototype and, to use a copy of it as a black box to encrypt and decrypt data. For the attack to succeed, however, an attacker must obtain all of the following components of the cryptographic system:

- user password
- sealed salt
- unmodified enclave which sealed the salt
- CPU on which the salt was sealed

The user password and the salt are needed as input for PBKDF2 to derive the encryption key. The salt can only be unsealed with the very same enclave on the same CPU, because it is bound to the enclave and the CPU. Consequently, it is not sufficient to steal the user password which makes our solution practically stronger than TRESOR and ARMORED, where it is sufficient to steal the password. We also verified that the user password which at some point has to be entered by the user and sent to daemon does not reside in RAM by using the Linux Memory Extractor (LiME).

Contrary to the user password, the *sealed* salt can be retrieved through a physical hardware attack at any time. Even though, the sealed salt cannot be unsealed without possessing the actual device, it is recommended storing the sealed salt on a removable storage device such as a thumb drive and only connect it to the computer if the encrypted partition needs to be mounted. This way, the computer needs to be stolen, the password needs to be retrieved, and also the removable storage device needs to be acquired to finally be able to decrypt sensitive data.

4.2.4 Discussion

Although our solution works quite well as a prototype, performance could be improved in future versions of the implementation. There is a lot of research comparing the throughput of the Linux Netlink interface to other communication channels like, for example, *SYS V Message Queues* and *SYS V Shared Memory* [144]. It has been shown that compared to the SYS V Message Queues, the Netlink user-to-kernel communication is around 30%, the kernel-to-user communication around 55%, and the startup time around 66% slower in terms of processor cycles. SYS V shared memory has a higher throughput, but needs longer startup times than message queues.

Single block cryptography leads to small chunks of data being sent over the Netlink interface and increasing the blocksize would result in a better throughput of the system. This, however, requires implementing the modes of operation within the enclave or at least the daemon.

With the current implementation it is not possible to encrypt all partitions including the root partition of a system, because the daemon must be executed in user space and loaded from some partition first. This problem could be addressed by providing a solution where the daemon is loaded from a different location like an initialization ramdisk.

4.3 Cache Attacks on Intel SGX

We regularly see reports of kernel exploits leading to privilege escalation attacks that enable attackers to get root access to a system. At the same time, not all privileged users can be trusted, e.g., cloud providers cannot be trusted when it comes to the protection of intellectual property or privacy, and end users cannot be trusted when it comes to DRM or the protection against cracking. In consequence of the threat of root-level attacks, there is high demand for a technology like SGX that guarantees the confidentiality and integrity of applications running inside untrusted execution environments.

Intel's whitepapers about SGX discard cache-timing attacks as unpractical physical attacks [87, 10, 115], not mentioning the case of software-based side-channel attacks. Software-based side-channel attacks, however, are particularly powerful due to Intel's Performance Monitoring Counters (PMC), which are restricted to the OS kernel, and according to Intel special care needs to be taken when writing enclave code. In addition to exploiting PMC, root-level attackers have fine-grained control over enclave caches because they are able to enforce a target enclave to run on the same hyperthreading core as the spy process.

Contribution We utilize a new method of attack, namely root-level cache-timing attacks, to infer secret information from an Intel SGX enclave. As a case study, we present an access-driven cache-timing attack on AES when running inside an enclave. We used OpenSSL version 0.9.7a as AES implementation, which is already known to be vulnerable to cache-timing attacks. As the method of attacking, we implemented Neve and Seifert's elimination method [122], which is already known to break vulnerable AES implementations. To the best of our knowledge, however, we are the first practically demonstrating that Intel SGX enclaves are vulnerable to cache-timing attacks. In detail our contributions are:

- We describe a root-level cache-timing attack relying on CPU pinning, influencing the hyperthreading affinity, and accessing Intel Performance Monitoring Counters (PMC). CPU pinning as well as accessing PMC are only feasible with superuser privileges and hence, have not yet appeared in the literature for non-SGX scenarios.
- For our case study, we implement a cache-based *Prime&Probe* algorithm that is able to identify memory locations accessed by enclave code on cache line granularity. We support our measurements by Intel PMC within the attacker thread and run both the enclave and the attacker thread on the same hyperthreading core. Thus, enclave and attacker thread also share their memory.
- Using an implementation of Neve and Seifert's elimination method, we are able to extract the AES secret key of an encryption application in less than 10 seconds. We are able to deterministically derive the secret key for every run of the vulnerable application when investigating 480 encrypted blocks on average to reduce the number of key candidates to one.

4.3.1 Attacker Model

We consider all software running on the target system untrusted, including low-level privileged software such as the operating system or even a hypervisor. Based on SGX's trust model, it is reasonable to assume that the OS is controlled by the attacker, because that is exactly what SGX should protect against. So the attacker is allowed to manipulate operating system functionality, influence the scheduling and pinning of specific threads to specific CPUs, and observe or interrupt running processes. Furthermore, an attacker is also allowed to change functionality of the application that is about to be attacked as long as the changes are not within enclave code. In summary, we assume an attacker with local superuser privileges, but no physical access or capabilities to manipulate hardware or parts of the CPU package itself. Above that, the attacker is assumed to be unable to break cryptographic primitives.

4.3.2 Gladman Implementation of AES

We will focus our attack on the Advanced Encryption Standard (AES), which has been standardized by the National Institute of Standards and Technology (NIST) in 2001 [55]. AES is a subset of the *Rijndael* algorithm, which has been designed by Vincent Rijmen and Joan Daemen [43] and chosen in a public selection process called out by NIST in 1997.

The Rijndael algorithm is a round-based, symmetric cipher operating on block and key sizes which are a multiple of 32 bits. In the AES version this was reduced to a fixed block size of 128 bits, as well as key sizes of 128, 192 or 256 bits. The number of rounds is based on the key size. In the case of a 128-bit key, 10 rounds are performed, with 192 bits the number of rounds is rising to 12 and with 256 bits 14 rounds are performed. In the following we focus on the 128-bit version of AES. Nevertheless, the key size and the corresponding number of rounds does not affect our attack. Before any encryption or decryption operation can be performed, the AES key needs to be expended such that a 128-bit key for every round is available. While we do not describe the AES *key expansion* algorithm here, we want to note that it is highly redundant, i.e., one round key is sufficient to calculate all other round keys including the AES key itself.

On a high level the AES-algorithm gets a 128-bit input called plaintext p, as well as a 128-bit secret key k. The algorithm is operating on an internal state s using a round key $k^{(r)}$ within each round. The state s is transferred to the output called ciphertext c after the last round. s, p, c, and k can each be represented as a 4x4 byte matrix. Each round transforms the state by applying different operations such as *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey* while in the last round, the *MixColumns* step is omitted. *SubBytes* replaces every byte in the state matrix with its modular inverse over GF $\left(2^8\right)$, *ShiftRows* performs a cyclic rotation of the bytes in each row of the state matrix, *MixColumns* performs a linear transformation over the finite field GF $\left(2^8\right)$ which can be represented as a matrix multiplication with a fixed matrix, and finally *AddRoundKey* combines each byte of the state matrix with the corresponding byte of the round key with a simple XOR.

A closer look tells us that all steps apart from *AddRoundKey* are not dependent on the key and can be precalculated. That's what Gladman's implementation [63] does. Suppose the initial state is represented by the following matrix:

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,0} & p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,0} & p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix}$$

The state after one round can then be expressed as follows:

$$\begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S[s_{0,j}] \\ S[s_{1,(j+1) \bmod 4}] \\ S[s_{2,(j+2) \bmod 4}] \\ S[s_{3,(j+3) \bmod 4}] \end{bmatrix} \oplus \begin{bmatrix} k^{(r)}_{0,j} \\ k^{(r)}_{1,j} \\ k^{(r)}_{2,j} \\ k^{(r)}_{3,j} \end{bmatrix}$$

Every S-box value is multiplied by fixed values during the *MixColumns* step and thus, the results can be precalculated for all 256 possible byte values and then saved within lookup tables. We need four tables with 256 entries of 4 byte size each, taking 4 kilobyte of space altogether. For the last round we need a separate table because of the missing *MixColumns* step. Let us define the lookup tables as follows:

$$T_0[x] = \begin{bmatrix} S[x]*02 \\ S[x] \\ S[x] \\ S[x]*03 \end{bmatrix} \qquad T_1[x] = \begin{bmatrix} S[x]*03 \\ S[x]*02 \\ S[x] \\ S[x] \end{bmatrix} \qquad T_2[x] = \begin{bmatrix} S[x] \\ S[x]*03 \\ S[x]*02 \\ S[x] \end{bmatrix} \qquad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x]*03 \\ S[x]*02 \end{bmatrix}$$

One round of AES can then simply be calculated with four table lookups and XOR operations. In detail, the state is calculated as follows:

$$\begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} = T_0[s_{0,j}] \oplus T_1[s_{1,(j+1) \bmod 4}] \oplus T_2[s_{2,(j+2) \bmod 4}] \oplus T_3[s_{3,(j+3) \bmod 4}] \oplus \begin{bmatrix} k^{(r)}_{0,j} \\ k^{(r)}_{1,j} \\ k^{(r)}_{2,j} \\ k^{(r)}_{3,j} \end{bmatrix}$$

Because of the missing *MixColumns* step, the last round definition is much simpler and uses a separate table:

$$T_4[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \\ S[x] \end{bmatrix}$$

The state after the last round which is equivalent to the ciphertext is then computed as follows:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = (T_4[s_{0,j}])_{\vec{1}} \oplus \left(T_4[s_{1,(j+1) \bmod 4}]\right)_{\vec{2}} \oplus \left(T_4[s_{2,(j+2) \bmod 4}]\right)_{\vec{3}} \oplus \left(T_4[s_{3,(j+3) \bmod 4}]\right)_{\vec{4}} \oplus \begin{bmatrix} k^{(10)}_{0,j} \\ k^{(10)}_{1,j} \\ k^{(10)}_{2,j} \\ k^{(10)}_{3,j} \end{bmatrix}$$

Where $(\cdot)_{\vec{i}}$ denotes a vector with all components apart from the i-th component set to zero. Thus a simpler form of the computation would be:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} S[s_{0,j}] \\ S[s_{1,(j+1) \bmod 4}] \\ S[s_{2,(j+2) \bmod 4}] \\ S[s_{3,(j+3) \bmod 4}] \end{bmatrix}$$

Nevertheless, the first variant has been chosen for the AES implementation within OpenSSL as it allows all operations to be performed on 32-bit words which is faster than a number of single byte operations especially when it comes to memory operations. The fourth table with all four lookups is crucial to our attack when we are applying Neve and Seifert's elimination method to extract the last round key $k^{(10)}$ based on cache access patterns for T_4.

4.3.3 Neve and Seifert's Elimination Method

We base our work on the access-driven approach by Neve and Seifert [122] which describes an attack against the last round of AES. For our attack, we assume we can isolate the last round. The general setup is a known ciphertext attack based on the standard Gladman implementation. Thus, the ciphertext byte $c_{i,j}$ depends on the output of the T-table T_4 and the last round key byte $k^{(10)}_{i,j}$:

$$c_{i,j} = k^{(10)}_{i,j} \oplus \left(T_4[s_{i,(i+j) \bmod 4}]\right)_i$$

Where $(\cdot)_i$ denotes the i-th byte within $(\cdot)$. Because XOR is self-inverse, the key byte $k^{(10)}_{i,j}$ can be obtained as follows:

$$k^{(10)}_{i,j} = c_{i,j} \oplus \left(T_4[s_{i,(i+j) \bmod 4}]\right)_i$$

Because the AES key schedule is redundant, the bytes $k^{(10)}_{i,j}$ are sufficient to obtain the bytes $k_{i,j}$ and thus the encryption key k. However, we need knowledge about the ciphertext and the result

of the table lookups for T_4. For our attack we assume to have access to the ciphertext (known ciphertext attack) and retrieve the result for T_4 with the cache-timing attack.

The *Prime&Probe* method works on the last round as one entity and consequently we have to find out which accessed cache line correspond to which byte of the ciphertext as well as which byte within this line was accessed. Of course it is very difficult to say what byte has been accessed exactly and that is why Neve and Seifert proposed their elimination method where certain candidates for each key byte are excluded:

$$k_{i,j}^{(10)} \notin c_{i,j} \oplus \neg[T_4 \text{ outputs}]$$

Here $\neg[T_4 \text{ outputs}]$ refers to all T_4 byte values contained within non-accessed cache lines. The basic idea is to discard all candidates for $k_{i,j}^{(10)}$ which would map to not accessed cache lines and thus cannot be candidates given the corresponding ciphertext byte $c_{i,j}$. To be more precise, let Φ be the set of all possible key byte values. When the operation starts Φ contains all possible byte values, i.e., $\Phi = \{i : 0 \leq i \leq 255\}$. The goal is to reduce those values such that only the original key byte $\Phi = \{k_{i,j}^{(10)}\}$ remains. To this end, we issue a couple of encryptions with different inputs while monitoring the cache activity. For each encryption we can eliminate some values $\Phi_e = c_{i,j} \oplus \neg[T_4 \text{ outputs}]$ and thus, reduce the size of the set Φ by assigning $\Phi \leftarrow \Phi \setminus \Phi_e$. For more details and examples on how the elimination method works, we refer to the original paper [122].

4.3.4 Cache Architecture

Intel's memory and cache architecture is not publicly disclosed, nevertheless, most details are required to be known by other hardware components or the operating system and are therefore de facto publicly available. The cache organization depends on the particular CPU architecture and model. Our attack has been performed on an Intel i7-6700HQ CPU and consequently, the cache architecture described in this section is based on the *Skylake* architecture. The cache structure described in this section, however, is applicable to other Intel chips as well.

A typical Intel CPU has three cache levels, the L1-, L2-, and L3-cache. While access to main memory has a latency of about 50 to 150 ns, the L1-cache only requires a delay in the order of 0.3 ns, which results in a speedup of 2-3 orders of magnitude [155]. Higher-level caches are typically able to store more data, but also have a higher latency. This resembles the compromise between speed and size of modern cache implementations. While smaller sized caches provide lower access latencies, they consume more power and space on the circuit board. Bigger sized caches can hold more data and provide smaller latencies for a bigger amount of addresses, being less power consuming but also orders of magnitude slower. Those trade-offs are balanced by a multilevel setup, combining the small fast caches with bigger back end caches.

Today's Intel CPUs use a set-associative cache for all cache levels. Those types of caches are split into S cache sets. Each set consists of a number of cache lines W and each of them has a size of B bytes. Thus, the total size of the cache can be calculated as $S \cdot W \cdot B$ bytes. Whenever a byte is fetched from memory, the corresponding B bytes are loaded into a cache line according to the alignment, performing some kind of prefetching.

The designated cache set of a particular cache line is determined by its address. This is done by splitting an address A, consisting of n bits, into t tag-bits, s set-bits and b offset-bits as shown in Figure 4.9. The least significant $b = log_2(B)$ bits are used to address a byte inside a cache line. The following $s = log_2(S)$ bits are used to determine the particular cache set. The remaining $t = n - (b + s)$ most significant bits are stored as a tag inside the cache's meta data and used as the unique identifier of a cache line [30].

If a memory access occurs, the particular set is identified with the set bits s and all tags inside are compared with the tag bits t. If the correct line was found, the byte at offset b is returned, otherwise the same procedure is repeated on the next level. If a cache set has n cache lines per cache set, the cache is called n-way set associative cache. In this case n cache lines with the same

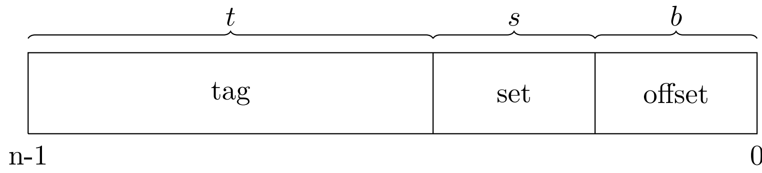

Figure 4.9: Address partitioning for set-associative caches.

set bits can be stored concurrently. A one-way set associative cache is called direct mapped cache. Direct mapped caches reduce the latency caused by comparing tags at the cost of an increased number of evictions. A cache with only one set is called fully associative cache.

Our Intel i7-6700HQ CPU has four physical cores, each one has a separate L1- and L2-cache, as well as a shared L3-cache. The L1-cache is split into data and instruction cache, both have a size of 32 KB and are 8-way associative with 64 byte cache lines. This means a 64-bit address is split into $b = log_2(64)$ bit $= 6$ bit offset, $s = log_2(32 \text{ KB}/(64 \text{ B} \cdot 8))$ bit $= 6$ bit set and $t = 64 \text{ bit} - (b + s) = 52$ bit tag.

For this specific cache topology it does not matter whether the cache is indexed physically or virtually, because 4096 byte pages require an offset of $log_2(4096)$ bit $= 12$ bit to address bytes within one page. These 12 bits correspond to the cache set s and the cache line offset b and thus, s and b keep the same value regardless of whether the physical or virtual address is used for indexing within the cache.

The L2-cache is unified for data and instructions, is 4-way associative, and has a size of 256 KB while the L3-cache is 12-way associative with a total size of 6 MB. Because our attack only relies on the L1-cache, we do not need more detailed information about the L2- and L3-cache.

4.3.5 Cache Priming and Probing

In order to gain the knowledge about accessed memory addresses on cache line granularity we utilize the *Prime&Probe* method [126]. While the theory behind the *Prime&Probe* algorithm is fairly simple, associative caches introduce another challenge to this problem, because Neve and Seifert describe their attack on a direct mapped cache (Sect. 4.3.3). Consequently, they are able to trace each miss back to the evicted cache line.

Unfortunately, direct mapped caches are not used by modern Intel CPUs and we have to take a closer look at the placement of the Gladman T-tables in memory. In the case of our 8-way associative L1-cache we have 8 cache lines per cache set. This has two effects on our algorithm. First of all two addresses having the same set bit, but belonging to different cache lines, can be stored concurrently in the same cache set. Ultimately, this makes an access to one of them indistinguishable from the other, on the cache set level of our probing step. Secondly, in the priming phase we have to access each cache set 8 times to be sure we filled the whole cache and force an eviction of our own data by the victim application, as the cache line position cannot be predicted. In fact, it could be, if the eviction algorithm was known and deterministic, but unfortunately this is not the case. If two cache lines of the T_4 table were stored in the same cache set, they would be unusable for the elimination method, because if one of them is accessed, the access would be indistinguishable from an access to the other cache line.

A single Gladman T-table has a size of 1 KB and our L1-cache has a cache line size of 64 B, so placing the T_4 table in the cache requires 16 cache sets assuming that there are no collisions, i.e., the sets are different for each cache line. A collision, however, is highly unlikely, because the T-tables lie contiguously within memory. Furthermore, in our 8-way associative 32 KB L1-cache there are $32 \text{ KB}/(64 \text{ B} \cdot 8) = 64$ cache sets, hence the table fits completely into the cache. Consequently, we can use the information about an access to a cache set similarly compared to an access to a cache line for direct mapped caches. Nevertheless, we cannot predict which cache line in each set

holds the T-table data. Thus, our priming algorithm has to access every cache set 8 times using particularly aligned addresses to fill every line and eventually detect every single eviction.

This leads to the next challenge we need to solve: Identifying an eviction in the probing phase. The original idea of Neve and Seifert is to distinguish an evicted line from a present one by the timing difference between an L1 and L2 access. The Intel Performance Monitoring Counters (PMC) [167], however, give us even more accurate information compared to relying on any time-stamp counter. With the PMC, Intel introduced a high-level interface for developers to gain access to several CPU internal performance metrics. This data is obtained from the so-called Performance Monitoring Units (PMU) and contains information about L1-, L2-, as well as L3-cache misses. The shortcoming of using the PMC to detect cache misses is the fact that the counters need to be started from privileged kernel space, i.e., ring zero, however they can be read from user space afterwards.

Local attackers with superuser rights are covered by the SGX attacker model and thus, an attacker is allowed to run ring zero code by simply loading a kernel module. We did not write the PMC driver ourselves, but instead we use the Loadable Kernel Module (LKM) from Agnar Fog's PMC-based performance test suite [56]. After starting the PMC with the desired counters we can read the current count with the `readpmc` assembler instruction from non-enclave user space with the desired PMC identifier as argument. This provides us with a perfect measurement for L1-cache hits or misses. Using the PMC, our probing algorithm works as follows:

1. Read the PMC count using the `readpmc` instruction.
2. Force the CPU to serialize all instructions by dispatching a dummy `cpuid` instruction.
3. Access the desired cache line using a particularly aligned address.
4. Serialize all instructions again (dummy `cpuid` instruction).
5. Read the PMC count (`readpmc` instruction) and return the difference to the last read PMC count.

If the difference is greater than zero, the particular cache line was evicted from the L1-cache, otherwise it was present. This procedure is repeated 8 times again to be able to catch all evictions for a given cache set. If one difference is greater than zero, it can be concluded that the corresponding cache line for T_4 has been accessed. The serializations with the `cpuid` instruction are necessary to prevent out-of-order execution in modern CPUs which would tamper with our results.

Summarized, the PMC provide a perfect probing functionality and complete our *Prime&Probe* implementation. Nevertheless, it is worth mentioning that every access to variables mapped to the same cache sets as the T_4 table would introduce noise to our measurements, because they could not be distinguished from the cipher's T-table accesses. Because our victim enclave provides perfect timing conditions without any additional memory accesses, this is currently not a problem. In more complicated scenarios, our probing solution would work without any modifications but require a higher number of encryptions. The reason is, that probing always leaks some information about accessed T-table indexes unless the whole T_4 table is overlapped by other variables within the cache. Consequently, there are always some key byte candidates which can be eliminated by Neve and Seifert's method.

Elimination Method One advantage of Neve and Seifert's elimination method is that it is resilient against false positives. The probing phase provides us with information about the accessed cache lines during the last round. Theoretically we could perform a probing request successively to each cache set one after the other. This way we would receive the access pattern of a ciphertext with a single encryption. Unfortunately, we encountered that accessing certain cache lines of the T-table has influence on other subsequent cache lines of the T-table, which get preloaded in their particular set. Consequently, we decided to probe each cache set separately for each ciphertext, requiring 16 encryptions per ciphertext until we receive the whole access pattern.

While this would be sufficient for the elimination algorithm, we additionally decided for fault-tolerance. In addition to probing each cache line once, we provide a possibility to repeat the overall

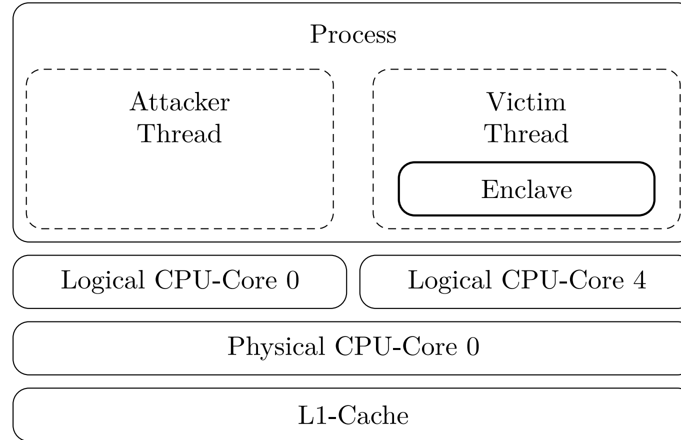

Figure 4.10: Our attack setup consisting of one application with two threads pinned to two logical CPUs sharing the same physical core and L1-cache.

process a certain number of times and to count each line evicted more often than a certain threshold as accessed line. The repetition rate and the threshold are configurable and can be adapted by the attacker. If each cache line is only probed once, i.e., no repetition, no threshold is used as well. While smaller repetition rates result in less encryptions needed to receive the secret key, higher ones cause less fluctuation in encryptions needed per ciphertext. We decided to set a repetition rate of 20 and a threshold of five as default values if fault-tolerance is enabled.

Furthermore, we observed that some of the cache lines for our measurements were evicted regardless of whether they have been accessed or not. Although these lines were unusable for gaining information about the access pattern, we retrieved enough information due to the other lines, such that this does not affect the result. In fact, resistance to false positives is a big advantage of the elimination method in general.

Whenever a cache line has been evicted less often than the threshold value and is consequently counted as not accessed, all corresponding key byte candidates are removed from the remaining possibilities for every key byte according to Neve and Seifert's algorithm. This is repeated until only one value remains for each byte which immediately leaks the secret last round key. Afterwards, the fact that Rijndael's key schedule is reversible is used to calculate the secret key based on the last round key.

4.3.6 Attack Setup

Our implementation has been developed for Ubuntu Linux (14.04 LTS) using the official Linux SGX SDK provided by Intel. We present an overview of our attack setup before describing the design decisions behind our implementation including decisions specific to a root-level attacker scenario.

Overview Our attack design consists of one application with two threads pinned to two logical CPUs sharing the same physical core as shown in Figure 4.10. The victim thread runs an OpenSSL AES implementation which is known to be vulnerable to cache-timing attacks within an SGX enclave. The encryption key is generated within this enclave and never leaves it. The attacker thread executes regular ring 3 code in non-enclave mode. It takes care of priming, probing, and the elimination by reading and comparing PMC values.

Furthermore, communication is exclusively performed using shared memory, no process context switches occur, and the enclave is never exited. This setup combines specific possibilities of root-level attackers within one single root-level cache-timing attack. It demonstrates that long-term secrets within enclaves that would usually never be exposed to the non-enclave world can be obtained

using cache-timing attacks. Our implementation aims to serve as a case study and shows that software developers writing enclave code need to take particular care when it comes to crypto implementations.

Communication with Shared Memory To demonstrate cache-timing attacks, often a classic client-server application is deployed in such a way that the two involved processes are pinned to the same CPU core. When porting this setup to SGX enclaves we would need to handle the communication outside of the enclave, because certain instructions such as system calls are not allowed within enclaves. System calls, however, are needed to communicate between the client and the server such that the enclave has to be exited first. Unfortunately, this setup implies using *ECALLs* and *OCALLs* which introduce too much noise, at least when used with the official Intel SGX SDK. In fact, they evict the whole T_4 table when triggering the encryption function after the priming step which makes the detection of accessed cache lines in the probing phase impossible.

When designing our victim application, we came to the conclusion that it is not possible to perform a cache-timing attack solely focusing on the L1-cache, whenever an ECALL or OCALL is included between the measurements. This left us with two options. First, we could focus on higher cache levels. Second, we can build our victim application in a way that it does not have to leave enclave mode during the measured section. While the first option is an interesting path to pursue in future work, we decided to continue with our attack focused on the L1-cache. We solved the communication problem inside the enclave mode by allocating and using shared memory and utilizing control flags. Our attacker process copies the particular plaintext to the shared memory and triggers the encryption process by setting a particular flag. The enclave is actively waiting for this flag to be set and immediately starts the encryption process. Afterwards, it sets a flag itself to signal the end of the encryption. The attacker already waits for the encryption to finish and reads the ciphertext from the shared memory. Finally, it continues with the next plaintext. This setup allows the attacker to perform the priming right before and the probing right after the encryption. Shared memory is a well-known and fast communication technique which can be used between SGX enclaves and unprotected code as well. The control flags are used to reduce the synchronization effort. Finding the right timing for a regular encryption loop, would lead to the same results, but complicate our setup.

Attacker and Victim Thread Using two separate processes for attacker and victim still does not allow us to observe the cache activity. Whenever a process switch occurs, the CPU has to replace all the memory abstractions of the current process with the ones of the next process. This includes the page tables which are necessary for translating virtual addresses to physical ones. In order to set the correct page tables, the CPU has to update the `CR3` register which contains the base address of the current paging hierarchy. Unfortunately, changing the page tables invalidates all information connected to virtual addresses. Consequently, a write to `CR3` will trigger a Translation Lookaside Buffer (TLB) flush, as well as flushing all other virtually tagged caches. In our scenario, the process switch causes too much noise such that the L1-cache would be flushed every time we switch from the attacker to the victim or vice versa occurring after the priming step and thus, destroying our probing results.

Consequently, we changed our scenario to have the attacker in the same process as the victim, but within two different threads. To this end, we pinned two kernel-level threads on the same core using the *PThread* threading library. In our implementation the attacker enters the victim enclave within a separate thread in the process and again uses shared memory for communication. Additionally, we eliminated every system-call in between the encryption and probing phase, forcing both parties to perform active waiting. Note, however, that in our SGX scenario the enclave contents are the only parts being protected, and hence, manipulating the surrounding application or the operating system is covered by the strong attacker model of SGX.

Hyperthreading Having two threads pinned on the same CPU did not provide us with the measurements we expected either. The reason is that whenever the kernel switches between the

execution of the two threads, the CPU has to enter or leave the victim enclave which, although less expensive than an enclave exit triggered by software through an ECALL, still causes enough evictions in the L1-cache to evict the whole T_4 table and to ultimately destroy our probing measurements. In fact, this leads to a contradiction: In order to work on the same L1-cache we need to operate both threads on the same core and simultaneously they need to operate on different cores to omit the enclave exit. At a first glance, it seems impossible to satisfy both requirements, but luckily modern Intel CPUs include a feature which helps to indeed satisfy both requirements, called *hyperthreading*. With hyperthreading usually two logical CPUs per physical core are offered, for example, by duplicating the Arithmetic Logic Unit (ALU) but sharing everything else including the L1-cache. Hence, we decided to pin the attacker thread and the victim thread running in enclave mode to two different logical CPUs which are mapped to the same physical core. This way enclave exits are omitted while still sharing the same L1-cache between both threads. With superuser privileges controlling the hyperthreading affinity, i.e., pinning the threads to appropriate logical CPUs, is easily possible, for example, by using the `sched_setaffinity()` system call.

Enclave Interaction To plausibly demonstrate that our implementation is able to extract unknown secrets from enclaves, our victim enclave generates an AES key with the help of `sgx_read_rand()` provided by the SGX SDK *within* the enclave. Additionally, the victim enclave contains functions to safely store and load this key afterwards by utilizing SGX's sealing mechanism. Furthermore, it contains a function for encrypting single blocks as well as an endless encryption loop. In the actual attack, the attacker thread creates random plaintext and triggers the encryption inside the enclave, which runs just until the start of the last round and waits for the next flag. In the next step it issues its priming algorithm which fills all the T_4 table's cache lines. After priming, the attacker thread sets the particular flag, in order to signal the execution of the last round to the enclave. When the encryption has finished, it continues the attack, by issuing the probing algorithm and feeding the results to the elimination method. Those steps are repeated until all key bytes have been reduced to one remaining possibility. Finally, the attacker thread verifies the result by creating a decryption key, which is used to decrypt a string encrypted by the enclave beforehand. Note, that the AES key is never leaving the secured enclave and without the attack would not be accessible besides within the victim enclave itself.

4.3.7 Performance Analysis

All attacks have been executed on a notebook with a 2.60 GHz Intel Core i7-6700HQ CPU and 16 GB of RAM, running Ubuntu Linux 14.04 LTS (Trusty Tahr). Each core has a 32 KB 8-way set-associative data L1-cache, a 256 KB 4-way set-associative L2-cache, as well as a shared 6 MB 12-way set-associative L3-cache.

In order to measure the performance of our attack, we take three different scenarios into account. For each scenario, we perform 5000 attacks with different keys and let each attack run until the respective key can be retrieved. To evaluate the performance we measured for each attack the overall time required to leak a secret key, as well as the amount of required elimination rounds, i.e., how many encryptions and thus ciphertext blocks are needed.

For the first scenario, we perform the eviction detection with a threshold of 25% by repeating the probing step for every cache line 20 times and handling every line which has been evicted more than five times as an accessed one. On average, this approach requires 30 elimination rounds to reduce the number of possible key candidates down to one. With each round requiring $20 \cdot 16 = 320$ encryptions, the attack needs an average number of 9600 encryptions to leak the secret key. On our target system this takes an average time of 5 minutes and 29 seconds. Figure 4.11 shows the average number of remaining key candidates after each elimination round. It can be seen that the number of possible key candidates decreases almost exponentially. Furthermore, the minimum and maximum number of remaining candidates shows that the average number of candidates generally stays close to the minimum. Although the elimination method is not prone to false positives, the threshold helps to prevent arbitrary evictions to be included in the result. Consequently, in this

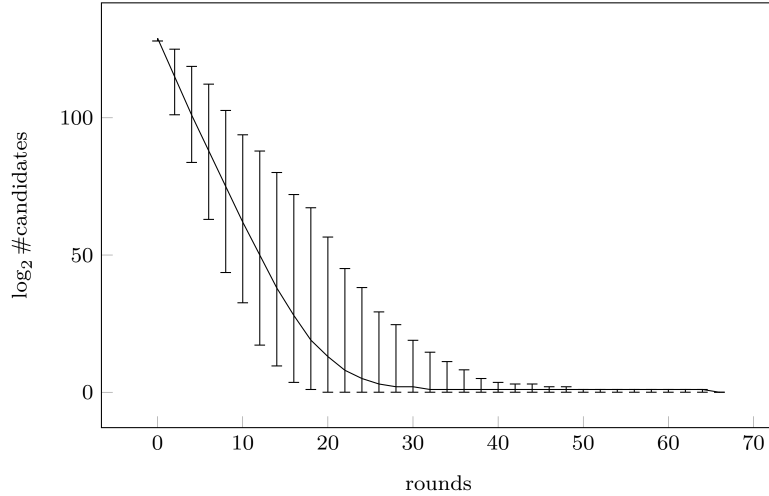

Figure 4.11: Number of remaining key candidates after each elimination round with 25% threshold and on-demand CPU frequency scaling between 800 MHz and 2.60 GHz (default setting).

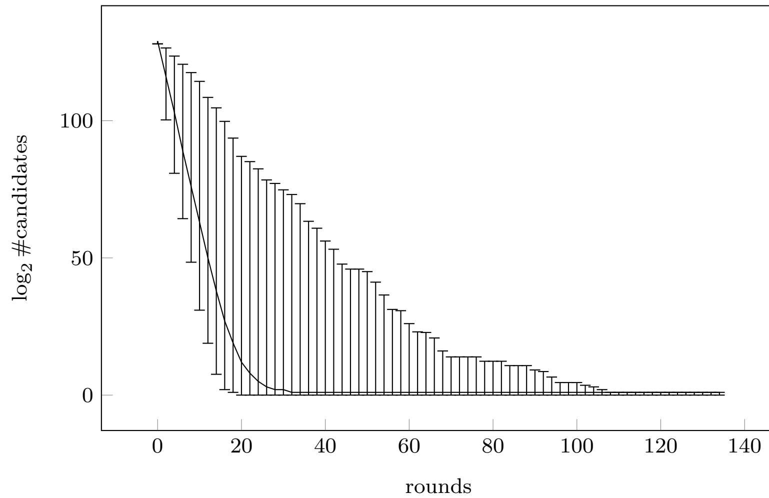

Figure 4.12: Number of remaining key candidates after each elimination round without threshold and on-demand CPU frequency scaling between 800 MHz and 2.60 GHz (default setting).

scenario the deviations for the number of remaining candidates per elimination round are rather low compared to the next two scenarios.

For the second scenario, we measure the performance without any threshold, i.e., we only probe once per elimination round. Interestingly, on average this setup requires the same amount of elimination rounds, but due to the reduced probing frequency, only $16 \cdot 30 = 480$ encryptions are necessary for all elimination rounds together. Consequently, the time to leak the secret key is drastically reduced down to only 10 seconds on average. When comparing the results from Figure 4.12 to the previous scenario, however, we observe a significant increase in the deviation of remaining candidates per elimination round. This means that it is more likely that without a threshold a larger amount of elimination rounds is needed for certain secret keys. Nevertheless, the average number of elimination rounds remains roughly constant which means that faster results are usually achieved without threshold.

Finally, for the last scenario, we examine the impact of CPU frequency scaling on our attack. On our target system on-demand frequency scaling is the default and was thus used for the first two scenarios. We now manually set the CPU frequency to the maximum frequency of 2.60 GHz and

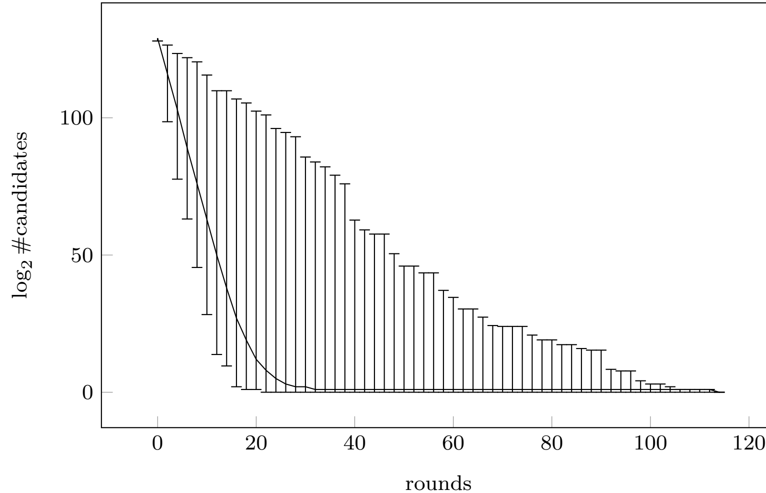

Figure 4.13: Number of remaining key candidates after each elimination round without threshold and maximum CPU frequency of 2.60 GHz.

disable on-demand scaling. Our results show that CPU frequency scaling does not significantly affect the time needed to obtain the secret key as shown in Figure 4.13. An attack without on-demand frequency scaling requires an average of 30 elimination rounds and takes 9 seconds of time which is slightly faster than the previous attack.

Note that the time necessary to obtain the secret key could be even further decreased by aborting the attack before the number of remaining key candidates is reduced to only one. Instead, the full key could be brute-forced for a certain number of remaining key candidates, thus saving elimination rounds. For our scenarios, for example, after 20 elimination rounds it should be feasible to brute-force the key. Besides being faster, aborting earlier also has the advantage that less elimination rounds and thus less encryptions are needed which might be helpful for attack scenarios where only a limited number of encryptions is allowed.

4.3.8 Practicability Analysis

We described a cache-timing attack against AES running within an Intel SGX enclave. Our implementation relies on certain possibilities which are only available for root-level attackers. In this section, we want to discuss the feasibility of such root-level cache-timing attacks in a real world scenario.

Anti Side-Channel Interference The victim enclave is currently run in debug mode. While the functionality SGX provides in debug mode is similar to production mode, certain security features are disabled. In production mode, Intel provides the possibility to suppress performance monitoring activities by entering a so-called *opt-out enclave* (Sect. 43.6 of the Intel SDM [89]) which sets the Anti Side-Channel Interference (ASCI) bit within the `IA32_PERF_GLOBAL_STATUS` MSR. If this bit is set, the PMC are not accumulated normally, but instead activities by the enclave are not counted. While we could not test the effect of this bit on our attack, we are confident that it does not affect the attack. The reason is that we first prime the cache, then execute the last round of AES within the enclave, and finally probe the cache from the attacker thread *outside* of the enclave. The PMC are only used during the probing phase, i.e., the last phase, and only from the attacker thread (Sect. 4.3.5). Consequently, even if the ASCI bit is set and the cache activities by the victim enclave are not counted, the activities by our attacker thread are counted. This is based on the assumption that for two threads running in parallel on a single core (hyperthreading case) the ASCI bit does not affect counting activities of the core running non-enclave code.

Enclave Interface The interface of our victim enclave currently allows us to separately stop and resume the encryption process just before the last round of AES starts. Although such an interface is a simplification and would be different for a real-world application which usually would directly execute all ten rounds after the invocation of one single encryption routine, we are confident that our attack is still feasible.

Gullasch et. al. [79] presented a solution which aims at isolating the last round by triggering precise interrupts with the help of a modified scheduler. Because modifications of the operating system kernel are within the root-level attacker model, changes to the scheduler are possible and the only question is how easily they could be applied to the running kernel by an attacker. At least offline modifications to the kernel, however, are perfectly possible. Furthermore, an attacker would need to perform more encryptions if the last round is not perfectly isolated, but for an enclave storing a long-term sealed secret key, for example, the number of encryptions required to extract that key are not necessarily a restriction.

Depending on the particular cipher implementation used within a given real-world application and the corresponding enclave interface, an appropriate attack vector has to be chosen. For other ciphers than AES, for example, Neve and Seifert's elimination method cannot be applied. The powerful possibilities of root-level attackers such as CPU pinning, accessing Intel PMC, or influencing the hyperthreading affinity, however, are applicable to many attacks. If an attack, for example, targets higher level caches, i.e., L2 or L3, a root-level attacker could disable other cores sharing the same cache to reduce noise.

Cipher Implementation For our implementation, we use a Gladman AES implementation adopted from an old version of OpenSSL (version 0.9.7a) known to be vulnerable against certain types of cache-timing attacks. As our goal is to demonstrate that SGX enclaves in general are indeed vulnerable to cache-timing attacks, this choice is reasonable. The practicability of a cache-timing attack against enclave code depends on whether the cipher implementation used within the enclave is actual vulnerable to such kinds of attacks.

Recent state-of-the-art AES implementations have long been fixed and are not vulnerable anymore to simple cache-timing attacks. For example, the current OpenSSL release (version 1.1.0) [8] utilizes the AESNI x86 instruction set extension [78] which provides a secure AES hardware implementation. It is now the recommended method of implementing AES on the x86 platform, because the AESNI hardware implementation is considered secure against all known types of software attacks and ultimately prevents any type of cache-timing attack.

Interestingly, the AES implementation within the official Intel SGX SDK for Linux [9] is not making use of AESNI and implements AES in software instead, even though AESNI instructions would be usable within enclaves. The software implementation used is almost a textbook version of AES without the use of Gladman tables. Intel manually included instructions, though, which perform an access to all S-Box indices in each round causing cache evictions for all S-Box entries. Consequently, Neve and Seifert's elimination method is not applicable to the AES implementation from the SGX SDK either, because there are no cache lines which have not been accessed and would allow to eliminate key byte candidates. However, the question is raised, why Intel did not just use AESNI when they apparently were aware of the threat through cache-timing attacks.

Consequently, the presented attack would not be feasible against those two implementations which shows that applications *can* be protected against cache-timing attacks by using secure implementations. Developers have to do so, however, *manually.* Custom cipher implementations are not rare at all and thus, there are good chances of a custom implementation ending up within an SGX enclave and then being vulnerable like we have shown. With the powerful possibilities of root-level attackers, cache-timing attacks are more likely to succeed and therefore particular care needs to be taken by enclave developers when it comes to crypto implementations.

4.3.9 Discussion

Our implementation solely focused on the Gladman implementation of AES. Although AES currently is probably the most important symmetric cipher, it would be interesting to have a look at a wider range of implementations, including implementations for other ciphers, and examine whether cache-timing attacks against those implementations are feasible.

Besides cache-timing attacks other side-channels might be promising when targeting Intel SGX as well. In particular, it might be worth considering hardware attacks such as power analysis or frequency glitching to extract secrets regarding Intel SGX itself and not only secrets belonging to software enclaves. Such attacks, of course rely on a different attacker model including a physical hardware attacker.

Although the performance of our implementation is already good enough for practical use cases, it could be further improved. Instead of eliminating all key candidate bytes until only one possibility for each byte remains, the elimination process could be aborted and the remaining key bytes could be brute-forced instead.

4.4 Summary

We presented an authentication protocol Stark that *mutually* authenticates the computer and the user by utilizing a trusted external device together with the TPM built into virtually every modern computer. Our protocol prevents the user from entering his disk encryption password into a forged password prompt on a potentially compromised machine. We also presented an implementation Mark of the protocol which inherits all the security properties from Stark and furthermore is operating system independent, resists towards cold boot attacks and provides plausible deniability.

We then showed a concept to enforce isolation for kernel components by moving parts of their functionality to user mode and wrapping it within Intel SGX enclaves. To this end, we require a daemon running in user mode communicating with an LKM in kernel mode. In addition to the LKM interacting with the kernel API, the daemon is also allowed to directly provide functionality to user space.

Finally, we demonstrated an access-driven cache attack against an AES implementation running within an SGX enclave. In particular, our implementation is able to derive the secret key of an AES Gladman implementation taken from OpenSSL. The attack is able to deterministically retrieve the key for every single run within an average time of less than 10 seconds.

5 Software-based Memory Encryption

As previously shown, hardware-based trusted computing architectures provide a variety of possible applications ranging from the protection of software modules for embedded devices over secure full disk encryption to the isolation of kernel components. However, in cases where hardware-based architectures are not available or cannot be used, software-based solutions need to be provided.

SGX, for example, also is capable of encrypting physical memory of software running within enclaves. However, enclaves are restricted to a static size and cannot dynamically grow once they have been created. Also software within enclaves needs to be specially written for SGX as the restricted enclave environment does not permit system calls. Consequently, SGX cannot provide compatibility with existing applications that should be protected against memory disclosure.

In contrast, software-based attempts to thwart reverse engineering are typically based on *obfuscation*, i.e., transformations making programs harder to analyze [40]. It is well known today that there exists both a class of programs which provably *cannot* be obfuscated [16] as well as a class of programs which *can* provably be obfuscated [163, 33]. Both classes of programs are rather small and cannot be generalized. It becomes possible to achieve perfect obfuscation (in a cryptographic sense) for programs in general only if a scheme for fully homomorphic encryption is available [59]. Since fully homomorphic encryption [61], however, is far from being practical these days it remains a theoretical construction. While there are working practical obfuscation techniques, even the most sophisticated techniques can be circumvented like we have recently shown [98].

One possible approach against memory disclosure besides encryption is to reduce data lifetime, e.g., to take care of early deallocation and to securely wipe data during deallocation, like we have shown in previous work [12]. Reducing data lifetime, however, requires modifying and recompiling existing applications and thus, it is impractical for already compiled applications.

In this chapter, we want to focus on software-based memory encryption techniques to mitigate memory disclosure attacks. There are different sources of errors that lead to memory-based information leakages: *software-based* or *logical isolation* flaws, such as core dumps and swap files, and *hardware-based* or *physical access* flaws, such as DMA and cold boot attacks. Moreover, *improper deletion* within one process and across different processes can lead to the unintended disclosure of memory [38, 39]. With transparent data encryption all information a process contains is always encrypted and automatically decrypted when accessed. With this approach, sensitive information that is used rarely can be safely hidden without requiring lifetime knowledge.

Our first solution presented in Sect. 5.1, targets the problem by providing mechanisms to transparently encrypt whole process address spaces and has been developed as an operating system kernel patch. Executable binaries of certain processes can be flagged to be included into the set of encrypted applications. In the following, memory pages of those processes are encrypted apart from a small current working set of pages and thus, protected against physical memory disclosure vulnerabilities as well as software-based isolation flaws which expose parts of physical memory.

Since encrypting process address spaces is operating system specific, we also implemented a memory encryption solution within the hypervisor which we describe in Sect. 5.2. Besides being operating system independent, encrypting at the hypervisor-level also has the advantage of protecting sensitive information stored within the kernel space of the guest operating system. However, performance is decreased drastically compared to our kernel-based solution.

5.1 Kernel-based Address Space Encryption for User-mode Processes

Process isolation and access control have proven to be conceptually elegant and widely deployed principles for preventing one process from accessing another process' memory [104]. In practice, however, the improper deployment of access control and side effects of memory optimizations and frequently debugging undermine the principle of isolation, leading to unexpected disclosure of otherwise isolated memory [38].

Prominent examples of such inadvertent memory disclosures rely on established operating system design principles such as swap files and crash reports (so-called *core dumps*) that intentionally write process contents to disk, and thereby disclose process memory in plain. Once a swap file or core dump file exists on disk, it is only protected by logical means against illegal access; hence, it is susceptible to improper configuration of access control [1, 2, 3] or to starting a secondary OS that is not constrained to adhere to the original access control restrictions. Similarly, kernel drivers, including those provided by third parties, are prone to error and can give attackers full access to the physical memory space. For example, Samsung's official firmware for the Exynos chipset used in Android phones of the Galaxy series disclosed memory to attackers by accidentally offering an unprotected `/dev/mem` device [4]. Once there is access to core files, the swap file or a full memory image, an attacker can typically recover known process structures, or start a pattern-based recovery of cryptographic keys. Moreover, since RAM constitutes a shared resource and, hence, portions of RAM are frequently re-used, improper de-classification or insufficient deletion may cause the inadvertent disclosure of sensitive information [39, 133].

Contribution We propose RamCrypt, a kernel-assisted encryption of the entire address space for user-mode processes. If non-interference cannot be provided and the sensitive data of a process is leaked, it is still encrypted and cannot be read from attackers without kernel privileges.

In detail, our RamCrypt solution faces two major challenges:

1. The key used for encryption must never be stored in RAM to avoid chicken-and-egg problems and disclosure.
2. Even though the contents of any user-mode address space are always encrypted, processes shall experience unmodified behaviour, i.e., unhindered program execution and transparent data access.

Our technical contributions are:

- Sensitive data within the address space of a process is encrypted on a per-page basis inside the OS page fault handler. Sensitive data within core dumps and memory that get accessed by debuggers, for example, are therefore protected.
- Only a small working set of pages, also called the *sliding window*, remains unencrypted at any given time. By default, the size of this sliding window is four pages.
- The prototype implementation of RamCrypt leads to a performance drawback of 25% for single-threaded tests of the *sysbench* benchmark suite. For multi-threaded tests, the performance drawback is generally higher and heavily depends on the chosen sliding window size. For non-protected programs, RamCrypt imposes no performance overhead.
- By using *CPU-bound encryption*, cryptographic keys are never stored in RAM, but solely inside CPU registers over the entire uptime of a system. All user-mode processes are protected with the same key, which is only visible in kernel mode.
- RamCrypt can be enabled on a per-application basis by setting a flag inside the ELF program header, without the need for binary rewriting or recompilation.

5.1.1 Linux Virtual Memory Management

On a modern CPU, process isolation is realized by simulating pristine address spaces to each process, which cannot interfere with each other. Every process starts with an empty address space that is divided into so-called *pages*. While executing code and accessing data, the necessary pages are loaded into the address space on demand by catching the first attempt to access them. Contiguous virtual pages do not necessarily have to be backed by contiguous physical pages. Each access to a page is first handled in hardware by the Memory Management Unit (MMU), which translates access to a virtual page into an actual physical page. Whenever such a translation fails, the OS page fault handler is invoked and determines whether missing pages need to be loaded or if an access to invalid memory occurred. Since every access to code and data has to go through the page fault handler at least once, this is a suitable part to incorporate RamCrypt. RamCrypt modifies the page fault handler such that not only first accesses are caught, but it additionally ensures to catch the next access to that page again by manipulating flags within the corresponding Page Table Entry (PTE).

The page fault handler distinguishes between valid reasons that can be handled and invalid access, which results in the immediate termination of the faulting process. Valid reasons include a page that was paged out to swap or is not yet loaded from a file. The handler instructs the OS to make these pages available. Invalid access, on the other hand, occurs if a page is not supposed to be present, e.g., null pointer dereference, or if a process tried to write to a read-only page.

From the hardware's point of view, a page fault might have different reasons depending on the flags that are set within the lowest bits of the PTE. The x86 architecture indicates the cause by setting flags in the `CR2` register. The

Present Flag is set, whenever a page is not swapped out and has been loaded already.

Writable Flag is set if the process is allowed to write to that particular page.

NX Flag (*no execute*) prevents code execution in that particular page.

The hardware, however, does not know about the reason of a page fault but just passes the virtual address of the page together with a status word indicating which flag caused the fault to the OS page fault handler.

Under Linux a page fault happens hundreds of times a second for various reasons. For each page fault, it is checked whether an access violation occurred, e.g., whether a write access to a non-writable page or an instruction fetch from a non-executable page occurred. The memory region that corresponds to the virtual address causing the page fault is searched and the access flags of this region are compared with the reason of the fault. If an access violation is detected for any memory region, the process is terminated by a segmentation fault. This part of the page fault handler is architecture specific because it accesses the status word set by the MMU. If there is no access violation, the generic architecture independent page fault handler is called. Within the architecture independent page fault handler, different scenarios such as new mappings, shared mappings and swapping can be handled. Most page faults are handled within this architecture independent handler, and this is also the place where we implemented RamCrypt.

The virtual address space of a process is organized in Virtual Memory Areas (VMAs), which represent contiguous memory regions with associated access rights and rules for page faults within such a region. The access rights and mappings within PTEs may differ from the associated access rights for a VMA and differences are resolved within the page fault handler for that VMA. There exists, for example, a special zero page that is always mapped if a new page is allocated and just contains zeros. The page is set read-only within the corresponding PTEs and gets replaced by a newly allocated page once the first write access occurs. A VMA can be shared between processes, backed by a file or just contain anonymous private data.

Another important concept of Linux' memory management is Copy-on-Write (COW): A new process under Linux is created with the help of the `fork()` system call, which might be followed by a call to `execve()` to execute a new binary. To make `fork()` an efficient system call, the entire

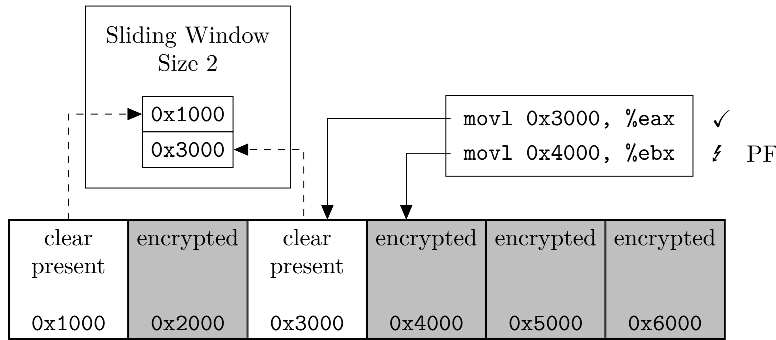

Figure 5.1: Sliding window. Next access will cause a Page Fault (PF) for page $0x4000$.

address space is not copied but only its virtual mappings point to the same physical pages. In order to avoid interference with the parent process, the cloned mappings are marked read-only, regardless of their original access rights. This ensures that read access can occur normally, while write attempts will be caught resulting in a transparent writable copy of the affected page.

5.1.2 RamCrypt Architecture

RamCrypt encrypts physical pages of a given process during its runtime and automatically only decrypts exactly those pages, which are currently accessed by the process. To this end, we leverage the existing Linux page fault mechanism, which already provides logic to transparently map not-yet-existing pages gradually into the address space. We augmented the binary *present/non-present* understanding of the kernel by another dimension: *encrypted/cleartext* data. This allows us to build upon the existing techniques that trap code, which tries to access non-present data – but with the additional distinction that this data might already exist, just in encrypted form. If so, it is automatically decrypted by the kernel such that access to that data seems to be in clear to the program.

RamCrypt can be activated for each process separately and, if activated, encrypts all pages associated with anonymous private memory. Anonymous private memory is the opposite of shared process memory in Linux. In other words, private memory is anything created by a process and includes the BSS segment, heap and other allocated memory, the stack, and private data of all loaded shared libraries. Shared memory mappings between processes and mappings that are backed by a file, e.g., code, are intentionally not encrypted by RamCrypt as these can intuitively be considered *public* anyway. Consequently, all sections of a program that can contain sensitive information are protected by RamCrypt and only code sections mapped as executable are left unmodified.

Sliding Window The memory pages that have been encrypted by RamCrypt need to be decrypted when data stored inside them is about to be accessed. Due to limitations of the x86 architecture, one can only detect memory access on a page (4 kB) granularity. Therefore, RamCrypt decrypts an entire page whenever a process tries to access data stored within a still encrypted page. As long as data residing within that page is accessed, no measures need to be taken by RamCrypt since data is temporarily available in clear inside that page. As soon as data residing in another page is accessed, RamCrypt can decrypt that other page after encrypting the last accessed page again. This way, only one page at a time is available in clear. However, this strict encryption/decryption pattern is not practical for workloads that heavily access data residing in two or more different pages. To overcome this performance bottleneck, we introduce a *sliding window* per process, i.e., per virtual address space, which represents the last n pages that were accessed and are hence kept in clear. Every other page is always only kept in encrypted form. RamCrypt always ensures that accessing an encrypted page triggers a page fault and the sliding window is calculated anew. The size of the sliding window, which is illustrated in Figure 5.1, can be configured as a kernel boot parameter and is a trade-off between performance and security.

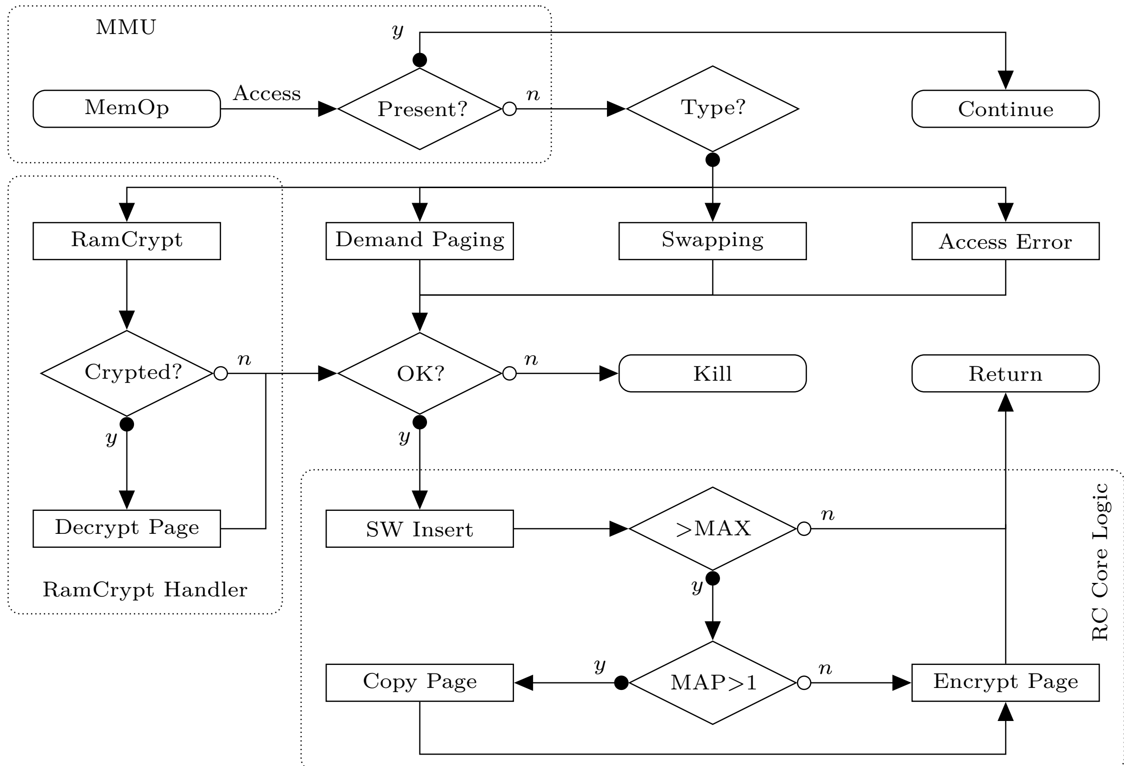

Figure 5.2: Flow diagram of how the MMU and the Linux kernel with RamCrypt interact.

RamCrypt Workflow From a high level perspective, RamCrypt consists of two parts: (1) a modified page fault handler that handles accesses to encrypted pages, and (2) the RamCrypt core logic. Basically, the RamCrypt page fault handler decrypts the page for which a page fault occurred while the RamCrypt core logic determines which page needs to be encrypted again in order not to exceed the maximum number of n cleartext pages at a time. In detail, the new page fault handler decrypts a page and then makes it accessible by changing its corresponding PTE. The RamCrypt core logic in turn checks whether the executed page fault handler has succeeded, then appends the new page to the sliding window and checks if and which page to remove from the window. Removal includes encrypting the page and clearing the *present* flag to ensure a page fault will be triggered the next time the page is accessed.

Note, that due to Linux' demand paging, it is sufficient to hook RamCrypt into the page fault handler as access to all new mappings always triggers a page fault. This also registers them in the sliding window mechanism. The overall process of how RamCrypt interacts with the remaining parts of the Linux kernel down to the hardware is shown in Figure 5.2.

Topping off RamCrypt In order to address all eventualities of a program, the whole lifecycle of its data must be considered. This foremost includes the controlled deletion of data. To this end, RamCrypt scrubs de-allocated pages with zeros. This happens either if a memory region is deliberately freed by the programmer, or when a process is terminated. This security measure ensures no clear pages are leaked once memory pages became inaccessible to the program. Additionally, it makes decryption impossible should the key ever be retrieved.

While the overall concept of RamCrypt may sound simple, a lot of challenges had to be addressed in order to handle the inner peculiarities of memory management in a modern operating system. In particular, Copy-on-Write (COW), multithreading and multiprocessing make the abstracted sliding window approach that we presented so far more complex in reality. To meet the goal of backward compatibility with unmodified binaries, a lot of corner cases had to be added to the RamCrypt logic

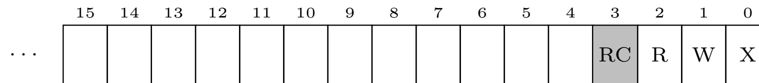

Figure 5.3: Flag field of the PT_GNU_STACK program header.

to support multiple threads within a single address space and all flavors of forking, which involves the creation of a new address space with temporarily shared COW pages. A naive implementation would destroy common COW semantics for newly `fork`'ed processes, as child and parent process share the same physical memory pages. Encryption in one process would lead to encryption in another process, of which the other process must be aware. To address this unwanted behavior, the COW semantics had to be made aware of the additional encryption dimension to copy data on encryption. Another challenge that arises is that `fork`'ed and `execve`'d processes might request RamCrypt encryption, while their parent was unencrypted. Here, COW cannot be used at all. Our implementation fully supports transitions from encrypted processes to non-encrypted processes and vice versa.

We use a variant of the TRESOR cipher (Sect. 2.4) which is configured to operate like AES-128 in the XEX mode of operation. The Initialization Vector (IV), which is fed into the encryption routine to get the tweak for XEX, is the virtual address of the page that is about to be encrypted, concatenated with the PID of the currently running process. The reason we use virtual addresses instead of physical addresses for the IVs is that physical addresses may change over time when the kernel relocates pages. As a consequence of using virtual addresses, a single page cannot be mapped to two different virtual addresses within the same address space, because otherwise the decryption of both mappings would lead to different plaintext data. As shared memory regions are out of scope for our implementation, and since temporarily shared pages due to COW semantics are always mapped to the same virtual location, using virtual addresses as initialization vectors does not limit RamCrypt.

Using PIDs as part of the IVs prevents attackers, who have access to a physical memory disclosure vulnerability, to guess the content of an encrypted page by creating a malicious process that maps pages to the same virtual addresses. After reading out an encrypted page, both ciphertexts could be compared to conclude whether the plaintext data has been guessed correctly.

If a process forks, the child process uses the PID of the parent process instead of its own PID as part of the IV such that temporarily shared pages between the parent and the child process can be decrypted. When the child process executes another RamCrypt-enabled binary by calling `execve`, the child's own PID is used as part of the IV as no temporarily shared mappings between the parent and the child process are used anymore. The combination of a virtual address and the PID as IV ensures that no patterns of encrypted data can be extracted when using RamCrypt.

5.1.3 RamCrypt Implementation

After a high-level overview of RamCrypt's architecture, we now give details about its implementation. RamCrypt is a Linux 3.19 kernel patch consisting of 2200 LoC located in 30 different files including TRESOR and its key derivation and loading mechanism. Currently, RamCrypt only supports both the IA-32 and AMD64 architecture, but thanks to its modular design, the architecture specific part of RamCrypt can easily be adopted to more architectures in the future. For ARM, for example, a CPU-bound cipher already exists [65] and thus, RamCrypt can be ported to ARM with low effort.

To be able to activate RamCrypt on a per-process basis, executable binaries need to be flagged in a way that the kernel program loader is able to read this information. The existing mechanism we used are the flags stored in the `PT_GNU_STACK` program header of an ELF executable file (Figure 5.3). This header carries access flags (readable, writeable and executable) for the program's stack. The `PT_GNU_STACK` header is interpreted by the Linux program loader which creates a new address space and is therefore suited to incorporate an additional RamCrypt bit (RC). As currently only

63		11	10	9	8	7	6	5	4	3	2	1	0
NX	··· Page Table Address ···	SW3 HID	SW2 SPL	SW1 SPC	G	S	0	A	C	W	U	R	P

63		11	10	9	8	7	6	5	4	3	2	1	0
NX	··· Physical Page Address ···	SW3 HID	SW2 RC	SW1 SPC	G	0	D	A	C	W	U	R	P

Available to the OS

Figure 5.4: Page Middle Directory (PMD) and Page Table Entry (PTE) with new RC flag.

the lowest 3 bits of the 32-bit header are in use, we reserved the 4th bit of the flag field as this bit is never set for unmodified binaries.

Our RamCrypt patch adds functionality to check for the RC bit inside the `PT_GNU_STACK` program header during the `exceve()` system call which loads a new program. The newly created task structure for the corresponding process gets assigned a RamCrypt flag, which the RamCrypt core logic then interprets when handling memory pages. Since the task structure is copied during `fork()`, child processes are encrypted as well. However, the encryption is disabled if the child process executes another binary for which the RC bit is not set. With this behavior, the stored RC bit associated with an executable file is perfectly reflected as encrypted/unmodified processes. To help setting the RC bit in executable files, we created a user-mode utility to enable or disable the status of RamCrypt for any given ELF file without the need for recompiling the binary.

Managing Memory Pages If the RC bit is set for a given process, RamCrypt has to manage and catch accesses to each memory page that is supposed to be encrypted. It achieves that goal by modifying Linux' internal representation of virtual memory pages: The Page Table Entries (PTEs). In order to cause a page fault on each new access to a page, we clear the present flag (bit 0) within the PTE and additionally set a new flag (bit 10) indicating that the page is encrypted. These changes only take effect after instructing the hardware to remove the according cache line from the TLB. As shown in Figure 5.4, this new flag reuses the second software defined flag for x86 page tables, which is only valid for Page Middle Directories (PMDs) within the Linux kernel and therefore available for PTEs. With the help of this new flag, the Linux page fault handler is able to decide whether to call the RamCrypt handler. RamCrypt also changes the Linux macro that detects whether a page is present since we abuse the original present flag. This way, we do not need to modify other parts of the kernel since the kernel always handles encrypted pages as present.

In addition to the newly introduced encryption flag in the virtual PTEs, we also introduced a flag for each physical page's management structure. The Linux kernel maintains a structure for each page and thus, including this new flag imposes no additional storage overhead. The encryption flag per physical page indicates whether this physical page is actually encrypted whereas the information indicated by the flag within a PTE might be outdated because of COW semantics, for example, if a page mapping is duplicated.

Sliding Window A separate sliding window is associated with each process. The sliding window is simply a doubly linked list containing the virtual addresses of pages stored in clear. It is initialized when a new process is created, i.e., forked, and all memory structures including the sliding window are copied to the new child process. For a newly started process, the sliding window is empty and grows according to the access pattern of the program. If the size of the window exceeds the maximum defined window size n, a page has to be selected to be deleted from the window. This replacement strategy is very similar to cache replacement strategies and the selected replacement algorithm heavily defines the overall performance. We chose a doubly linked list, or more specifically a FIFO replacement strategy as it requires very little bookkeeping to implement. More advanced algorithms such as Least Recently Used (LRU) might improve the performance of the system but require ageing information of access patterns to be available.

Multithreading Our implementation is fully compatible to multithreaded applications. However, the performance suffers from too many threads as we show in Sect. 5.1.4. The reason is that the sliding window is created *per process* rather than *per thread*, which results in threads competing for pages inside the sliding window. All changes to the sliding window must go through a semaphore to ensure atomic access. Of course, the synchronization mechanisms are necessary to make the implementation correct, however, they lead to a performance overhead. Adjusting the size of the sliding window n dynamically at runtime by scaling it by the number of concurrent threads is not an option because then the limited number of plaintext pages could not be guaranteed anymore.

Address Space Creation The RamCrypt implementation also supports forking. Please recall that PTEs and the sliding window are copied when forking a newly created process. Consider a page fault occurs within the child process for an encrypted page that is shared between the parent and the child process. The RamCrypt page fault handler then decrypts the page as usual but only changes the PTE for the child process and not the parent process to avoid deadlocks. When another page fault occurs for the parent process at some point in time, the RamCrypt page fault handler is not allowed to decrypt that page again.

The RamCrypt core logic must also be aware of forking. Forking results in physical pages being mapped into more than one virtual address space. As a result, RamCrypt cannot encrypt such a shared page because another process would then work on encrypted data. Instead, we use a technique similar to COW: RamCrypt duplicates the page in question, adjusts the appropriate attributes and lets the process continue as if nothing happened. For that to work, RamCrypt has to allocate a new physical page, store the encrypted content of the old page inside the newly allocated page and finally modify the PTE within the current process to let it point to the new page instead. The old page is now still in clear but is managed by the sliding window of another process and will eventually be encrypted when it is no longer used. Note that these precautions have to be only taken for processes but not for threads. If a second thread was currently reading or writing the page, a page fault for this thread is triggered immediately and the page is decrypted again. Race conditions are excluded by proper locking mechanisms between the different threads.

Cipher The RamCrypt implementation uses the single block cipher API within the Linux kernel to access the TRESOR block cipher. As a consequence, TRESOR can be used for disk encryption as well as for RAM encryption in parallel. Furthermore, it is possible to exchange the TRESOR cipher with any other cipher that is supported by the Linux crypto API. Other CPU-bound ciphers within the Linux kernel such as ARMORED [65] can be included in the future.

5.1.4 Runtime Performance

All evaluations have been performed on a standard desktop computer with an Intel Quadcore CPU (Intel Core i5-2400) running at 3.1 GHz and eight gigabytes of RAM. From the software side, we used an unmodified installation of Debian Wheezy with a base system installed and a RamCrypt patched kernel.

To evaluate the runtime performance of RamCrypt, we used the *sysbench* benchmarking utility that is shipped with most Linux distributions. The performance overhead is calculated as the relative runtime difference of the RamCrypt-enabled version of the benchmark compared to the unmodified version that ships with Debian. The unmodified version serves as ground truth to which the flagged executable is compared. We flagged the executable file by means of our script that adds the RC flag as described in Sect. 5.1.3.

To compare the different RamCrypt settings, such as sliding window size, number of concurrent threads, and choice of cipher, we repeated all experiments with different parameters. The way sysbech works is that it tries to execute as many pre-defined requests in a given time frame. One sysbench request consists of calculating all prime numbers up to 10.000. The measured performance is proportional to the achieved number of requests for that fixed time frame. If more concurrent

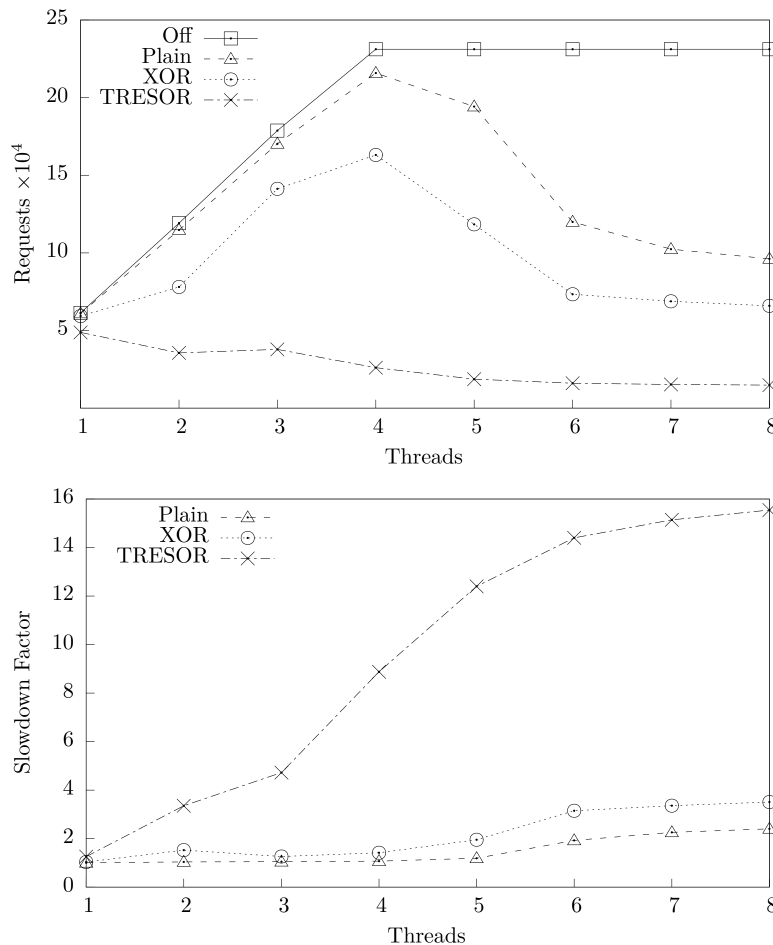

Figure 5.5: RamCrypt performance with sliding window size 4 and different encryption methods.

threads are used, the achieved number of requests per time frame increases. We chose a time frame of 60 seconds and one thread on the unmodified Debian system to establish ground truth. All the RamCrypt parameter changes result in slightly lower number of requests achieved in those 60 seconds. The slow-down factor we want to measure is the quotient of the established number of ground truth requests divided by the achieved number of requests with RamCrypt enabled.

Cipher Performance Impact In Figure 5.5, different encryption methods for RamCrypt are evaluated with the default sliding window size of four pages to show how big the overhead of the TRESOR cipher alone is. To this end, we ran RamCrypt with the TRESOR cipher enabled, with the identity function that outputs its input and with XOR encryption. The encryption using the identity function basically measures the overhead of the paging and RamCrypt logic but has almost no overhead due to the lack of CPU-bound encryption. Hence the name in the graph is *plain* encryption.

In the upper diagram of Figure 5.5, the absolute number of requests for the 60 second time window is shown while in the lower diagram, the slowdown factor compared to the ground truth is depicted. All measurements with RamCrypt enabled have in common that the sliding window size of four becomes a bottleneck for more than four threads as their concurrent execution competes for the next available window. The RamCrypt implementation that uses CPU-bound TRESOR encryption shows a slowdown of approximately 25% for the singlethreaded test. Most of this slowdown can be attributed the CPU-bound cipher as the plain encryption only shows a slowdown of 0.5% for the same singlethreaded configuration. As expected, when the number of concurrent threads increases,

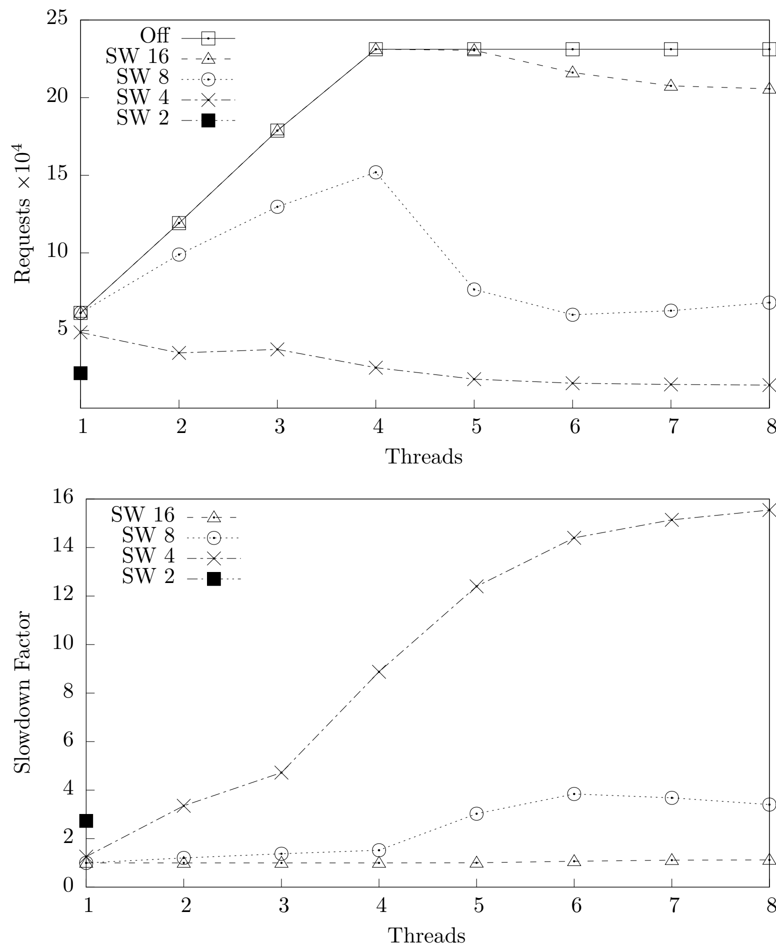

Figure 5.6: RamCrypt performance with TRESOR cipher and different SW sizes.

the slowdown is larger due to the fixed-size sliding window. Note that RamCrypt is always enabled for the whole process, meaning for the main executable and all loaded libraries, and thus includes data of libraries, as well. The overhead would be significantly lower if only the main executable was protected. So this number constitutes a worst case scenario.

Sliding Window Performance Impact In Figure 5.6, RamCrypt is evaluated with different sliding window sizes. For a sliding window size of 16, i.e., sixteen clear pages are available for up to eight threads, our implementation scales almost with the distribution version of sysbench. With eight threads, a slowdown of approximately 12% is reached. However, in a multithreaded process, the threads compete for the same amount of decrypted and readily accessible pages at the same time. For a sliding window size of two, sysbench is not able to launch more than one thread because it requires an additional control thread as well. In the singlethreaded run with a sliding window size of two, the slowdown factor is as high as 170%, while size four of the sliding window pushes the slowdown to acceptable 25%. With sizes of eight and sixteen, there was no slowdown measurable.

Computation- vs. Data-Intense Programs In addition to the sysbench benchmarking suite, we also performed evaluations for an exemplary set of data-intense programs. In Table 5.1, the average number of page faults and the total execution time for a selection of data-intense command line programs is shown. While RamCrypt has a negligible impact on the runtime behavior of a process and its number of page faults for a sliding window size of sixteen, a notable performance drawback is measured for smaller sliding window sizes like eight and four. This is to be expected and can be

Command	Off		SW Size 16		SW Size 8		SW Size 4	
	#PFs	Time (s)	#PFs	Time (s)	#PFs	Time (s)	#PFs	Time (s)
`ls linux-3.19.7`	118	0.00	148	0.00	759	0.05	4,032	0.29
`md5sum linux-3.19.7.tar.xz`	84	0.16	89	0.16	52,463	4.07	55,485	4.26
`sha512sum linux-3.19.7.tar.xz`	87	0.32	91	0.32	52,465	4.22	55,722	4.47
`tar -xf linux-3.19.7.tar`	118	1.94	183	2.01	1,768,810	184.41	5,685,424	500.15
`tar -xJf linux-3.19.7.tar.xz`	16,660	6.45	16,702	6.47	1,785,438	151.92	4,459,535	377.10

Table 5.1: Avarage number of page faults and total time for data-intense programs.

explained since data-intense programs work on different data pages in a row causing more page faults than CPU-intense programs. Hence, more data accesses have to be handled by the RamCrypt page fault handler. Note that in usual scenarios, data-intense programs like `tar` would not be protected by RamCrypt when no sensitive information is expected to be disclosed.

Summary Concluding the performance evaluation, it can be said that in a singlethreaded environment the slowdown of only 25% for a sliding window size of four is acceptable. For highly multithreaded environments, however, the sliding window needs to be adjusted accordingly. This, however, might lead to a higher probability of disclosing secrets as more cleartext pages remain in RAM at any given time.

5.1.5 Binary Compatibility and Practical Security Analysis

Compatibility is an important property of new concepts to become widely accepted. With RamCrypt, we have designed and implemented a solution that is fully compatible to existing systems, meaning that all legacy programs are able to run with RamCrypt enabled by the flick of a button. If a program is known to carry sensitive data, the corresponding binary just needs to be flagged with the help of our provided command line tool and every following execution automatically and transparently makes use of RamCrypt. In particular, no recompiling, no designated programming language features and no user interaction is needed – apart from flagging executables once. Also scripts can be protected by flagging the interpreter (or a copy of the interpreter) that is used to interpret the script. For our tests, for example, we have also flagged the system shell (`/bin/dash`) and executed other binaries (flagged and non-flagged) within this shell.

Effectiveness Within our security analysis we made sure that no sensitive data of RamCrypt enabled processes is visible in RAM. For our test setup, we wrote a program that maps a large area of memory (256 MB) and successively writes different, but deterministic 32 byte patterns throughout the entire memory area. After the data has been written, the program stops execution and waits for the user to press a key, i.e., it blocks. We then dumped the entire 8 GB of memory while the process was still running to search it for these known, deterministic patterns. In particular, the first 16 bytes of each pattern are filled with a unique 128-bit value we randomly generated, while the second 16 bytes are filled with an integer that represents the index of the current pattern within the large memory area. The dump was created using the Linux Memory Extractor (LiME) and is essentially equivalent to a cold boot attack without bit errors. We then searched for the unique previously generated 128-bit value which is included in every pattern within this large dump and observed the following: If RamCrypt is disabled for our test application, we find thousands of matches of the pattern. If RamCrypt is enabled, we cannot find any match of the pattern. We did not perform a real cold boot attack or tried to acquire the physical memory in other ways than with LiME. Note, however, that dumping the physical memory with LiME is even a stronger security guarantee than performing a real cold boot attack because the whole physical memory is perfectly obtained without any bit errors. The reason why we did not find a single match for our unique 128-bit value within the physical dump despite our sliding window, is that calling the library function, which waits for the user to press a key, does replace the pages in the sliding window.

		Temporal Exposure per Page (%)		
		n=4	n=8	n=16
Secret Key Pages		3.07	14.37	21.68
All Pages	Min	0.0000	0.0005	0.0017
	Avg	7.63	12.66	17.95
	Max	99.83	99.76	99.99
	StdDev	19.77	21.82	25.43

Table 5.2: Temporal exposure of unencrypted pages (in %) for different sliding window sizes n. Measured *nginx* process served 1500 SSL requests.

As an additional test, we triggered a *core dump* to inspect it for leaked data. To this end, we terminated our test application with the *SIGQUIT* signal by pressing "CTRL-\" while the application waits for the user to press a key. The *core dump* file of the application was then inspected and revealed the following result: The core dump of a non-flagged binary contained several matches of the deterministic pattern while the core dump of a flagged binary did not contain any match, for the same reason as for the physical dump. We therefore conclude that core dumps are effectively protected by RamCrypt as well.

Efficiency over Time Due to the nature of RamCrypt, pages that are currently listed within the sliding window have to reside unencrypted in the RAM and thus might leak sensitive data over a limited time span. We therefore measured the relative time of unencrypted pages residing in memory for a real world application and specifically evaluated how long secrets are exposed in clear. We flagged the *nginx* web server to use RamCrypt and measured how long each page is accessible in clear, i.e., how long it was listed within the sliding window. To make nginx contain actual secrets, we set it up such that it delivers SSL-encrypted HTML pages under maximum load. The HTML document we transferred was a standard welcome site of nginx with a size of 151 bytes. Furthermore, we identified the memory pages in which the private exponent of the RSA key resides. We then observed how long these pages have been exposed to RAM in clear.

Table 5.2 shows the minimum, average and maximum relative time as part of the overall runtime of a page being exposed in clear for different sliding window sizes. Since nginx was used under maximum load, these numbers represent worst case exposure times for the key material, which under less load is consequently exposed shorter. From the minimum, maximum and standard deviation, we can conclude that some pages are used over the entire lifetime of a process while others are used only rarely, for example, during startup. The first row of Table 5.2 shows how long the pages containing secret key material are exposed in clear over a process' lifetime. With increasing sliding window size, the average time of pages accessible in clear increases because more pages are stored in clear in parallel. For our default sliding window size of four, however, we consider a relative time frame of only 3% (where the secret is being exposed in clear) a good result. Although RamCrypt is not able to entirely prevent accessing the secret with the help of memory disclosure attacks, the bar for attackers is raised since precise timing becomes necessary. Furthermore, attacks like cold boot might entirely be prevented if the remaining 3% of cleartext data only resides in CPU caches but never enters RAM.

5.1.6 Discussion

Although RamCrypt is able to encrypt the data of whole process address spaces based on the user's choice, process-related data outside the virtual address space of a given process might still be accessible in clear. In particular, kernel or driver buffers and the buffers of peripheral components might contain sensitive data which cannot be protected by RamCrypt. Recently, it has been shown that the frame buffers of the VRAM can be recovered even after a reboot [17]. One approach to at least partially mitigate the issue of sensitive data stored in buffers outside the process' virtual

memory is the use of ephemeral channels [49]. The downside, however, is that this approach requires modified peripheral devices and explicit use in source code.

Attacks such as the popular OpenSSL Heartbleed vulnerability (CVE-2014-0160) or other memory disclosure attacks [83] that are able to read out RAM with privileges of the attacked process cannot be prevented as RamCrypt transparently decrypts RAM when accessed by the owning process. There is no direct fix to this limitation as RamCrypt needs to act transparently to any process that is RamCrypt-enabled. To address this issue, source code needs to be changed, e.g., to protect itself against inadvertent access when the data is not needed by using CPU-bound encryption schemes [77, 76, 60] that do not store keys in the address space of a process.

Although RamCrypt with a sliding window size of four imposes a performance overhead of only 25%, performance must still be improved for multithreaded applications. As shown in Sect. 5.1.4, the performance is strongly influenced by the cipher that is used. Consequently, speeding up the TRESOR cipher would also give a performance boost to RamCrypt. To speed up TRESOR, hardware transactional memory could be exploited as it has been done for a CPU-bound RSA implementation [77]. Another possibility is to compute the key schedule for a certain number of blocks at once (similar to ARMORED [65]) to avoid re-computations for every block.

5.2 Hypervisor-based Encryption of Kernel and User Space

When a computer is turned off, data in main memory is not lost immediately, but instead gradually fades away over time. This fact can be exploited by certain attacks such as cold boot attacks [81], attacks using the Firewire interface [23] and other DMA attacks that are capable of extracting RAM contents [135]. To effectively protect RAM, one of the most sensitive parts of modern computers, including private RSA keys, disk encryption keys, online banking credentials and user logins, we decided to encrypt RAM on hypervisor-level to be able to protect kernel space and hence, to ensure maximum compatibility and security.

On the one hand, encrypting RAM on the hypervisor-level compared to the kernel-level, has the downside that only memory disclosure vulnerabilities exploited by physical attackers can be prevented. The exploitation of vulnerabilities in kernel drivers which expose parts of the physical address space, for example, cannot be prevented, because the correspondent data would transparently be decrypted for the attacker by the hypervisor. On the other hand, encrypting on the hypervisor-level includes encrypting kernel space and thus, protects sensitive information stored in kernel buffers from being exposed to physical attackers.

Contribution We present HyperCrypt, a hypervisor-based encryption of the guest's main memory to protect against physical attacks such as cold boot and DMA attacks, while being transparent for applications and the OS kernel running on top of it. In detail, our contributions are:

- At any time, the vast majority of main memory is kept encrypted. Only a small, configurable working set of pages remains unencrypted. These pages are kept within caches with a high probability.
- We demonstrate the practicability of our approach with a prototype implementation running a standard Linux system with an nginx webserver.
- The prototype of HyperCrypt leads to an overhead of 37% for the nginx webserver under heavy load.
- Utilizing CPU-bound encryption, the encryption keys used by HyperCrypt are never exposed to RAM.
- HyperCrypt, as a hypervisor-based solution, is fully transparent to all applications as well as the operating system kernel.

HyperCrypt has been developed as an open source patch building on top of the BitVisor hypervisor [142].

5.2.1 BitVisor Memory Management

BitVisor [142] is a thin hypervisor based on Intel VT-x and AMD-V for enforcing I/O device security. It is a so called *parapass-through* hypervisor meaning that only a small set of hardware accesses, for which security should be enforced, are intercepted by the hypervisor while other accesses are passed through to the hardware. The advantage of this concept is a dramatically reduced code size in comparison to traditional VMMs such as XEN or KVM. BitVisor with enabled ATA parapass-through driver, for example, needs only 21,400 lines of code and is capable of performing transparent full disk encryption. Naturally, thin hypervisors have limited functionality compared to real VMMs; most notably, BitVisor only allows running a single VM on top of it, eliminating the components for sharing and protecting system resources amongst different VMs.

To manage memory within a hypervisor, there are basically two widely known concepts for x86. The first concept, known as Shadow Page Tables (SPT), requires a hypervisor to deny the guest operating system write access to its own page tables. Instead, write accesses are trapped and handled in the hypervisor after applying certain mapping rules. The advantage of this approach is that, apart from basic virtualization support, no other hardware support is needed. The disadvantage,

however, is that handling all page table changes within the hypervisor has a serious impact on performance.

We therefore decided to implement HyperCrypt based on the modern approach of implementing memory management within a hypervisor called Second Layer Address Translation (SLAT). SLAT requires support from the processor such as, for example, the Extended Page Tables (EPT) feature provided for Intel CPUs. With EPT, virtual addresses are first translated to guest physical addresses with the help of regular page tables managed by the guest. These guest physical addresses themselves are then translated to host physical addresses with the help of the EPT managed by the hypervisor. Consequently, the guest has a faked view of physical memory which can be provided by the hypervisor by manipulating the EPT. For our implementation of HyperCrypt, we mislead the guest operating system regarding the number of pages that are currently accessible in real physical memory by targeted EPT manipulations.

5.2.2 HyperCrypt Architecture

HyperCrypt encrypts host physical pages and automatically decrypts those pages which are currently accessed by the guest OS or an application running on top of the guest OS. To this end, the EPT fault handler of BitVisor is extended to support the dynamic encryption and decryption of physical pages. As BitVisor is a thin hypervisor with only a single guest OS running, the EPT mapping within BitVisor is simple. Almost every guest physical address is mapped one-to-one to the corresponding host physical address with the exception of the physical pages containing the hypervisor itself. These pages are hidden from the guest operating system by hooking the BIOS call for getting the system memory map and re-mapping the addresses within the EPT is not allowed.

Applications running on top of the guest OS, as well as the guest OS itself, can access pages only if a corresponding EPT entry exists. If no such entry is found, a trap into the hypervisor occurs and the fault is handled by the EPT fault handler. HyperCrypt leverages the one-to-one mapping mechanism by dynamically adding and removing entries from the EPT when physical pages are encrypted and decrypted, respectively.

Sliding Window Ideally, only the single page that is currently in use by the guest OS is left in plain while all remaining pages are encrypted. Hence, the EPT only contains a single entry and access attempts of other pages cause a fault followed by a trap into the hypervisor. The hypervisor could then encrypt the current page, remove the current EPT entry, decrypt the correspondent requested page, and add an EPT entry for the new page. Although this approach guarantees maximal security, because of minimal expose time of a given page, we decided against it for two reasons. First, one single page is not sufficient for the guest OS to run instruction and data fetches from two different pages in parallel, and second, trapping into the hypervisor each time a different page needs to be accessed decreases the performance dramatically such that a protected system needs days to boot up.

Instead of encrypting all but one page, we introduce a *sliding window* which keeps references to all pages that are currently kept in clear. Every page not referenced from within the sliding window is always kept encrypted. Consequently, instead of keeping only the currently used page in clear, the last n pages that have been accessed are kept in clear. The size n of the sliding window is a system wide constant and can be defined when building HyperCrypt; the default value is 1024. Note, that 1024 pages correspond to 4 MB in size which is less than most cache sizes of modern CPUs and therefore even the pages that are currently kept in clear are likely not exposed in clear within RAM.

HyperCrypt Workflow As depicted in Figure 5.7, the basic HyperCrypt workflow consists of two main parts. First, the extended EPT fault handler which on demand decrypts pages for which a fault has occurred, and second, the sliding window mechanism which after each page fault checks whether there are more than n cleartext pages and potentially re-encrypts pages referenced by the sliding window if the limit of n pages is exceeded. In detail, the extended EPT fault handler

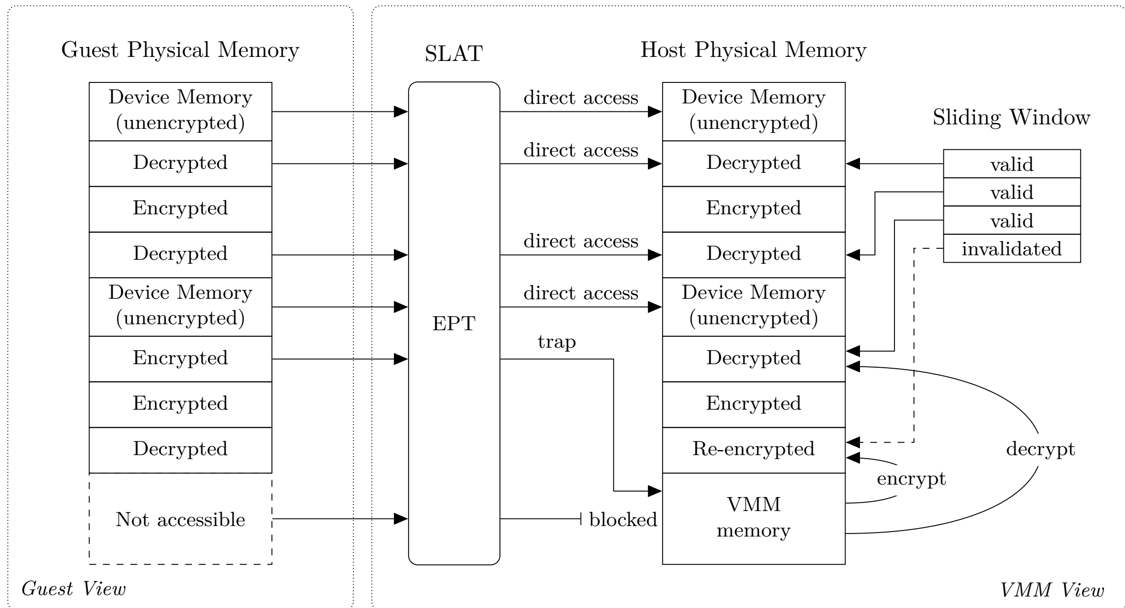

Figure 5.7: Overview of HyperCrypt's workflow.

decrypts a page and then makes it accessible to the guest by adding an entry for this physical page to the EPT. The sliding window mechanism first adds the newly decrypted page to the sliding window if the page fault was caused by an encrypted page. It then checks whether the limit is exceeded and which page to remove from the window. The chosen page is removed from the EPT table, taken out of the sliding window and finally encrypted.

Hooking the EPT fault handler within BitVisor is sufficient for implementing HyperCrypt, because the guest system is started with an empty EPT table in the hypervisor. Thus, for each first access of a page, the EPT fault handler is called and the sliding window mechanism inserts the page into the sliding window. To decide whether a page needs to be decrypted within the EPT fault handler or whether the fault just occurs because of being first accessed, a bit marks the page as currently being encrypted or in cleartext for each physical page.

HyperCrypt Challenges While the concept of HyperCrypt may sound simple, a lot of challenges had to be solved to transparently encrypt main memory out of a thin hypervisor. Since BitVisor is a parapass-through hypervisor, it does not know about the guest's memory layout and how the guest OS uses certain memory regions. We cannot simply encrypt the whole physical address space, but instead device memory and DMA buffers need to be excluded, for example. To exclude device memory, we check each memory access that causes a fault against the system memory map provided by the BIOS. If a region is accessed which does not belong to usual system RAM, the access is mapped through with a one-to-one entry in the EPT and the page is not considered for encryption. Handling accesses to DMA buffers is more complicated, because the hypervisor does not know which areas have been allocated by the guest OS to be passed to devices as DMA regions. Encrypting DMA buffers causes problems, because the devices would read encrypted data. During writing, the devices would write cleartext data which the hypervisor then potentially decrypts for the operating system leading to unpredictable results. Our approach for mitigating this problem involves using device drivers provided by BitVisor and will be further explained in Sect. 5.2.3.

For the encryption itself, we use a variant of the TRESOR cipher (Sect. 2.4) which behaves like AES-128. We further chose the XEX mode of operation with the initialization vector which is needed to generate the tweak set to the physical address of the page that is about to be encrypted. The physical page address is a reasonable choice because it is unique and cannot be forged by the guest operating system as host physical addresses are only used within the hypervisor.

5.2.3 HyperCrypt Implementation

The EPT fault handler is the point of choice to start implementing HyperCrypt. As the guest can only access pages that have an entry within the EPT and every other access results in a page fault, the EPT can be manipulated to only hold the currently decrypted memory pages while encrypted pages are never inserted into the EPT. When the guest wants to access a memory page that is currently encrypted, the MMU cannot find an entry for this address in the EPT and issues a page fault which is handled by the hypervisor. The hypervisor then decrypts the page before adding it to the EPT and returns control to the guest which will repeat the previous memory access. This memory access succeeds because the appropriate mappings exist.

Page Management In BitVisor, the EPT is empty at boot time and is subsequently filled page by page using the page fault handler. HyperCrypt stores the current status (encrypted or clear) of each physical page frame using a single bit within a bitmap. When a page fault occurs, the bitmap is used to check whether this fault was caused because the page was accessed for the first time or because it is currently encrypted and therefore not referenced by the EPT. In the latter case, the page is decrypted using the TRESOR cipher and added to the sliding window. If the page is accessed for the first time, it is added to the sliding window as well to ensure that it will be encrypted at some point in time. The sliding window contains structures of decrypted pages which store the physical addresses of these pages. In addition, these structures contain information whether or not the decrypted page is actually mapped within the EPT. This information is needed because the hypervisor itself is allowed to access encrypted pages, too, which get first decrypted. These pages, however, are not part of the EPT as the hypervisor does not use the EPT but rather accesses the pages directly via its own page tables.

To ensure that only a certain previously defined maximum number of pages are decrypted at the same time, the number of decrypted pages, that is the number of pages referenced by the sliding window, is compared to this limit each time after a page has been decrypted. If the limit has been exceeded, the first page within the sliding window is encrypted again and removed from the EPT if necessary. As in standard BitVisor no entry is ever removed from the EPT, the functionality for removing pages from the EPT had to be implemented first.

The current strategy for choosing the first page for re-encryption is basically First-In-First-Out (FIFO). However, for more recent CPUs, we also implemented a better strategy. Recent Intel CPUs now support *accessed* and *dirty* bits within EPTs which allow to use a second chance algorithm. With second chance, before encrypting a page, the pages within the sliding window are checked for the accessed bit. If the bit is set, the page was accessed recently and therefore gets a second chance. The accessed bit is then cleared and the page is added to the back of the list again. This process continues until the first page without an access bit set within the sliding window is found. This page is then encrypted instead of just using the very first page within the sliding window. The second chance algorithm performs better if there are pages that are accessed frequently, because these pages get almost never encrypted. On the other hand, if a lot of different pages are accessed constantly, the second chance algorithm introduces additional overhead by iterating over the sliding window.

BitVisor has the capabilities to map and access pages within the guest memory, and thus, it must be dealt with that encrypted pages can be accessed by the hypervisor. Consequently, the mapping functions used by BitVisor are modified in a way such that mapped pages are checked for encryption before adding them to the hypervisor page table. This is particularly important for the BitVisor drivers as they usually map guest pages to copy content from the guest to pass it to devices.

Device Memory When dealing with RAM encryption, an important part is to consider which different kinds of memory regions there are. Besides user data, program code, and kernel level data, parts of the memory are also used to communicate with devices like hard disks and keyboards. Some devices are, for example, controlled using memory mapped I/O, where the operating system writes to specified memory areas which are then read by a device and interpreted as a command. If

the entire memory used by the guest OS should be encrypted, these memory mapped regions have to be excluded. Otherwise, the encryption routine would write to these memory areas and send randomized commands to the devices. To check which regions to exclude, we use the memory map provided by the BIOS. This map is obtained by BitVisor at an early stage to check for available memory regions.

Before checking for encryption within the EPT fault handler, it is additionally checked if the address that causes a fault is part of a memory region marked as *available* in the memory map. Available means that the respective page can actually be used by the system and is not already reserved for any device. If the faulted address points to memory within a reserved region, the extended EPT fault handler will not be called and the page will be added to the EPT directly. For future accesses, this device page is excluded from the encryption process altogether and remains permanently mapped within the EPT.

DMA Buffers The most critical part of memory that has to be considered when encrypting data stored in RAM are DMA buffers. BitVisor allows the guest to directly communicate with devices and therefore the guest directly initiates DMA transfers. The guest just passes an address of a DMA buffer to a device and then proceeds with other operations. By executing other instructions, other pages are mapped and decrypted by the hypervisor which causes pages from the DMA buffer to be encrypted, potentially before the DMA transfer is completed. A device would then, for example, write unencrypted data to encrypted memory which results in unpredictable data once the page gets decrypted.

To deal with this issue, the BitVisor drivers have been used to mitigate this problem. BitVisor provides drivers to intercept commands issued to devices to transparently implement full disk encryption and tunnel Ethernet packets through a virtual private network, for example. These drivers perform the actual DMA transfer, meaning that if data should be provided for a device, the driver uses BitVisor's internal mapping functions to map the guest memory, copy the data from the guest and finally send the data to the device. With HyperCrypt, we hook all available mapping functions and ensure that data gets decrypted properly before being sent to the device. Receiving data from a device works similar, but instead the data is encrypted by HyperCrypt before being written to guest memory.

Booting HyperCrypt For booting HyperCrypt, we basically provide two options: A user password to derive the encryption key, or automatically generating a random key on each boot. The *userpassword* option can be used to define if the user wants to enter a particular key for encryption. In that case, the user is asked to enter a password at boot time, which is used to generate the TRESOR encryption key stored in the debug registers. Otherwise, the key used for the encryption is generated randomly. The random number generator of the Trusted Platform Module (TPM) is used to generate 256 bits of key material consisting of the 128-bit AES key and the 128-bit XEX tweak key. The default setting is to use a random key as the key only has to be remembered until the system is rebooted. If the user, however, does not trust the random generator of the TPM, the password method can still be used.

When configuring HyperCrypt for the first time, the size of the sliding window, which defines how many pages are unencrypted at the same time, has to be fixed. This is a security vs. performance parameter because a higher number causes more of the RAM to be exposed while the decryption routine has to be called less frequently. As one page is 4096 bytes is size, setting the value to 512 results in 2 MB of RAM being exposed, 1024 results in 4 MB and so forth. Although, 4 MB of exposed data could contain a lot of sensitive data, it is very unlikely that this data can be acquired by physical attacks on main memory due to large CPU caches nowadays. The size of the sliding window should be always chosen to expose less data than the size of the cache, and hence we set the default value to 1024 pages as a conservative choice and a reasonable trade-off between security and performance.

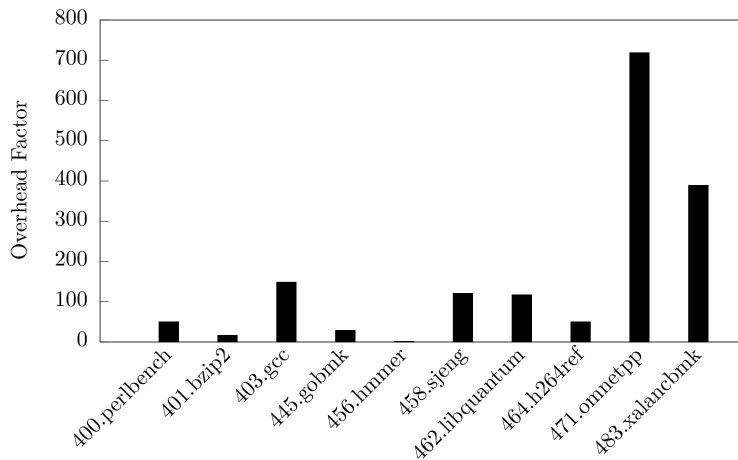

Figure 5.8: Performance overhead of HyperCrypt compared to a native Linux system using the SPECINT2006 benchmark suite.

5.2.4 Runtime Performance

In this Section, HyperCrypt is evaluated regarding performance. First, we run the SPECINT2006 benchmark on top of HyperCrypt using a standard Linux distribution and secondly, we measure the average reply rate of an nginx webserver using HyperCrypt. All evaluations have been performed on a standard desktop PC with Intel Core i7-2600 CPU and 8 GB of RAM running a Debian GNU/Linux 7.8 operating system.

SPECINT2006 Benchmarks Naturally, a huge performance overhead is created by encrypting memory pages, and additionally HyperCrypt suffers from a significant performance drawback due to context switching between the hypervisor and the guest OS when a page is encrypted or decrypted, respectively. To have verifiable performance results, we decided to use the standardized SPECINT2006 benchmark suite.

These tests were performed with our default sliding window size of 1024. Additionally the same tests were executed with a plain Linux system and standard BitVisor running Linux. However, the overhead of Linux on top of standard BitVisor over Linux without hypervisor is neglectable and therefore in Figure 5.8, we show the overhead of HyperCrypt over a standard Linux system without hypervisor running SPECINT2006.

The overhead varies a lot depending on the chosen test which most likely depends on the data locality involved in the different tests. Altogether the overhead factor of HyperCrypt varies between 15 and 148 with three outliers with an overhead factor of 1.15, 388, and 718 respectively. The bzip2 test which is more memory intense than I/O intense has an overhead of 15.8 which is the second least of all tests. The second chance page replacement algorithm gave a 19% performance boost over the standard FIFO algorithm.

HTTP Server Performance While the SPECINT2006 suite is a standardized benchmark suite that makes a comparison between other setups easy, we also want to show how HyperCrypt performs in a real world scenario. One example, where HyperCrypt could be used is a secure web server that should be protected against physical attacks on main memory. We benchmarked the widely used nginx web server [123] on top of HyperCrypt using a standard Linux. To measure the performance, we used the program Httperf [32] on a computer directly connected to our test machine. Httperf sends various requests to a given host address and evaluates the response time that the web server needs to send back the response. Httperf was configured to send 10,000 requests with a timeout of five seconds.

Sliding Window Size	No Encryption	384	512	768	1024
Test duration (s)	81.0	1,641.3	1,120.8	513.5	110.8
Avg. reply rate (replies/s)	123.4	6.1	8.9	19.5	90.2
Avg. connection time (ms)	8.1	164.1	112.1	51.3	11.1

Table 5.3: Average reply rates and connection times for an nginx web server running on top of HyperCrypt with different sliding window sizes.

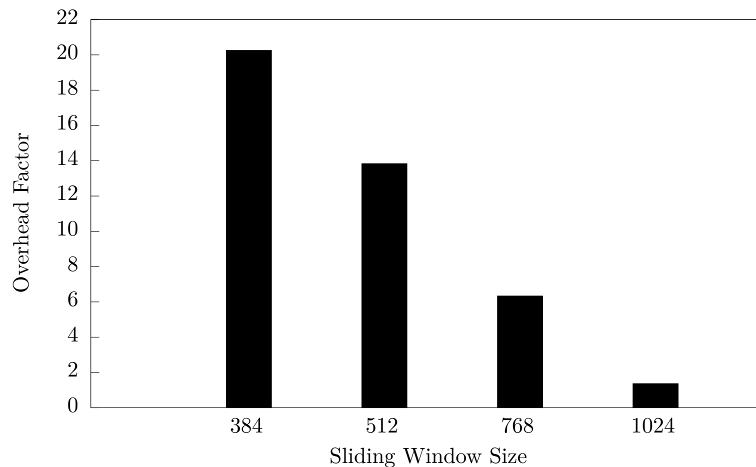

Figure 5.9: Overall performance overhead of the nginx web server running on top of HyperCrypt with different sliding window sizes.

Table 5.3 shows the absolute average reply rates and connection times for HyperCrypt configured with different sliding window sizes. As expected, reply rates are lager if more pages are allowed to reside within RAM in clear. For our default sliding window size of 1024, the reply rate is not much lesser than for a plain Linux system. The overall overhead for different sliding window sizes is visualized in Figure 5.9.

The overhead for our default sliding window size of 1024 is just 37% and has been calculated based on the runtime. For even more secure settings with less decrypted pages, the factor grows to 6.34 for 768 pages and 13.83 for 512 pages. 384 pages caused the web server to reply to the requests 20.26 times slower. As modern caches are usually much larger than 4 MB, the default sliding window size of 1024 is a good trade-off between security and performance.

With an overhead of only 37% for the default sliding window size of 1024, the evaluation shows that a real world example, like a web server, performs a lot better than the integer benchmark suite SPECINT2006. A web server is probably far more I/O intense than most tests within SPECINT2006. As our primary goal was to provide physical security for severs, this result shows a practical scenario for using HyperCrypt.

5.2.5 Practical Security Analysis

Besides performance, the most important aspect of HyperCrypt is of course whether it guarantees the security we claim. In this section, we show the effectiveness of HyperCrypt and how the sliding window size affects data exposure in general.

Effectiveness To verify that HyperCrypt mitigates data exposure, we filled a certain memory region with a repeated randomly generated pattern and afterwards searched for this pattern within physical memory. If HyperCrypt works as intended, the pattern should at maximum be only found

Sliding Window Size	No Encryption	256	512	768	1024
Minimum	249,023,439	4,465	6,505	87,595	179,140
Maximum	249,023,439	35,575	88,615	142,675	207,190
Average	249,023,439	14,767	62,605	132,959.5	200,713

Table 5.4: Number of pattern matches within physical address space for different SW sizes.

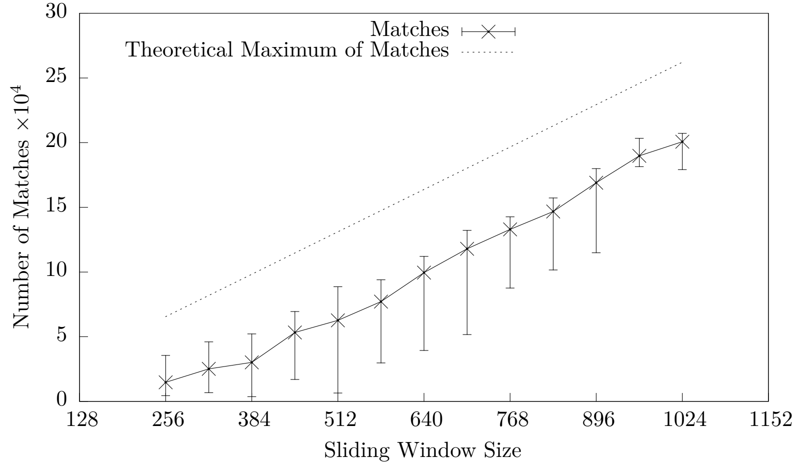

Figure 5.10: Minimum, maximum, and average number of matches for different SW sizes.

as often as it fits within all pages referenced by the sliding window. All other pages should be either encrypted or should not have been accessed at all, thus not containing the pattern.

We wrote a small program which runs on top of standard Linux and uses `/dev/urandom` to generate a random 128-bit pattern each time it is started. The program then allocates four gigabytes of memory and fills the allocated memory area with the just generated pattern. Afterwards, the program issues a `vmcall` and passes the pattern as a parameter to the hypervisor. The hypervisor searches for the 128-bit pattern within the whole physical address space and counts the matches. Performing the search from the hypervisor guarantees atomicity, meaning during the search, the guest memory contents do not change.

Despite atomicity, however, there is some fluctuation in the number of matches for a given sliding window size over multiple runs. This is due to the fact that not only the test program but also other programs are executing and that the guest OS is responsible for scheduling. We therefore ran the test program ten times for each sliding window size starting with 256 pages (1 MB) to 1024 pages (4 MB) with a step size of 64 pages.

The minimum, maximum, and average number of pattern matches for different sliding window sizes are shown in Table 5.4. Without encryption enabled, the number of matches always stays the same for each run, because the allocated area just resides in RAM in clear. With encryption, the difference between the maximum and the minimum is quite large, which depends on, for example, whether another process was scheduled in the meantime. The maximum number of matches, however, never exceeds the potential possible number given by the size of the sliding window which confirms that the security claims are really guaranteed.

Figure 5.10 visualizes the minimum, maximum, and average number of matches with error lines. Furthermore, the potential possible number of matches determined by the sliding window size is shown as separate graph. It is clearly visible that the maximum is always below the potential maximum of matches. Note that because the `vmcall` is issued by the very same program, the number of matches is generally rather high, while it is usually much lower in real world scenarios where more processes are scheduled between the storing and acquiring of sensitive data.

Cache Sizes Generally the size of the sliding window should be chosen smaller than the cache size to ensure that sensitive information is not exposed in clear to main memory. During our evaluation, the CPU cache was 8 MB which is twice the size needed for our default window size of 1024. It is unlikely that sensitive data is ever exposed to RAM with HyperCrypt running with a sliding window size of half the cache size. If, however, additional security guarantees are needed, existing measures like *Cache as RAM* [109] which is used in CoreBoot, must be used to additionally control the cache. Cache as RAM has also been used to mitigate cold boot attacks against full disk encryption keys in FrozenCache [96] which is capable of running a full Linux system. A commercial virtualization-based solution which leverages Cache as RAM to protect against physical attacks is called *vCage* [131]. Of course, including Cache as RAM as a security guarantee would also come at an additional performance cost. Because Cache as RAM is not necessary for HyperCrypt to protect sensitive data in general, we decided against it and did not include it into our current implementation for performance reasons.

5.2.6 Discussion

HyperCrypt's performance slowdown is caused by both the encryption mechanism and the overhead of the VMM itself. Limiting the times the VMM performs the actual page encryption or decryption would of course lead to a major increase in performance. Therefore, a future version of HyperCrypt could only enable the encryption when it deals with sensitive data and disable it for performance critical operations. This implies, however, that the physical memory must be decrypted and sensitive data must be erased from memory before the encryption is disabled. Further performance gains could be achieved by switching the VMM on and off during system execution, thereby entirely disabling the VMM's trapping mechanism. Stopping the VMM from running without shutting down the guest would require a different underlying design of the VMM though, as BitVisor is implemented as a bare metal hypervisor. Like Rutkowska's proof of concept *Bluepill* [137], the VMM would need to support on-the-fly system virtualization and devirtualization.

5.3 Summary

We presented the design, implementation and evaluation of RamCrypt, a Linux kernel patch that transparently encrypts the address spaces of user mode process under Linux. RamCrypt effectively protects against memory disclosure attacks that give an attacker access to the physical memory and a variety of physical attacks on RAM such as cold boot and DMA attacks. RamCrypt can be easily enabled on a per-process basis without the need for binary rewriting or recompilation, and it is fully compatible to existing systems. If enabled for a single-threaded process with a sliding window size of four, which is a reasonable choice regarding the trade-off between security and performance, it slows processes of the sysbench benchmarking suite down by 25%. Data-intense processes, however, suffer a higher performance drawback as well as multithreaded applications.

To the best of our knowledge, RamCrypt is the first solution that transparently encrypts the address space of user processes within the Linux kernel while being fully compatible to existing applications and storing the key outside RAM.

Furthermore, we presented HyperCrypt, a hypervisor-based solution to transparently encrypt guest memory, including both kernel and user space. We designed and implemented our solution on top of BitVisor using a cold boot resistant AES implementation and took care to handle device memory and DMA buffers securely. The security of HyperCrypt can be adjusted by configuring the size of the sliding window to determine how many pages are left unencrypted within main memory at a time. We evaluated HyperCrypt regarding performance and security and showed that data exposure is effectively decreased. With the default sliding window size of 1024 pages, cold boot attacks are rendered unlikely due to large caches while the reply rate of an nginx web server is 37% lower than for an unprotected system.

6 Conclusion

In this thesis, we described approaches to protect against strong attackers such as physical and root-level attackers. First, we designed trusted computing architectures for embedded devices with a focus on root-level attackers and code confidentiality. Second, we exploited trusted computing architectures for general purpose devices and used them for secure disk encryption and to shield kernel components from each other. We also presented a cache attack to obtain sensitive information from software running within an enclave. Finally, we developed countermeasures against physical memory disclosure attacks as pure software solutions at the operating system and hypervisor level.

Figure 6.1 revisits the categorization of the architectures developed throughout this thesis. First of all, software-based architectures cannot protect against root-level attackers by design, because if an attacker controls the entire software stack he can easily change the software of the architecture itself. Thus, our software-based memory encryption solutions RamCrypt and HyperCrypt protect only against physical attackers. Since no integrity tags are used and confidentiality is guaranteed solely through encryption, they also do not give any integrity guarantees.

Soteria is built upon a protected module architecture with strong isolation guarantees against all software attackers and consequently provides integrity guarantees against root-level attackers. It also provides confidentiality against root-level attackers through isolation at runtime and encryption at load-time. Because code and data reside unencrypted within memory, however, neither confidentiality nor integrity guarantees can be given against physical attackers. To the contrary, Atlas provides confidentiality with the help of a memory encryption unit placed in hardware between cache and main memory and thus, is able to guarantee confidentiality against both, root-level and physical attackers. Due to the lack of integrity tags or an isolation scheme, Atlas cannot give any integrity guarantees though.

Our secure full disk encryption solution Mark has been specifically developed against sophisticated evil maid attacks with repeated physical access and consequently ensures the confidentiality of data stored on the protected disk against physical attackers. Although targeted data manipulations are impossible due to the use of the XTS mode of operation, data manipulations can generally not be prevented since no integrity tags are stored on the disk. Mark also does not protect against root-level attackers, because an attacker controlling the entire software stack could acquire the encryption key stored within CPU registers and decrypt confidential data as well as write arbitrary data to disk.

Our SGX-based solution to shield kernel components from each other protects against both physical and root-level attackers while giving confidentiality and integrity guarantees. It is the only architecture with maximum coverage in the categorization matrix. Note, however, that it highly depends on the application what those guarantees mean. Our prototype implementation for secure full disk encryption, for example, will always guarantee the integrity of the encryption enclave and also protect the confidentiality of the disk encryption key. Nevertheless, a root-level attacker could transparently let decrypt data for him by just using the regular API. Because most probably the data stored on the disk is the most valuable information, physical attacks are also prevented for our prototype implementation while root-level attacks are not strictly prevented.

In general, trusted computing architectures like SGX or the TPM would have maximum coverage in the categorization matrix if considered *individually*. However, the properties offered by a solution, which is built onto an existing trusted computing architecture, highly depends on how the underlying architecture is used and what artefacts need to be protected. For example, Mark protects the integrity of the boot process, but because we use it in a full disk encryption context and data on the disk is not integrity protected, we state that it does not provide integrity guarantees.

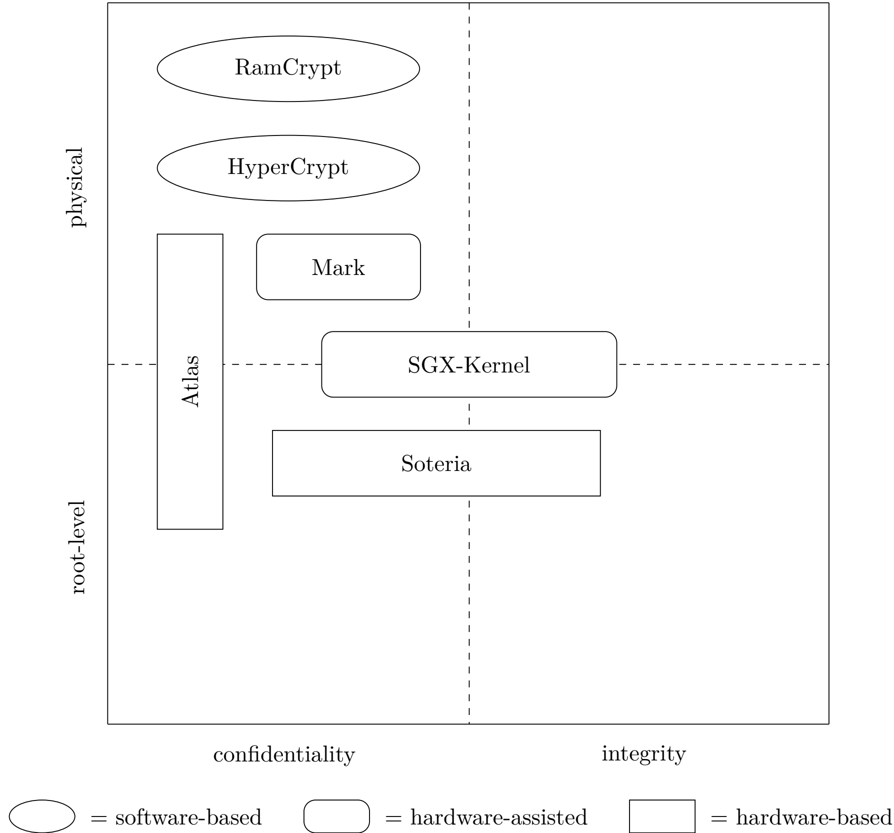

Figure 6.1: Categorization of the architectures developed throughout this thesis with respect to attacker types and security properties.

Even if a trusted computing architecture has maximum coverage in the categorization matrix, some physical attacks such as side channel attacks are often excluded from the attacker model and it is close to impossible to protect against them. For example, although SGX is often expected to secure an application against all kinds of software and hardware attacks with only the CPU package considered trusted, this is not true for cache attacks and forces developers to protect their applications themselves. However, this is contrary to the trend of unmodified legacy applications being ported to enclaves. In the end, implementing software within enclaves remains a critical task for which particular care has to be taken by software developers.

To sum up, while our approaches can be used to defend against strong attackers at different levels, it became obvious that a certain solution can only protect against a subset of attacks. Our software-based approaches against memory disclosure vulnerabilities cannot protect against root-level attackers and the trusted computing architectures designed for embedded devices can only partially protect against physical attackers. Consequently, depending on the particular attacker model a reasonable choice regarding the target architecture has to be made.

Bibliography

[1] ftpd core dump bug, October 1996. URL http://insecure.org/sploits/ftpd.pasv.html.

[2] FTP server core problem, November 1997. URL http://insecure.org/sploits/solaris.secdynamics.core.html.

[3] Coredump hole in imapd and ipop3d, February 1998. URL http://insecure.org/sploits/slackware.ipop.imap.core.html.

[4] Root exploit on Exynos, December 2012. URL http://forum.xda-developers.com/showthread.php?t=2048511.

[5] TrouSerS: The open-source TCG Software Stack, 2013. URL http://trousers.sourceforge.net/.

[6] Exploiting the DRAM rowhammer bug to gain kernel privileges, March 2015. URL https://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[7] Dirty COW (CVE-2016-5195), October 2016. URL https://dirtycow.ninja.

[8] OpenSSL 1.1.0 EVP-AES implementation, July 2016. URL https://github.com/openssl/openssl/blob/9515accaf973ad314e03bdc048b13ec3509cb2e9/crypto/evp/e_aes.c.

[9] Intel SGX SDK AES implementaion, December 2016. URL https://github.com/01org/linux-sgx/blob/0fb9f47e784261369c52c1b49d1484f34409ecaf/external/crypto_px/sources/ippcp/src/pcprij128safeenc2pxca.c.

[10] Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. Technical report, Intel Corporation, 2013. URL https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing.

[11] Andreas Galauner. EFI Rootkits: Pwning your OS before it's even running. In *SIGINT*, 2012. URL https://papers.put.as/papers/macosx/2012/efi_rootkits.pdf.

[12] Maxim Anikeev, Felix C. Freiling, Johannes Götzfried, and Tilo Müller. Secure garbage collection: Preventing malicious data harvesting from deallocated Java objects inside the Dalvik VM. *J. Inf. Sec. Appl.*, 22:81–86, 2015. doi: 10.1016/j.jisa.2014.10.001. URL https://doi.org/10.1016/j.jisa.2014.10.001.

[13] *ARM Security Technology – Building a Secure System Using TrustZone Technology.* ARM Limited, prd29-genc-009492c edition, 2009. URL http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[14] ARM Limited. SSL Library mbed TLS, May 2015. URL https://tls.mbed.org/.

[15] Dmitri Asonov and Rakesh Agrawal. Keyboard Acoustic Emanations. In *2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA*, pages 3–11. IEEE Computer Society, 2004. doi: 10.1109/SECPRI.2004.1301311. URL https://doi.org/10.1109/SECPRI.2004.1301311.

[16] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2001. doi: 10.1007/3-540-44647-8_1. URL https://doi.org/10.1007/3-540-44647-8_1.

[17] Bastian Reitemeier. Palinopsia: Reconstruction of FrameBuffers from VRAM, March 2015. URL https://hsmr.cc/palinopsia/.

[18] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 267–283. USENIX Association, 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[19] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013. URL http://eprint.iacr.org/2013/404.

[20] Eli Bendersky. pyelftools – Library for analyzing ELF files and DWARF debugging information, February 2015. URL https://github.com/eliben/pyelftools.

[21] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl: A Crypto Library in 100 Tweets. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology - LATINCRYPT 2014 - Third International Conference on Cryptology and Information Security in Latin America, Florianópolis, Brazil, September 17-19, 2014, Revised Selected Papers*, volume 8895 of *Lecture Notes in Computer Science*, pages 64–83. Springer, 2014. doi: 10.1007/978-3-319-16295-9_4. URL https://doi.org/10.1007/978-3-319-16295-9_4.

[22] Erik-Oliver Blass and William Robertson. TRESOR-HUNT: attacking CPU-bound encryption. In Robert H'obbes' Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 71–78. ACM, 2012. doi: 10.1145/2420950.2420961. URL http://doi.acm.org/10.1145/2420950.2420961.

[23] Benjamin Böck. Firewire-based Physical Security Attacks on Windows 7, EFS and BitLocker. Technical report, Secure Business Austria Research Lab, August 2009. URL https://www.helpnetsecurity.com/dl/articles/windows7_firewire_physical_attacks.pdf.

[24] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. Spongent: A Lightweight Hash Function. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 312–325. Springer, 2011. doi: 10.1007/978-3-642-23951-9_21. URL https://doi.org/10.1007/978-3-642-23951-9_21.

[25] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A Low-latency Block Cipher for Pervasive Computing Applications (Full version). *IACR Cryptology ePrint Archive*, 2012:529, 2012. URL http://eprint.iacr.org/2012/529.

[26] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In William Enck and Collin Mulliner, editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017.* USENIX Association, 2017. URL https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[27] Franz Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 34:1–34:6. ACM, 2015. doi: 10.1145/2744769.2744922. URL `http://doi.acm.org/10.1145/2744769.2744922`.

[28] M. Brenner, J. Wiebelitz, G. von Voigt, and M. Smith. Secret program execution in the cloud applying homomorphic encryption. In *5th IEEE International Conference on Digital Ecosystems and Technologies (IEEE DEST 2011)*, pages 114–119, May 2011. doi: 10.1109/DEST.2011.5936608. URL `https://doi.org/10.1109/DEST.2011.5936608`.

[29] Peter T. Breuer and Jonathan P. Bowen. A Fully Homomorphic Crypto-Processor Design. In Jan Jürjens, Benjamin Livshits, and Riccardo Scandariato, editors, *Engineering Secure Software and Systems - 5th International Symposium, ESSoS 2013, Paris, France, February 27 - March 1, 2013. Proceedings*, volume 7781 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2013. doi: 10.1007/978-3-642-36563-8_9. URL `https://doi.org/10.1007/978-3-642-36563-8_9`.

[30] Randal E. Bryant and David R. O'Hallaron. *Computer systems - a programmers perspective*. Pearson Education, 2003. ISBN 978-0-13-178456-7.

[31] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, 2017. USENIX Association. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck`.

[32] Ted Bullock. httperf, January 2016. URL `http://sourceforge.net/projects/httperf`.

[33] Ran Canetti and Ronny Ramzi Dakdouk. Obfuscating Point Functions with Multibit Output. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2008. doi: 10.1007/978-3-540-78967-3_28. URL `https://doi.org/10.1007/978-3-540-78967-3_28`.

[34] Brian D. Carrier and Eugene H. Spafford. Getting Physical with the Digital Investigation Process. *IJDE*, 2(2), 2003. URL `http://www.utica.edu/academic/institutes/ecii/publications/articles/A0AC5A7A-FB6C-325D-BF515A44FDEE7459.pdf`.

[35] David Champagne and Ruby B. Lee. Scalable architectural support for trusted software. In Matthew T. Jacob, Chita R. Das, and Pradip Bose, editors, *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*, pages 1–12. IEEE Computer Society, 2010. doi: 10.1109/HPCA.2010.5416657. URL `https://doi.org/10.1109/HPCA.2010.5416657`.

[36] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992. doi: 10.1109/4.126534. URL `https://doi.org/10.1109/4.126534`.

[37] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990. doi: 10.1109/52.43044. URL `https://doi.org/10.1109/52.43044`.

[38] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation (Awarded Best Paper!). In Matt Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 321–336. USENIX, 2004. URL `http://www.usenix.org/publications/library/proceedings/sec04/tech/chow.html`.

[39] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005. URL https://www.usenix.org/conference/14th-usenix-security-symposium/shredding-your-garbage-reducing-data-lifetime-through.

[40] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[41] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016. URL http://eprint.iacr.org/2016/086.

[42] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 857–874. USENIX Association, 2016. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[43] Joan Daemen and Vincent Rijmen. The Block Cipher Rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, volume 1820 of *Lecture Notes in Computer Science*, pages 277–284. Springer, 1998. doi: 10.1007/10721064_26. URL https://doi.org/10.1007/10721064_26.

[44] Ruan de Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. Secure interrupts on low-end microcontrollers. In *IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014, Zurich, Switzerland, June 18-20, 2014*, pages 147–152. IEEE Computer Society, 2014. doi: 10.1109/ASAP.2014.6868649. URL https://doi.org/10.1109/ASAP.2014.6868649.

[45] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. SOFIA: Software and control flow integrity architecture. *Computers & Security*, 68:16–35, 2017. doi: 10.1016/j.cose.2017.03.013. URL https://doi.org/10.1016/j.cose.2017.03.013.

[46] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Information Theory*, 29(2):198–207, 1983. doi: 10.1109/TIT.1983.1056650. URL https://doi.org/10.1109/TIT.1983.1056650.

[47] Gabor Drescher, Christoph Erhardt, Felix C. Freiling, Johannes Götzfried, Daniel Lohmann, Pieter Maene, Tilo Müller, Ingrid Verbauwhede, Andreas Weichslgartner, and Stefan Wildermann. Providing security on demand using invasive computing. *it - Information Technology*, 58(6):281–295, 2016. doi: 10.1515/itit-2016-0032. URL https://doi.org/10.1515/itit-2016-0032.

[48] Guillaume Duc and Ronan Keryell. CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*, pages 483–492. IEEE Computer Society, 2006. doi: 10.1109/ACSAC.2006.21. URL https://doi.org/10.1109/ACSAC.2006.21.

[49] Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 61–75. USENIX Association, 2012. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/dunn.

[50] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and CMAC. Special Publication 800-38D, National Institute of Standards and Technology (NIST), November 2007. URL http://dx.doi.org/10.6028/NIST.SP.800-38D.

[51] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. Special Publication 800-38E, National Institute of Standards and Technology (NIST), January 2010. URL http://dx.doi.org/10.6028/NIST.SP.800-38E.

[52] Ebfe. Ebfe's Anti-BOOTKIT Project, March 2010. URL http://ebfes.wordpress.com/tag/bootloader/.

[53] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. URL http://www.internetsociety.org/smart-secure-and-minimal-architecture-establishing-dynamic-root-trust.

[54] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry V. Ponomarev, Nael B. Abu-Ghazaleh, and Ryan Riley. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 190–202. IEEE Computer Society, 2014. doi: 10.1109/MICRO.2014.25. URL https://doi.org/10.1109/MICRO.2014.25.

[55] FIPS. Advanced Encryption Standard (AES). FIPS PUB 197, National Institute of Standards and Technology (NIST), November 2001. URL https://doi.org/10.6028/NIST.FIPS.197.

[56] Agner Fog. Test programs for measuring clock cycles and performance monitoring, May 2017. URL http://www.agner.org/optimize/#testp.

[57] Pierre-Alain Fouque, Gwenaëlle Martinet, Frédéric Valette, and Sébastien Zimmer. On the Security of the CCM Encryption Mode and of a Slight Variant. In Steven M. Bellovin, Rosario Gennaro, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, 6th International Conference, ACNS 2008, New York, NY, USA, June 3-6, 2008. Proceedings*, volume 5037 of *Lecture Notes in Computer Science*, pages 411–428, 2008. doi: 10.1007/978-3-540-68914-0_25. URL https://doi.org/10.1007/978-3-540-68914-0_25.

[58] Aurélien Francillon, Quan Nguyen, Kasper Bonne Rasmussen, and Gene Tsudik. A minimalist approach to Remote Attestation. In Gerhard Fettweis and Wolfgang Nebel, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014. doi: 10.7873/DATE.2014.257. URL https://doi.org/10.7873/DATE.2014.257.

[59] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49. IEEE Computer Society, 2013. doi: 10.1109/FOCS.2013.13. URL https://doi.org/10.1109/FOCS.2013.13.

[60] Behrad Garmany and Tilo Müller. PRIME: private RSA infrastructure for memory-less encryption. In Charles N. Payne Jr., editor, *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 149–158. ACM, 2013. doi: 10.1145/2523649.2523656. URL http://doi.acm.org/10.1145/2523649.2523656.

[61] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009. doi: 10.1145/1536414.1536440. URL http://doi.acm.org/10.1145/1536414.1536440.

[62] Olivier Girard. OpenCores: openMSP430, January 2009. URL http://opencores.org/project,openmsp430.

[63] Brian Gladman. AES and Combined Encryption/Authentication Modes, 2001. URL http://www.gladman.me.uk/AES.

[64] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996. doi: 10.1145/233551.233553. URL http://doi.acm.org/10.1145/233551.233553.

[65] Johannes Götzfried and Tilo Müller. ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, pages 161–168, 2013. doi: 10.1109/ARES.2013.23. URL https://doi.org/10.1109/ARES.2013.23.

[66] Johannes Götzfried and Tilo Müller. Fast Software Encryption with SIMD: How to speed up symmetric block ciphers with the AVX/AVX2 instruction set. In *Proceedings of the 6th European Workshop on System Security, EUROSEC 2013, Prague, Czech Republic, April 14, 2013*. ACM, 2013. URL http://www1.cs.fau.de/filepool/projects/avx.crypto/avxcrypto.pdf.

[67] Johannes Götzfried and Tilo Müller. Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption. *ACM Trans. Inf. Syst. Secur.*, 17(2):6:1–6:23, 2014. doi: 10.1145/2663348. URL http://doi.acm.org/10.1145/2663348.

[68] Johannes Götzfried and Tilo Müller. Analysing Android's Full Disk Encryption Feature. *JoWUA*, 5(1):84–100, 2014. URL http://isyou.info/jowua/papers/jowua-v5n1-4.pdf.

[69] Johannes Götzfried, Johannes Hampel, and Tilo Müller. Physically Secure Code and Data Storage in Autonomously Booting Systems. In *10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015*, pages 199–204. IEEE Computer Society, 2015. doi: 10.1109/ARES.2015.19. URL https://doi.org/10.1109/ARES.2015.19.

[70] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix C. Freiling, and Ingrid Verbauwhede. Soteria: Offline Software Protection within Low-cost Embedded Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 241–250. ACM, 2015. doi: 10.1145/2818000.2856129. URL http://doi.acm.org/10.1145/2818000.2856129.

[71] Johannes Götzfried, Nico Dörr, Ralph Palutke, and Tilo Müller. HyperCrypt: Hypervisor-Based Encryption of Kernel and User Space. In *11th International Conference on Availability, Reliability and Security, ARES 2016, Salzburg, Austria, August 31 - September 2, 2016*, pages 79–87. IEEE Computer Society, 2016. doi: 10.1109/ARES.2016.13. URL https://doi.org/10.1109/ARES.2016.13.

[72] Johannes Götzfried, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes. RamCrypt: Kernel-based Address Space Encryption for User-mode Processes. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 919–924. ACM, 2016. doi: 10.1145/2897845.2897924. URL http://doi.acm.org/10.1145/2897845.2897924.

[73] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In Cristiano Giuffrida and Angelos Stavrou, editors, *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*, pages 2:1–2:6. ACM, 2017. doi: 10.1145/3065913.3065915. URL http://doi.acm.org/10.1145/3065913.3065915.

[74] David Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach.* Intel Press, 1st edition, 2009. URL http://dl.acm.org/citation.cfm?id=1610416.

[75] Michael Gruhn and Tilo Müller. On the Practicability of Cold Boot Attacks. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, pages 390–397, 2013. doi: 10.1109/ARES.2013.52. URL http://dx.doi.org/10.1109/ARES.2013.52.

[76] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. Copker: Computing with Private Keys without RAM. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. URL http://www.internetsociety.org/doc/copker-computing-private-keys-without-ram.

[77] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, California, USA, May 17-21, 2015*, pages 3–19. IEEE Computer Society, 2015. doi: 10.1109/SP.2015.8. URL https://doi.org/10.1109/SP.2015.8.

[78] Shay Gueron. Intel's New AES Instructions for Enhanced Performance and Security. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2009. doi: 10.1007/978-3-642-03317-9_4. URL https://doi.org/10.1007/978-3-642-03317-9_4.

[79] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505. IEEE Computer Society, 2011. doi: 10.1109/SP.2011.22. URL https://doi.org/10.1109/SP.2011.22.

[80] Peter Gutmann. Data Remanence in Semiconductor Devices. In Dan S. Wallach, editor, *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*. USENIX, 2001. URL http://www.usenix.org/publications/library/proceedings/sec01/gutmann.html.

[81] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60. USENIX Association, 2008. URL http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf.

[82] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009. doi: 10.1145/1506409.1506429. URL http://doi.acm.org/10.1145/1506409.1506429.

[83] Keith Harrison and Shouhuai Xu. Protecting Cryptographic Keys from Memory Disclosure Attacks. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 137–143. IEEE Computer Society, 2007. doi: 10.1109/DSN.2007.77. URL https://doi.org/10.1109/DSN.2007.77.

[84] Nadia Heninger and Hovav Shacham. Reconstructing RSA Private Keys from Random Key Bits. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009. doi: 10.1007/978-3-642-03356-8_1. URL https://doi.org/10.1007/978-3-642-03356-8_1.

[85] Michael Henson and Stephen Taylor. Beyond Full Disk Encryption: Protection on Security-Enhanced Commodity Processors. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2013. doi: 10.1007/978-3-642-38980-1_19. URL https://doi.org/10.1007/978-3-642-38980-1_19.

[86] Michael Henson and Stephen Taylor. Memory encryption: A survey of existing techniques. *ACM Comput. Surv.*, 46(4):53:1–53:26, 2013. doi: 10.1145/2566673. URL http://doi.acm.org/10.1145/2566673.

[87] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, page 11. ACM, 2013. doi: 10.1145/2487726.2488370. URL http://doi.acm.org/10.1145/2487726.2488370.

[88] *Intel Software Guard Extensions Programming Reference*. Intel Corporation, 329298-002us edition, October 2014. URL https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[89] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 325462-063us edition, July 2017. URL https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[90] Blake Ives, Kenneth R. Walsh, and Helmut Schneider. The domino effect of password reuse. *Commun. ACM*, 47(4):75–78, 2004. doi: 10.1145/975817.975820. URL http://doi.acm.org/10.1145/975817.975820.

[91] Joanna Rutkowska. Evil Maid goes after TrueCrypt, October 2009. URL http://theinvisiblethings.blogspot.de/2009/10/evil-maid-goes-after-truecrypt.html. The Invisible Things Lab.

[92] Joanna Rutkowska. Anti Evil Maid, September 2011. URL http://theinvisiblethings.blogspot.de/2011/09/anti-evil-maid.html. The Invisible Things Lab.

[93] Moritz Jodeit and Martin Johns. USB Device Drivers: A Stepping Stone into Your Kernel. In *2010 European Conference on Computer Network Defense*, pages 46–52, October 2010. doi: 10.1109/EC2ND.2010.16. URL https://doi.org/10.1109/EC2ND.2010.16.

[94] John Heasman. Implementing and Detecting an ACPI BIOS Rootkit. In *BlackHat Briefings, Europe*, 2006. URL https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf.

[95] Simon P. Johnson, Vincent R. Scarlata, Carlos V. Rozas, Ernie Brickell, and Frank McKeen. Intel SGX: EPID Provisioning and Attestation Services. Technical report, Intel Corporation, March 2016. URL https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services.

[96] Jürgen Pabel. FrozenCache - Mitigating cold-boot attacks for Full-Disk-Encryption software. In *27th Chaos Communication Congress*, Berlin, Germany, December 2010. CCC. URL https://events.ccc.de/2010/12/28/frozen-cache.

[97] B. Kaliski. PKCS #5: Password-Based Cryptography Specification. Number 2898 in Request for Comments. Internet Engineering Task Force, September 2000. URL https://www.ietf.org/rfc/rfc2898.txt.

[98] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, pages 2:1–2:10. ACM, 2017. doi: 10.1145/3098954.3098995. URL http://doi.acm.org/10.1145/3098954.3098995.

[99] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. Technical report, AMD, April 2016. URL http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[100] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: a security architecture for tiny embedded devices. In Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel, editors, *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 10:1–10:14. ACM, 2014. doi: 10.1145/2592798.2592824. URL http://doi.acm.org/10.1145/2592798.2592824.

[101] Kokke. Small portable AES128/192/256 in C, February 2015. URL https://github.com/kokke/tiny-AES128-C.

[102] Markus G. Kuhn. Optical Time-Domain Eavesdropping Risks of CRT Displays. In *2002 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 12-15, 2002*, pages 3–18. IEEE Computer Society, 2002. doi: 10.1109/SECPRI.2002.1004358. URL https://doi.org/10.1109/SECPRI.2002.1004358.

[103] Klaus Kursawe, Dries Schellekens, and Bart Preneel. Analyzing Trusted Platform Communication. In *ECRYPT Workshop, CRASH - CRyptographic Advances in Secure Hardware*, page 8, 2005. URL https://www.esat.kuleuven.be/cosic/publications/article-591.pdf.

[104] Donald C. Latham. *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, 1985. URL http://csrc.nist.gov/publications/history/dod85.pdf.

[105] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. doi: 10.1109/CGO.2004.1281665. URL https://doi.org/10.1109/CGO.2004.1281665.

[106] Xiang Li, Yan Wen, Minhuan Huang, and Qiang Liu. An Overview of Bootkit Attacking Approaches. In Junliang Chen, Huadong Ma, and Ivan Stojmenovic, editors, *Seventh International Conference on Mobile Ad-hoc and Sensor Networks, MSN 2011, Beijing, China, December 16-18, 2011*, pages 428–431. IEEE Computer Society, 2011. doi: 10.1109/MSN.2011.19. URL https://doi.org/10.1109/MSN.2011.19.

[107] Moses Liskov, Ronald L. Rivest, and David A. Wagner. Tweakable Block Ciphers. *J. Cryptology*, 24(3):588–613, 2011. doi: 10.1007/s00145-010-9073-y. URL https://doi.org/10.1007/s00145-010-9073-y.

[108] Loukas K. DE MYSTERIIS DOM JOBSIVS – Mac EFI Rootkits. In *BlackHat Conference Proceedings, USA*, 2012. URL http://ho.ax/De_Mysteriis_Dom_Jobsivs_Black_Hat_Paper.pdf.

[109] Yinghai Lu, Li-Ta Lo, Gregory R. Watson, and Ronald G. Minnich. CAR: Using Cache as RAM in LinuxBIOS. Technical report, Advanced Computing Laboratory, September 2005. URL https://www.coreboot.org/data/yhlu/cache_as_ram_lb_09142006.pdf.

[110] Pieter Maene and Ingrid Verbauwhede. Single-Cycle Implementations of Block Ciphers. In Tim Güneysu, Gregor Leander, and Amir Moradi, editors, *Lightweight Cryptography for Security and Privacy - 4th International Workshop, LightSec 2015, Bochum, Germany, September 10-11, 2015, Revised Selected Papers*, volume 9542 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2015. doi: 10.1007/978-3-319-29078-2_8. URL https://doi.org/10.1007/978-3-319-29078-2_8.

[111] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers*, PP(99):1–1, 2017. doi: 10.1109/TC.2017.2647955. URL https://doi.org/10.1109/TC.2017.2647955.

[112] Andrew Martin. The Ten Page Introduction to Trusted Computing. Technical Report RR-08-11, OUCL, December 2008. URL http://www.cs.ox.ac.uk/files/1873/RR-08-11.PDF.

[113] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. In Joseph S. Sventek and Steven Hand, editors, *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 315–328. ACM, 2008. doi: 10.1145/1352592.1352625. URL http://doi.acm.org/10.1145/1352592.1352625.

[114] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 143–158. IEEE Computer Society, 2010. doi: 10.1109/SP.2010.17. URL https://doi.org/10.1109/SP.2010.17.

[115] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, page 10. ACM, 2013. doi: 10.1145/2487726.2488368. URL http://doi.acm.org/10.1145/2487726.2488368.

[116] Microsoft Corporation. Windows BitLocker Drive Encryption: Technical Overview, July 2009. URL https://technet.microsoft.com/en-us/library/cc732774(v=ws.10).aspx.

[117] Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. In *BlackHat Briefings, Las Vegas, Nevada, USA*, August 2015. URL http://illmatics.com/Remote%20Car%20Hacking.pdf.

[118] Tilo Müller, Felix C. Freiling, and Andreas Dewald. TRESOR Runs Encryption Securely Outside RAM. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. URL http://static.usenix.org/events/sec11/tech/full_papers/Muller.pdf.

[119] Tilo Müller, Tobias Latzo, and Felix Freiling. Hardware-based Full Disk Encryption (In)Security Survey. Technical report, Friedrich-Alexander University of Erlangen-Nuremberg, September 2012. URL https://www1.cs.fau.de/sed.

[120] Tilo Müller, Benjamin Taubmann, and Felix C. Freiling. TreVisor - OS-Independent Software-Based Full Disk Encryption Secure against Main Memory Attacks. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, volume 7341 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2012. doi: 10.1007/978-3-642-31284-7_5. URL https://doi.org/10.1007/978-3-642-31284-7_5.

[121] Tilo Müller, Hans Spath, Richard Mäckl, and Felix C. Freiling. Stark - Tamperproof Authentication to Resist Keylogging. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, volume 7859 of *Lecture Notes in Computer Science*, pages 295–312. Springer, 2013. doi: 10.1007/978-3-642-39884-1_25. URL https://doi.org/10.1007/978-3-642-39884-1_25.

[122] Michael Neve and Jean-Pierre Seifert. Advances on Access-Driven Cache Attacks on AES. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*, volume 4356 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006. doi: 10.1007/978-3-540-74462-7_11. URL https://doi.org/10.1007/978-3-540-74462-7_11.

[123] NGINX Inc. NGINX: High Performance Load Balancer, Web Server, and Reverse Proxy, January 2016. URL http://nginx.com/.

[124] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 479–494. USENIX Association, 2013. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/noorman.

[125] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.*, 20(3): 7:1–7:33, 2017. doi: 10.1145/3079763. URL http://doi.acm.org/10.1145/3079763.

[126] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. doi: 10.1007/11605805_1. URL https://doi.org/10.1007/11605805_1.

[127] T. Paul Parker and Shouhuai Xu. A Method for Safekeeping Cryptographic Keys from Memory Disclosure Attacks. In Liqun Chen and Moti Yung, editors, *Trusted Systems, First International Conference, INTRUST 2009, Beijing, China, December 17-19, 2009. Revised Selected Papers*, volume 6163 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 2009. doi: 10.1007/978-3-642-14597-1_3. URL https://doi.org/10.1007/978-3-642-14597-1_3.

[128] Peter Kleissner. Stoned Bootkit. In *BlackHat Conference Proceedings, USA*, July 2009. URL http://www.blackhat.com/presentations/bh-usa-09/KLEISSNER/BHUSA09-Kleissner-StonedBootkit-SLIDES.pdf.

[129] P. A. H. Peterson. Cryptkeeper: Improving security with encrypted RAM. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, pages 120–126, November 2010. doi: 10.1109/THS.2010.5655081. URL https://doi.org/10.1109/THS.2010.5655081.

[130] PJRC Electronic Projects. Teensy USB Development Board, 2013. URL http://www.pjrc.com/teensy/.

[131] PrivateCore. Trustworthy Cloud Computing with vCage, August 2014. URL https://privatecore.com/vcage.

[132] Niels Provos. Encrypting Virtual Memory. In Steven M. Bellovin and Greg Rose, editors, *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000.* USENIX Association, 2000. URL https://www.usenix.org/conference/9th-usenix-security-symposium/encrypting-virtual-memory.

[133] Joel Reardon, David A. Basin, and Srdjan Capkun. On Secure Data Deletion. *IEEE Security & Privacy*, 12(3):37–44, 2014. doi: 10.1109/MSP.2013.159. URL https://doi.org/10.1109/MSP.2013.159.

[134] Lars Richter, Johannes Götzfried, and Tilo Müller. Isolating Operating System Components with Intel SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution, SysTEX@Middleware 2016, Trento, Italy, December 12, 2016*, pages 8:1–8:6. ACM, 2016. doi: 10.1145/3007788.3007796. URL http://doi.acm.org/10.1145/3007788.3007796.

[135] Robert David Graham. Thunderbolt: Introducing a new way to hack Macs, February 2011. URL http://erratasec.blogspot.com/2011/02/thunderbolt-introducing-new-way-to-hack.html. Errata Security.

[136] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003. doi: 10.1145/937527.937529. URL http://doi.acm.org/10.1145/937527.937529.

[137] Joanna Rutkowska. Subverting Vista Kernel For Fun And Profit. In *BlackHat Briefings, Las Vegas, Nevada, USA*, August 2006. URL http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.

[138] Anibal L. Sacco and Alfredo A. Ortega. Persistent BIOS Infection: The early bird catches the worm. In *Proceedings of the Annual CanSecWest Applied Security Conference*, Vancouver, British Columbia, Canada, 2009. Core Security Technologies. URL https://www.coresecurity.com/system/files/publications/2016/05/Persistent-BIOS-Infection.pdf.

[139] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM trustzone to build a trusted language runtime for mobile applications. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 67–80. ACM, 2014. doi: 10.1145/2541940.2541949. URL http://doi.acm.org/10.1145/2541940.2541949.

[140] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 38–54. IEEE Computer Society, 2015. doi: 10.1109/SP.2015.10. URL https://doi.org/10.1109/SP.2015.10.

[141] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, volume 10327 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2017. doi: 10.1007/978-3-319-60876-1_1. URL https://doi.org/10.1007/978-3-319-60876-1_1.

[142] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. BitVisor: a thin hypervisor for enforcing i/o device security. In Antony L. Hosking, David F. Bacon, and Orran Krieger, editors, *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009*, pages 121–130. ACM, 2009. doi: 10.1145/1508293.1508311. URL http://doi.acm.org/10.1145/1508293.1508311.

[143] Patrick Simmons. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 73–82. ACM, 2011. doi: 10.1145/2076732.2076743. URL http://doi.acm.org/10.1145/2076732.2076743.

[144] Ryan Slominski. Fast User/Kernel Data Transfer, April 2007. URL https://pdfs.semanticscholar.org/39e7/cbbeb1c2c7df6c78c67f73bae4ba5078b032.pdf. College of William & Mary, Master Thesis.

[145] Raoul Strackx and Frank Piessens. Fides: selectively hardening software application components against kernel-level or process-level malware. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 2–13. ACM, 2012. doi: 10.1145/2382196.2382200. URL http://doi.acm.org/10.1145/2382196.2382200.

[146] Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, and Frank Piessens. Protected Software Module Architectures. In Helmut Reimer, Norbert Pohlmann, and Wolfgang Schneider, editors, *ISSE 2013 - Securing Electronic Business Processes, Highlights of the Information Security Solutions Europe 2013 Conference, Brussels, Belgium, October 22-23, 2013*, pages 241–251. Springer, 2013. doi: 10.1007/978-3-658-03371-2_21. URL https://doi.org/10.1007/978-3-658-03371-2_21.

[147] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In Utpal Banerjee, Kyle Gallivan, and Antonio González, editors, *Proceedings of the 17th Annual International Conference on Supercomputing, ICS 2003, San Francisco, CA, USA, June 23-26, 2003*, pages 160–171. ACM, 2003. doi: 10.1145/782814.782838. URL http://doi.acm.org/10.1145/782814.782838.

[148] G. Edward Suh, Christopher W. Fletcher, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Author retrospective AEGIS: architecture for tamper-evident and tamper-resistant processing. In Utpal Banerjee, editor, *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 68–70. ACM, 2014. doi: 10.1145/2591635.2591665. URL http://doi.acm.org/10.1145/2591635.2591665.

[149] Andrew S. Tanenbaum. *Modern operating systems, 3rd Edition.* Pearson Education, 2008. ISBN 978-0-13-600663-3.

[150] Christopher Tarnovsky. Deconstructing a “Secure” Processor. In *BlackHat Briefings, Washington DC, USA*, February 2010. URL http://www.blackhat.com/presentations/bh-dc-10/Tarnovsky_Chris/BlackHat-DC-2010-Tarnovsky-DASP-slides.pdf.

[151] Texas Instruments. GCC - Open Source Compiler for MSP430 Microcontrollers, February 2015. URL http://www.ti.com/tool/msp430-gcc-opensource.

[152] Texas Instruments. Ultra-Low-Power MSP430 Microcontroller, February 2015. URL http://www.ti.com/msp430.

[153] Shreekant Thakkar and Tom Huff. Internet Streaming SIMD Extensions. *IEEE Computer*, 32(12):26–34, December 1999. doi: 10.1109/2.809248. URL https://doi.org/10.1109/2.809248.

[154] Tim Thornburgh. Social Engineering: The Dark Art. In *Proceedings of the 1st Annual Conference on Information Security Curriculum Development*, pages 133–135. ACM, 2004. doi: 10.1145/1059524.1059554. URL http://doi.acm.org/10.1145/1059524.1059554.

[155] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23(1):37–71, 2010. doi: 10.1007/s00145-009-9049-y. URL https://doi.org/10.1007/s00145-009-9049-y.

[156] TrueCrypt Foundation. TrueCrypt: Free Open-Source On-The-Fly Disk Encryption Software for Windows 7/Vista/XP, Mac OS X and Linux, 2012. URL http://www.truecrypt.org/.

[157] *TPM Main: Part 1 Design Principles.* Trusted Computing Group, Version 1.2, Revision 116 edition, March 2011. URL https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf.

[158] *TPM Main: Part 2 TPM Structures.* Trusted Computing Group, Version 1.2, Revision 116 edition, March 2011. URL https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-2-TPM-Structures_v1.2_rev116_01032011.pdf.

[159] *TPM Main: Part 3 Commands.* Trusted Computing Group, Version 1.2, Revision 116 edition, March 2011. URL https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-3-Commands_v1.2_rev116_01032011.pdf.

[160] *Trusted Platform Module Library: Part 1: Architecture*. Trusted Computing Group, Family 2.0, Level 00, Revision 01.16 edition, October 2014. URL https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.16.pdf.

[161] Meltem Turan, Elaine Barker, William Burr, and Lily Chen. Recommendation for Password-Based Key Derivation. Special Publication 800-132, National Institute of Standards and Technology (NIST), December 2010. URL http://dx.doi.org/10.6028/NIST.SP.800-132.

[162] Sven Türpe, Andreas Poller, Jan Steffan, Jan-Peter Stotz, and Jan Trukenmüller. Attacking the BitLocker Boot Process. In Liqun Chen, Chris J. Mitchell, and Andrew P. Martin, editors, *Trusted Computing, Second International Conference, Trust 2009, Oxford, UK, April 6-8, 2009, Proceedings*, volume 5471 of *Lecture Notes in Computer Science*, pages 183–196. Springer, 2009. doi: 10.1007/978-3-642-00587-9_12. URL https://doi.org/10.1007/978-3-642-00587-9_12.

[163] Hoeteck Wee. On obfuscating point functions. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 523–532. ACM, 2005. doi: 10.1145/1060590.1060669. URL http://doi.acm.org/10.1145/1060590.1060669.

[164] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, volume 9878 of *Lecture Notes in Computer Science*, pages 440–457. Springer, 2016. doi: 10.1007/978-3-319-45744-4_22. URL https://doi.org/10.1007/978-3-319-45744-4_22.

[165] Andreas Weichslgartner, Stefan Wildermann, Johannes Götzfried, Felix C. Freiling, Michael Glaß, and Jürgen Teich. Design-Time/Run-Time Mapping of Security-Critical Applications in Heterogeneous MPSoCs. In Sander Stuijk, editor, *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2016, Sankt Goar, Germany, May 23-25, 2016*, pages 153–162. ACM, 2016. doi: 10.1145/2906363.2906370. URL http://doi.acm.org/10.1145/2906363.2906370.

[166] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). Number 3610 in Request for Comments. Internet Engineering Task Force, September 2003. URL https://www.ietf.org/rfc/rfc3610.txt.

[167] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Intel Performance Counter Monitor - A better way to measure CPU utilization. Technical report, Intel Corporation, January 2017. URL https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[168] Peter Williams and Rick Boivie. CPU Support for Secure Executables. In Jonathan M. McCune, Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, M. Angela Sasse, and Yolanta Beres, editors, *Trust and Trustworthy Computing - 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*, volume 6740 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2011. doi: 10.1007/978-3-642-21599-5_13. URL https://doi.org/10.1007/978-3-642-21599-5_13.

[169] Johannes Winter and Kurt Dietrich. A hijacker's guide to communication interfaces of the trusted platform module. *Computers & Mathematics with Applications*, 65(5):748–761, 2013. doi: 10.1016/j.camwa.2012.06.018. URL https://doi.org/10.1016/j.camwa.2012.06.018.

[170] Alexander Würstlein, Michael Gernoth, Johannes Götzfried, and Tilo Müller. Exzess: Hardware-Based RAM Encryption Against Physical Memory Disclosure. In Frank Hannig, João M. P. Cardoso, Thilo Pionteck, Dietmar Fey, Wolfgang Schröder-Preikschat, and Jürgen Teich, editors, *Architecture of Computing Systems - ARCS 2016 - 29th International Conference, Nuremberg, Germany, April 4-7, 2016, Proceedings*, volume 9637 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2016. doi: 10.1007/978-3-319-30695-7_5. URL `https://doi.org/10.1007/978-3-319-30695-7_5`.

[171] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015. doi: 10.1109/SP.2015.45. URL `https://doi.org/10.1109/SP.2015.45`.