Lehrstuhl für Informatik 1
Friedrich-Alexander-
Universität
Erlangen-Nürnberg

MASTER THESIS

Isolation of Operating System Components with Intel SGX

Lars Richter

Erlangen, May 30, 2016

Examiner:	Prof. Dr. Felix Freiling
Advisor:	Dr.-Ing. Tilo Müller
Advisor:	Johannes Götzfried, M. Sc.

Eidesstattliche Erklärung / Statutory Declaration

Erlangen, May 30, 2016

Lars Richter

Zusammenfassung

Der Schutz eines Computersystems vor bösartigen Programmen und der Abschottung der Programme gegeneinander stellt seit jeher hohe Anforderungen an Entwickler, Softwarearchitekten und Forscher. Viele verschiedene Versuche wurden unternommen, moderne Computersysteme sicher zu gestalten, doch hoch spezialisierte Angriffe durchdringen auch diese Schutzmechanismen. Mit der zunehmenden Verlagerung von Systemen in fremde Rechenzentren gewinnt zusätzlich ein weiterer Angriffsvektor an Bedeutung, denn nur wenige Schutzmechanismen gehen von einem bösartigen Host-Betriebssystem aus.

Intel hat 2013 mit den Software Guard Extensions (Intel SGX) ein Modell vorgestellt, welches die Informationssicherheit in dem Bereich revolutionieren könnte. An statt das Betriebssystem gegen Anwendungen abzusichern, wird nun die Anwendung unter anderem gegen privilegierten Zugriff des Betriebssystems geschützt. Intel SGX erlaubt es einen vertrauenswürdigen Container zu erstellen, welcher weder vom Betriebssystem noch von anderen Applikationen modifiziert und ausgelesen werden kann. Mit der Veröffentlichung der SGX-fähigen Prozessoren im Oktober 2015 und der notwendigen Entwickler-Tools im Januar 2016 ist es möglich erste Anwendungen für SGX zu entwickeln.

Die Forschung um die Intel Software Guard Extensions befindet sich in den Anfängen. Aus diesem Grund ist eine kritische Auseinandersetzung mit dem Design und Implementation von Intel SGX notwendig. In dieser Arbeit wird Intel SGX analysiert, auf die Grenzen hin getestet und auf die Nutzbarkeit geprüft. Erste Anwendungsfälle werden vorgestellt und kritisch bewertet.

Es wird erstmalig der Ansatz vorgestellt Betriebsystemkomponenten des Linux Kernels mit Intel SGX abzusichern. Das System *TresorSGX* wird beschrieben, welches die Funktionalität eines Kernel Modules in ein geschützten SGX Container auslagert und damit dieses Modul gegen unberechtigten Zugriff absichert. TresorSGX bietet in einem Kernel Modul eine Verschlüsselungsmethode für die Linux Crypto API an, welche geschützt in einem SGX Container ausgeführt wird. Mit diesem System ist der Schlüssel des Algorithmus zu keiner Zeit auslesbar. Es ist dadurch resistent gegen *cold-boot* und *DMA* Angriffe.

Die daraus gewonnenen Erkenntnisse über das Auslagern von Kernel Komponenten in SGX Container können als Basis für künftige Forschungsthemen in dem Umfeld dienen.

Abstract

The defence of a computer system against malicious applications and the isolation of applications against each other has always been a challenge for developers, software architects and researchers. Different attempts were made to design secure systems. However, highly sophisticated attacks manage to overcome those security measurements. With the migration of computer systems into foreign datacenters another attack vector gains influence because only a few safety mechanisms take malicious host operating systems into consideration.

In 2013 Intel proposed the Software Guard Extensions (SGX) as new model to secure applications. Instead of securing the operating system against applications the applications are protected against privileged unauthorised access. Intel SGX provides a secured trustworthy container which cannot be accessed, read or modified by any unauthorised party. With the release of the SGX-capable processors in October 2015 and the required developer tools in January 2016 it is possible to develop first SGX applications.

The research about the Intel Software Guard Extensions is in the very beginning. Therefore, a critical examination of the design and implementation of SGX is required. In this thesis SGX will be analysed, its limits tested and the usability examined. Available use cases for SGX will be presented and critically evaluated.

A new attempt to isolate and secure operating system components of the Linux kernel will be introduced. The system *TresorSGX* which outsources a functionality of a kernel module to a SGX container will be described. TresorSGX provides a cryptographic algorithm for the Linux Crypto API which is executed in a secured container. The cryptographic key material is guarded from unauthorised access of unprivileged and privileged components at any time. This protects the disk-encryption system from *cold-boot* and *DMA* attacks.

The gained insights about the migration of kernel components into SGX containers can serve as foundation for future research in that field.

CONTENTS

1 INTRODUCTION

The protection of a computer system against malicious applications and the isolation of software components has been a difficult task for software developers, software architects and researchers since the beginning. Many attempts were made to secure modern systems but the rising complexity and dependency of big applications allow highly sophisticated attacks to succeed. With the current trend of outsourcing the physical layer or operating system layer of a datacenter to foreign resources another attack vector is rising because most security considerations assume that the physical layer can be trusted.

When discussing security related topics the broad term *security* has to be explained at first. Its meaning depends heavily on the field where it is applied. In general, the security guarantees that an asset is defended against a set of attacks. The asset can be the system itself or information. The attacker is trying to interfere with the security properties of the asset.

The basic security properties of an asset are *Confidentiality*, *Integrity*, *Availability* and *Authenticity*. The confidentiality guarantees that the asset can not be obtained by an unauthorized entity. A standard measure to achieve confidentiality is to encrypt sensitive data. The encryption keys itself become confidential assets too. The second security property is the integrity of data. Data must be complete and not be modified by an unauthorized entity. By using hashing and signature techniques alternations of data can be detected. The availability of an asset guarantees that an user can access the information or the system if he needs to. A classic attack against availability is the denial-of-service attack which leads to a complete unavailability of the system. By using redundancy and additional security measures the availability can be increased. The authenticity of an asset gains more and more significance. It ensures that the asset can proove its identity and its properties against a challenging user. If an attacker tries to impersonate the asset, the user will detect it. In the world wide web authenticity is guaranteed by using public key cryptography and digital certificates. The user trusts root certificate authorities and therefore trusts entities which proof their identity by signing with issued certificates. These basic security properties are applied to the trusted computing field too.

In 2013 Intel published the Intel Software Guard Extensions (Intel SGX) in a series of papers Hoekstra et al. [20] McKeen et al. [44] Anati et al. [1]. They proposed a new programming model which uses containers, so called *enclaves*, which can only be accessed by the CPU and no other party of the system. Furthermore neither the operating system nor a hypervisor can read the content of the enclave at runtime. In the light of the current movement into the *Cloud* this programming model looks really promising to retain the security, integrity and confidentiality of on-site datacenters.

The Intel Skylake processor, released in October 2015, allows the creation and execution of SGX enclaves via its extended instruction set. Multiple researchers published first ideas and proof of concepts of how applications can benefit from SGX Kim et al. [37]Baumann et al. [3]Ohrimenko et al. [48].

1.1. Motivation

The most of the security systems and security analysis assume that the host system can be trusted and that an attacker enters the system from the outside, for example the network or the periphery of the computer. Even if a malicious application is executed at the system, most of the research assume that at least the operating system can be trusted. Because of malware that affects the hypervisor and recurring security vulnerabilities in operating systems it becomes clear that this assumption can not be trusted any longer when our daily live depends on numerous computing systems.

Furthermore mobile devices are available which allow the execution of security critical applications like online banking. These platforms are becoming more and more open that users are able to install custom applications and operating systems, downloaded from the internet. These applications can be malicious to either trick the user to insert confidential information or to exploit other applications on the device.

The security model of Intel SGX assumes that only the CPU can be trusted and other applications and the host operating system per se can be malicious as shown in Figure 1.1. Secured containers (trusted enclaves) can be loaded into a protected memory area which can only be accessed by the CPU. No malicious application, operating system or virtual machine monitor can access the information in the secured container. The memory of the container is encrypted in the CPU to defend against observation via the hardware. These

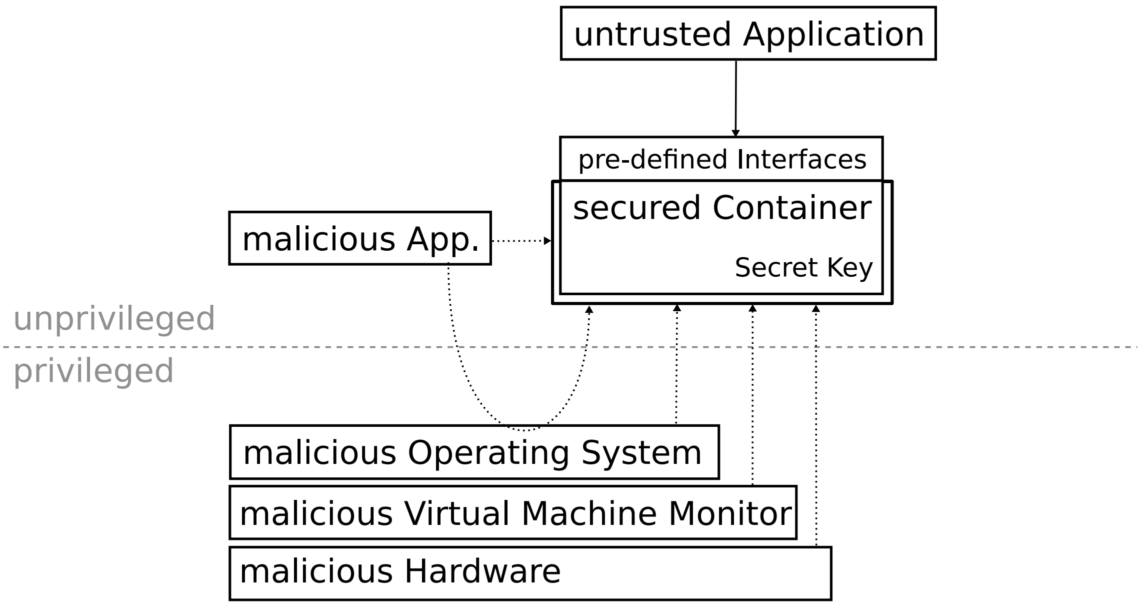

Figure 1.1.: The secured container defends against unauthorized access and allows access only over pre-defined interfaces

containers (enclaves) can be attested and remotely verified that they are unmodified. This allows the distribution and execution of trusted enclaves at scale.

The Intel software guard extensions have the potential to overcome known problems and risks regarding critical applications at unknown hosts. Intel itself proposed use cases for the new technology Hoekstra et al. [20]. They developed a software based *one-time password token generators* and secured *Enterprise Rights Management Clients*. Other researchers developed a model to *remote attestate* routing controllers of the Tor network to guarantee their integrity. Other use cases and filed patents will be discussed in section 2.1.8.

A custom enclave can only be launched with a special Launch Enclave from Intel. This fact was discovered during the examination of the SGX documentation in late 2015. At that time the only possibility to gain experience with the SGX software model was to use the OpenSGX emulator [15]. Since January 2016, the SGX SDK for Windows was made available to the public, allowing practical research based on real hardware.

The main use case for SGX is to include small functions into the enclave to harden the codebase against different attacks. In this thesis SGX is used to improve the security of the Linux Kernel. With TRESOR Müller et al. [46], Crypto API implementation of AES, it is able to withstand cold boot attacks on memory. This approach will be continued with the Software Guard Extensions. Instead of saving the cryptographic key material in CPU registers, the key will be saved in an enclave. This use case is the basis of a first proof of concept for the usage of Intel SGX in the Linux Kernel.

1.2. Task

The task of this thesis is divided into three main parts. In the beginning there has not been an official documentation apart from the SGX white papers and the Instruction Set Reference. During the elaboration of the thesis more and more official documentation were published.

1.2.1. Analysis of SGX

The first task was to analyse SGX from the 2013 white papers Hoekstra et al. [20] McKeen et al. [44] Anati et al. [1] and the Instruction Set Reference Intel [25] which was published in October 2015. Assumptions were made and key facts were left out in these documents. Other researchers reviewed these documents and set them in the context with filed patents Costan and Devadas [11]. In January 2016, the Intel SGX SDK was released and with it additional documentation [27] [28]. These documents give proof about the assumptions researchers made about the Intel Launch Enclave.[1]

The details about SGX are well described in these sources. The scope of this thesis is not to describe SGX into detail but to describe how the theory about SGX is applied in reality. The technology behind SGX will be described at a higher level to create understanding of SGX, its mechanics, the tools and the software model.

The use cases for SGX will be discussed based on patents that were filed by corporations. With the help of these patents it is possible to give an overview about the usage of SGX and its possible future field of application. Today it is unknown if an licensing scheme exists. Furthermore requirements for optaining production SGX certificates will be discussed.

1.2.2. SGX in Linux Kernel

In contrast to secure user mode applications against other user mode applications, this approach is using SGX to secure Linux kernel modules. Similar to a microkernel should be the kernel components isolated from each other. The goal is that even with a vulnerability in a kernel module, it is not possible to gain power over other modules by exploiting this vulnerability. With SGX it is possible to build resistant components which can not be exploited by privileged malicious applications.

However, in the official documentation of Intel SGX it is stated that the new instructions can be differentiated in privileged and unprivileged instructions. During the analysis of SGX it became clear that SGX is not built to enter enclaves from kernel mode. Instructions that are used to enter an enclave and execute enclave functions are not available in ring-0 [25]. Therefore SGX is used in a detour to the user space.

[1]https://jbeekman.nl/blog/2015/10/intel-has-full-control-over-sgx/

1.2.3. Tresor with SGX

The SGX enabled Tresor extends the Tresor by Müller et al. [46]. It is meant to be a proof of concept of isolation kernel functions. As previously described it was first planned to implement TresorSGX fully in the kernel space. Because of the findings about SGX, a three layered architecture is used to provide kernel module functionality by executing SGX in user mode.

1.3. Related Work

Intel SGX is a new technology. The required tools for executing SGX enclaves on hardware are just released in January 2016. Therefore, the published papers which discusses SGX and proposes possible use cases are limited. Papers and patents regarding use cases for enclave systems are discussed in section 2.1.8. A recommended paper which analyses the available Intel SGX documentation and Intel SGX patents is published by Costan and Devadas [11].

1.4. Results

This thesis is a first analysis of the possible usage of SGX in kernel space. During this elaboration the available SGX capable hardware and software changed, which resulted in changing requirements too.

Intel SGX has been compared to other technologies and its history has been outlined. Furthermore, a hands on experience with the SGX SDK has been described. With the help of examples its functionality and components have been described in detail. Furthermore, the current situation of SGX capable hardware and the limitations of SGX have been highlighted. Based on filed patents different use cases of SGX have been shown and discussed. In addition, a final outlook of the future of Intel SGX has been made build on these patents and the documentation.

Tresor SGX The TresorSGX architecture is a proof of concept of isolating kernel modules using Intel SGX enclaves. TresorSGX registers a cipher at the Crypto API to support encryption tools like dm-crypt. The encryption key of the TresorSGX cipher is at no time receivable by any privileged or unprivileged processes. The key is generated using sealing techniques discussed later in this thesis. This makes the cipher sturdy against cold-boot and DMA attacks. Furthermore, it is not possible to recover the key by modifying any component of the untrusted TresorSGX architecture.

The functionality of the original Tresor has been exceeded. The encryption key is not retrievable at any point. However, the additional software layers are causing a limited throughput.

1.5. Outline

Background At first the Intel Software Guard Extensions will be discussed in section 2.1. Afterwards a time line of the published SGX hardware, software and documents is presented in section 2.1.2. These SGX components were made available during the elaboration of this thesis. In section 2.1.3 an overview about other approaches on trusted computing, their benefits and disadvantages in comparison with Intel SGX is given. The security benefits of SGX are discussed in the section 2.1.4.

The techniques that are used by the Intel Software Guard extensions are described in section 2.1.5. Also the steps, tools and generated files which are used to create an enclave are discussed in that section. With the knowledge of the characteristics and technology, an overview about the software development kits (SDK) is given in section 2.1.6. The next section 2.1.7 describes the hard- and software requirements for SGX and the current state of SGX availability. The Intel SGX section is finalised with the section 2.1.8 about use cases and published patents regarding SGX and similiar technologies. In section 2.1.11 the possible future usage of SGX is discussed.

The section OpenSGX 2.2 describes the QEMU based Emulator for SGX like Enclaves. An insight into the architecture, limitations and differences to Intel SGX is given. OpenSGX has been used in the first half of the development phase of the SGX enabled Tresor but has been abandoned with the access to the Intel SGX SDK. However, OpenSGX provides nearly the same programming experience as Intel SGX without the need for SGX capable hardware. Furthermore it is possible to debug the complete emulated system, which allows a deeper understanding of the architecture.

The third part of the background chapter analyses the Linux kernel 2.3. During the analysis phase of Intel SGX different insights how SGX works and how it can be used became apparent. That lead to a higher complexity in the implementation. The differencies of the kernel- and usermode 2.3.1, the interaction and communication between these two modes will be discussed. Also the interfaces and workflow of the Linux Kernel Crypto API 2.3.3 will be analysed.

Design and Implementation Based on these findings about Intel SGX, OpenSGX and the Linux Kernel the implementation of the SGX enabled Tresor is described in section 3. After a motivation and short review on the original Tresor the benefits of using the SGX are portrayed in section 3.1.2. The design section 3.2 describes the considerations regarding enclave management, user to kernel communication and cryptography components.

The implementation of TresorSGX is described in section 3.3. After the architectural overview the enclave lifecycle is shown. The different components of the TresorSGX Linux Loadable Kernel Module are explained in 3.3.2. The possibilities to analyse and test the encryption of TresorSGX by using the Linux Crypto API testmanager are described in section 3.3.2 and by using the custom kernel module in section 3.3.2.

The user space component which communicates with the kernel module is the TresorSGX daemon and is discussed in section 3.3.3. The daemon communicates with the enclave which is explained in section 3.3.4. The usage of TresorSGX is explained in section 3.3.5. Information about the mandatory components of TresorSGX and about the setup of an encrypted partition are given there.

Evaluation The chapter 4 analyses how the initial motivation and expectations for the SGX enabled Tresor implementation matches the results. At first the usability of TresorSGX is analysed in section 4.1. This is followed by section 4.2 which discusses the correctness of the encryption and compatibility with other cryptography ciphers. In section 4.3 the performance of TresorSGX iscompared to plain usage and the standard aes encryption. The chapter ends with section 4.4 about the security properties of TresorSGX.

Conclusion and Future Work The limitations of the Software Guard Extensions are discussed in section 5.1. The observed limitations and acknowledgement during the design and implementation of TresorSGX are discussed in section 5.2. The findings about the usage of SGX to isolate OS components are finalised in section 5.3. An outlook to future work is given in section 5.4.

2 BACKGROUND

The Intel SGX capable Hardware and the Intel SGX SDK were made available just recently. A top down approach is used to describe Intel SGX in the following. This way a comprehension of the development model is created without the need to read through the complete Intel SGX Instruction Set Reference manual [25]. For deeper understanding of the mechanics important references will be given at any point. Also the paper by Costan and Devadas [11] is recommended for a more detailed description of the Intel Software Guard Extensions.

Because of the missing hard- and software (described in detail in section 2.1.7) during the first part of the research, the emulator OpenSGX was used and will be discussed in section 2.2. However, OpenSGX is only able to emulate user space applications, therefore only the user space part of TresorSGX can be executed in OpenSGX.

As previously outlined some aspects of the Linux kernel are also described in section 2.3. This section will help to gain some insight into Linux kernel development and the Linux kernel Crypto API. The possibilities of user to kernel space communication are also discussed and lead to the implementation of the Netlink interface.

2.1. Intel SGX

The Intel Software Guard Extensions is a technology which provides high level protection of data and was first published in 2013 as Innovative Instructions and Software Model for Isolated Execution [44], Using Innovative Instructions to Create Trustworthy Software Solutions [20] and Innovative Technology for CPU Based Attestation and Sealing [1].

With Intel SGX it is possible to create containers in protected memory as shown in Figure 1.1. This container is called an *enclave*. It is not possible to read or write in that protected memory from the outside of the enclave. The enclave provides integrity and availability of its data even if it is executed on a malicious host. The code of an enclave can be executed by special instructions which are available on Intel's newest generation of Skylake CPU's that were released in September 2015. The first batch of Skylake CPUs was not able to execute SGX. Only post-conversion CPUs that were made available on October 26, 2015 support SGX [29]. The special instructions are described in section 2.1.5.

In the following sections the motivation and other approaches which were created to secure the execution and the data of an application are discussed. Furthermore, the technology, architecture and software which is used by Intel SGX, is described. To put the theoretical background in relation with the reality, SGX is analysed regarding its usage. Based on filed patents and available research papers the future usage of SGX is estimated in section 2.1.8.

2.1.1. Motivation

The higher goal of the Intel Software Guard Extensions is to improve the overall information security on a computing device. This is achieved by securing and schielding the application and its memory against unauthorised access or by guaranteeing the authenticity and integrity of an enclave. The Software Guard Extensions help securing sensitive user passwords, confidential enterprise data and protecting intellectual property. Special use cases and an overview about the future use of Intel SGX based on filed patents can be found in section 2.1.8.

Furthermore, it is possible to use SGX in the cloud computing environment. Users must trust an unknown entity with their data and their applications. The user has no information how the platform provider handels the data, if the platform is secured against any insider or ousider attack or if the platform provider itself has an interest in optaining or modfying the users data.

The strategy is to distribute encrypted containers, which contain both the application and the data, to cloud computing vendors. The container is especially encrypted for that single server CPU in the foreign datacenter. A provisioning enclave can decrypt the data and initialise the enclave. Only the CPU is able to execute the enclave application and to decrypt its memory.

During the loading of an enclave its content is hashed by the CPU. This measurement hash can be used for remote-attestation (which is described in section2.1.5) against a challenger. Remote attestation can be used to verify the other communication party if it is the unmodified application one expect and if it is running in a SGX environment.

2.1.2. History

The Intel Software Guard Extensions published their whitepapers during the *The Second Workshop on Hardware and Architectural Support for Security and Privacy*[1] (HASP 2013) in Tel-Aviv in June 2013. The HASP is intented to be a workshop for security research on application level, hardware and architecture aspects in the era of cloud computing.

They at first presented their paper *Innovative Instructions and Software Model for Isolated Execution* [44][2]. Followed by the paper *Innovative Technology for CPU Based Attestation and Sealing* [1][3]. Based on this foundation the paper *Using Innovative Instructions to Create Trustworthy Software Solutions* [20][4] was published.

In September 2013 Intel published the first part of an explainatory series *SGX for Dummies*[5] describing eight SGX Design Objectives which will be discussed in section 2.1.4. Part two and three were released in the beginning of the year 2014.

In 2014 Baumann et al. [3] proposed *Haven* which allows the shielded execution of legacy applications on commodity operating systems and hardware. Haven describes a first use-case by Microsoft for the usage of SGX. Another secured architecture proposal was published by Li et al. [39] called *MiniBox*. Google is using MiniBox cross-platform concept for a Platform-as-a-Service cloud computing scenario.

The Intel Software Guard Extensions Programming Reference [25] was published in October 2014. Based on the description of the new instructions and memory operations it was possible to get a deeper understanding of SGX and how Intel wants to achieve their security goals.

In May 2015 Intel informed that SGX protects against memory attacks but not against sidechannels[6]. They are referencing a paper by Xu et al. [63] which allows the extraction of documents and images over controlled-channel attacks. Although the data itself and the memory is not accessable, these sidechannels allow a data reconstruction by observing access and timing patterns. The host application which uses the trusted enclave functions and the internal enclave functions must be hardened against these scenarios.

[1]https://sites.google.com/site/haspworkshop2013/
[2]Presentation: https://docs.google.com/file/d/0B_wHUJwViKDaRm00QlVITkYxckE/edit?usp=sharing
[3]Presentation: https://docs.google.com/file/d/0B_wHUJwViKDaaUhEUjVDcVBYUlk/edit?usp=sharing
[4]Presentation: https://docs.google.com/file/d/0B_wHUJwViKDaMm9PU3hOTUhLbHM/edit?usp=sharing
[5]https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx
[6]https://software.intel.com/en-us/blogs/2015/05/19/look-both-ways-and-watch-out-for-side-channels

At the ISCA 2015 Intel presented a set of tutorial slides [26][7] giving an in-depth architecture, SGX key hierarchy, attestation and provisioning description. They also demonstrated a side channel evaluation tool and gave an overview about the Linux SGX SDK.

The SGX capable CPU generation *Skylake* was released in August 2015[8]. However, SGX was not available at the first batch of Skylake CPU's. This information was published in a product change notification [29] on October 1st 2016, two month after the initial release. The customer should expect *post-conversion* material between October 26th and November 30th. The SGX capable CPU's do not differentiate from the non-capable CPU's. SGX can only be tested during runtime. However, testing SGX during runtime was not possible in 2015 because no SGX BIOS update was available for the desktop mainboards.

In October 2015 a researcher discovered that the *Launch enclave* is a mandatory component in the SGX lifecycle[9]. Only an Intel signed enclave can be startet without a Launch-Key / EINITTOKEN. Without this enclave no custom enclave can be initialised. That *feature* was not highlighted by Intel in the documentation and never mentioned in the SGX whitepapers. However, it is described in the issued patents by Intel. The Launch enclave guarantees that the enclave author is in a business relationship with Intel.

The Windows SGX SDK[10] containing Kernel Drivers, SGX Services and Architectural Enclaves, was published in January 2016. With the help of these tools it is possible to create and launch enclaves on supported machines.

The available SDK's permit an enclave creator to deploy debugging enclaves. For production licences a developer must undergo an evaluation process by Intel described in an article from February 2016[11]. To be granted a production license the developed application must follow defined coding practices. Found vulnerabilities must be fixed in a certain period of time. Additionally, the developers must demonstrate the ability to save the enclave signing key securely following industry best practices. These requirements will be discussed in section 2.1.7.

2.1.3. Related Approaches to Trusted Computing

Trusted Computing is a initiative by the Trusted Computing Group (TCG)[12] to enhance PC security. The TCG is a not-for-profit organisation which initially was a consortium of AMD, HP, IBM, Intel and Microsoft. Its goal is to define global industry specifications and standards which increase the security of devices.

Trusted Computing guarantees that the system behaves in predefined deterministic ways. This behavior is enforced by hardware and software. Furthermore, Trusted Computing

[7]https://software.intel.com/sites/default/files/332680-001.pdf
[8]https://newsroom.intel.com/chip-shots/chip-shot-intel-unleashes-next-gen-enthusiast-desktop-pc-platform-at-gamescom/
[9]https://jbeekman.nl/blog/2015/10/intel-has-full-control-over-sgx/
[10]https://software.intel.com/en-us/sgx-sdk
[11]https://software.intel.com/en-us/articles/intel-sgx-product-licensing
[12]https://www.trustedcomputinggroup.org/

protects critical data and defends against attacks on the system. It makes secure authentication possible and protects the cryptographic key material and certificates. Also it allows the attestation of the device itself and guarantees its identity. This allows to protect the user against some security risks, but it also hands over control to a third party.

The Electronic Frontier Foundation warns about the possible abuse of the features that come along with Trusted Computing Systems [53]. They state that Trusted Computing not only can be used to defend against malicious applications but it can also be used to defend against the system owner and can enforce policies against its will. A few examples for the negative use of Trusted Computing are application lock-in, forced upgrade, -downgrade and the forced installation of application specific spyware.

In the following some Trusted Computing approaches and individual objectives by different vendors will be discussed. This section provides background information about the past of Trusted Computing and why SGX has been developed the way it is. A complete breakdown of the different systems and a in-depth analysis can be found in [11] and will be referenced.

Trusted Platform Module

The goal of the Trusted Computing Group was to develop a specification for a standardised module which enables trusted computing features. As a result the specification *ISO/IEC 11889* was created.

Since then the TCG published updates to the original TPM specification. October 2014 the latest TPM release 2.0 was published.

The Trusted Platform Module must be able to perform asymmetric key generation, asymetric encryption / decryption, hashing and random number generation. Furthermore, it must be able to perform three different tasks [60].

- **Remote Attestation** - allows to create a hash which depends on the hard- and software configuration of the system. This hash stands as a state of the system and verifies against a third party that the state has not changed.
- **Binding** - allows the encryption of data with the public key of the TPM. The data can only be decrypted by using the secret key inside the TPM. Therefore, the data is bound to the TPM. A practical use case for binding is the provisioning of certificates and encryption keys.
- **Sealing** - allows to use binding on specific data but attaches the state of the TPM to it. Unsealing (decrypting) the data is only possible if the TPM is in the same state as during the sealing operation.

Today TPM is widely deployed as dedicated chip / module because it does not rely on CPU modifications. Therefore, the security guarantees are weaker compared to SGX. One disadvantage is that the measurement for the remote attestation includes the whole

OS kernel and drivers (which are therefore the Software Trusted Computing Base in the TPC model). With fast changing update cycles it is not possible to keep a list of all possible software configurations. Hence the remote attestation feature of TPM is not widely used.

TPM is used in Microsoft BitLocker drive encryption. The keys are saved in the TPM and only released after the TPM verifies the state of the computer. However, TPM is vulnerable to cold-boot attacks because it releases the keys to the RAM. The encryption and decryption are not executed in the TPM chip itself [19].

Intel Trusted Execution Technology

The Intel Trusted Execution Technology (TXT) is an hardware-based technology for enhancing server platform security [16]. Especially in highly visualised environments physical isolation of components is not longer possible. Intel TXT is designed to protect against threads by systems that are not in the user's control like the BIOS, hypervisor or firmware.

The Software TCB is smaller compared to the TPM model. It consists of a Virtual Machine (VM) which is hosted using CPU virtualization features. The VM is securely initialized using an authenticated code module and is protected against unauthorized direct memory access (DMA) using a protected range of memory. However TXT does not implement encryption of its DRAM so it is prone to physical attacks on the memory.

Furthermore the software running in System Management Mode (SMM) is not reset during the context switch to the enclave. Multiple times the SMM was compromised which leads to the possibility of accessing the TXT containers memory [13] [62].

ARM TrustZone

The ARM TrustZone approach to Trusted Computing is based on the concept of a trusted platform which enables infrastructure security by system design [2]. The TrustZone can execute two different systems which are called the non-secure and the secure world. This logical separation is enforced by hardware which is build into system on chip.

When compared to TPM, TrustZone is much more universal usable. By setting the non-secure bit, restrictive policies can be used to manage the access to peripherals. Instead of defining a fixed set of secure functions and protecting a single asset, it is possible to freely implement different security functions in the secure world if needed.

It is possible to implement secure boot sequences in TrustZone. When starting the system, a secure world ROM-based bootloader will initialize a flash device bootloader. The flash-based bootloader will boot the Secure World OS. It is also possible to execute two operating systems beside each other. That is achieved by starting a normal non-secure bootloader after the secure bootloader to boot the normal operating system.

The context switching between non-secure and secure world is managed by a monitor module. The functionality is similar to a traditional OS context switch, but with additional checks and policies when entering from non-secure mode. The monitor is located in the secure world, shielded from possible attacks of the normal world.

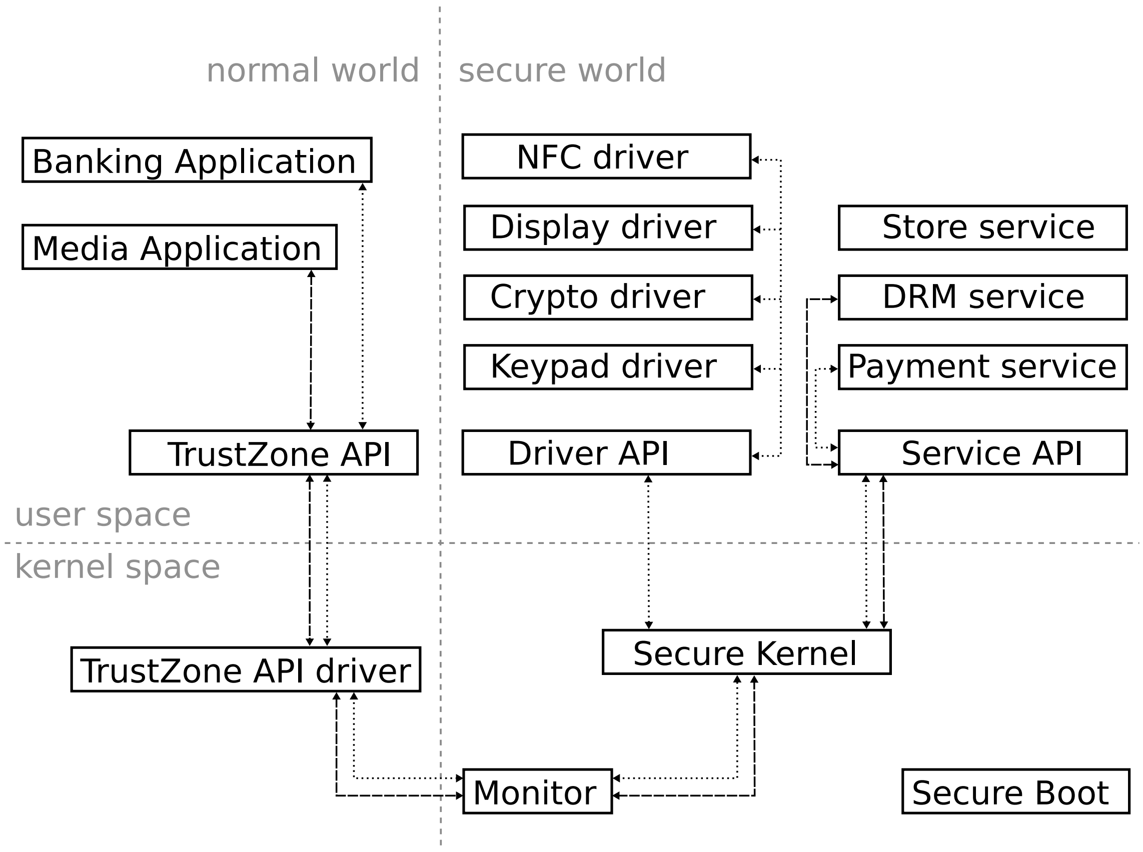

Figure 2.1.: The software architecture for the proposed DRM and Banking Use-Cases.

ARM proposes Gadget2008 as a possible product design for TrustZone. This design implements secure mechanics for the enforcement of policies for Digital-Rights-Management (DRM) and banking applications. Figure 2.1 shows the applied TrustZone architecture for the Gadget2008.

A DRM service can be placed in the secure world and is therefore secured from manipulation. The media player is placed in the normal world and can be attested by the secure world. Additional DRM data can be also saved in the secure world. When using the media player the DRM service can validate the integrity of the player by attestation. Afterwards the user's licence can be checked at a remote digital content provider service. If the user has the rights to play the media files they can be streamed or decrypted from the local storage.

To secure banking applications a trusted payment service which can access a trusted keypad and display can be implemented in the TrustZone platform. With this technology an application can display transaction details which can be signed by the user via the input of a PIN. Additional security can be achieved by using Near-Field-Communication (NFC) to read data from a physically available bank card. To allow the usage of the NFC or keypad by the normal world low level drivers must be implemented in this unsecured environment. The communication to the secure world can be protected with cryptographic methods when using normal-world device drivers.

Aegis Secure Processor

Previously described approaches enable authentication of the user and the software. However, they only protect against software based attacks against trusted components and not against physical attacks like tapping or probing chips and busses. The single-chip processor AEGIS by Suh et al. [57] uses additional mechanisms to defend applications against software and hardware based attacks.

The TCB in the AEGIS approach is smaller compared to TXT or TPM. Only the chip itself is trusted, all external modules and peripheries are not assumed to be insecure / malicious. AEGIS implements four different modes which differentiate in the allowed memory access schemes and tamper resistance.

Users can authenticate the CPU via a challenge-response mechanism. The private CPU key is created in-CPU using Physical Random Functions (PRF). Additionally the CPU can sign a hash of trusted operating system components to authenticate them.

Memory protection is enforced by access checks in the MMU and during a secure execution mode using integrity verification and encryption. AEGIS divides the memory into regions with different security guarantees and read / write modes. AEGIS leaves the control of the paging completely to the OS which can exploit the side-channel to learn about access patterns and timing schemes.

Bastion Architecture

Another approach which defends against physical and software attacks is the Bastion Architecture [8]. Based on a CPU and a hypervisor a secure scalable execution and storage platform is achieved. The Bastion architecture allows the execution of secure trusted containers inside an untrusted operating system and software stack.

First Bastion starts its secure hypervisor by using new functions in the processor. During the so called *Secure Launch* protected memory is allocated for the hypervisor and the integrity of the hypervisor is checked. The hash of the hypervisor is also used for measurement and attestation purposes.

The secure hypervisor can create virtual machines to boot an operating systems. Security-critical operating systems or modules can be started by using the Secure Launch processor feature which requests a protected execution environment from the hypervisor. An overall view on the Bastion Architecture can be seen in Figure 2.2. The integrity of the protected memory is guaranteed by cryptographic hash trees. Encryption is used to protect the confidentiality of the pages of each launched secured module. The encryption key for the memory protection is generated on each boot cycle. The saved data on the persistent disk is encrypted using symmetric cryptography. The key must be provided by the module that requests the secure storage.

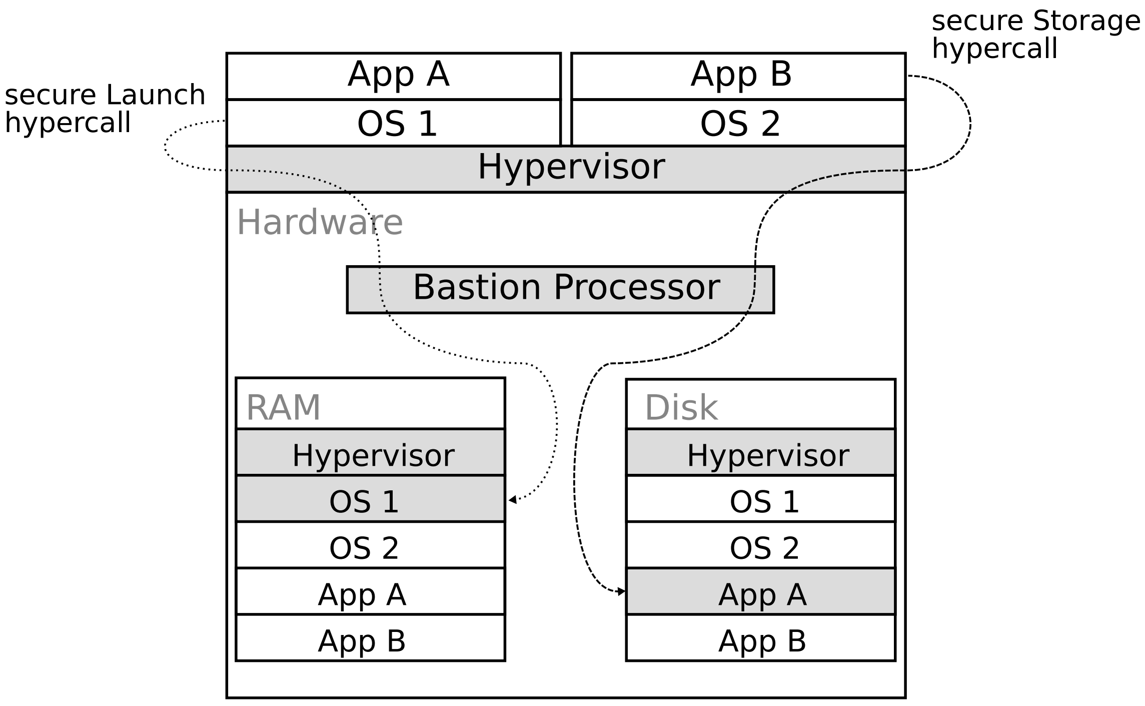

Figure 2.2.: The Bastion Architecture with trusted and untrusted components. Trusted components in grey. Secured memory can be requested by special bastion hypercalls.

With Bastion the software TCB only consists of the small hypervisor and not the complete OS (TPM) or the secured OS (TrustZone). Bastion allows the isolation of the secured modules against each other. However, Bastion is also vulnerable against side channel attacks based on memory access schemes.

Compared to Intel SGX

Different approaches have been developed in the past to achieve the goals of trusted computing which were specified by the Trusted Computing Group.

TPM as standardised and used platform is not able to achieve acceptance because the Trusted Computing Base contains all system software and does not support isolation. The system is either completely trusted or untrusted. In a world of numerous user application, drivers and OS updates that approach is not applicable.

Intel TXT enhances the security of visualisation techniques by using the TPM attestation feature to initialize virtual machines. Like TPM TXT does not protect against physical attacks on memory or bus.

The Arm TrustZone reduces the hardware TCB to the System-On-Chip. Therefore it is not prone to physical attacks on the peripheries. The main disadvantage of the TrustZone compared to SGX is that TrustZone only differentiates between the secure and non-secure world. The software TCB consists of the complete secure world, including drivers, firmware, secured applications. It is not possible to isolate and defend against threats from components in the secure world. Therefore all secure world components must be trusted.

Aegis is comparable to SGX regarding its independence to trusted system software. Aegis just uses a security kernel module whereas Intel depends on Intel-signed containers(more information about Intel Enclaves in section 2.1.5). Also contains the AEGIS processor a private key which is generated in the CPU. Furthermore, the Intel Skylake CPU includes a private key which is used for attestation and sealing mechanisms.

Bastion and Aegis provide features to allow the attestation of the secure software TCB. Intel SGX is using a hard coded key in the CPU to provide comparable attestation and measurement.

Like Bastion and Aegis, SGX is also encrypting the memory of secured modules. Intel SGX is using the *Memory Encryption Engine (MEE)* described in the paper by [17] and in the ISCA slides [26]. The MEE guarantees the confidentiality, integrity and freshness of the DRAM memory. The MEE is located in the CPU.

Intel SGX also shares the vulnerability to side-channel attacks on memory like Aegis or Bastion [63]. Intel is aware of this and informs[13] the SGX developer about possible side channels. Furthermore, they state that the developer makes sure not to leak any information via side-channels [28].

2.1.4. SGX Security Characteristics

As motivation for the following technology section the security properties of SGX will be discussed at first.

Intel also proposed methods to use SGX to deploy trustworthy software solutions [20]. This is achieved by using attestation, provisioning and sealing techniques [1]. With SGX it is possible to create an application which uses an encrypted enclave. This enclave can be measured and its integrity can be attested against a challenging service provider. Via sealing it is possible to encrypt data which can only be accessed by one enclave or all enclaves of the enclave signer.

Intel defines multiple security properties which are achieved by the Software Guard Extensions. [44] [27]

- The enclave memory is secured against observation and modification of any non-enclave party. That excludes virtual machine monitors, ring-0 applications or other

[13]https://software.intel.com/en-us/blogs/2015/05/19/look-both-ways-and-watch-out-for-side-channels

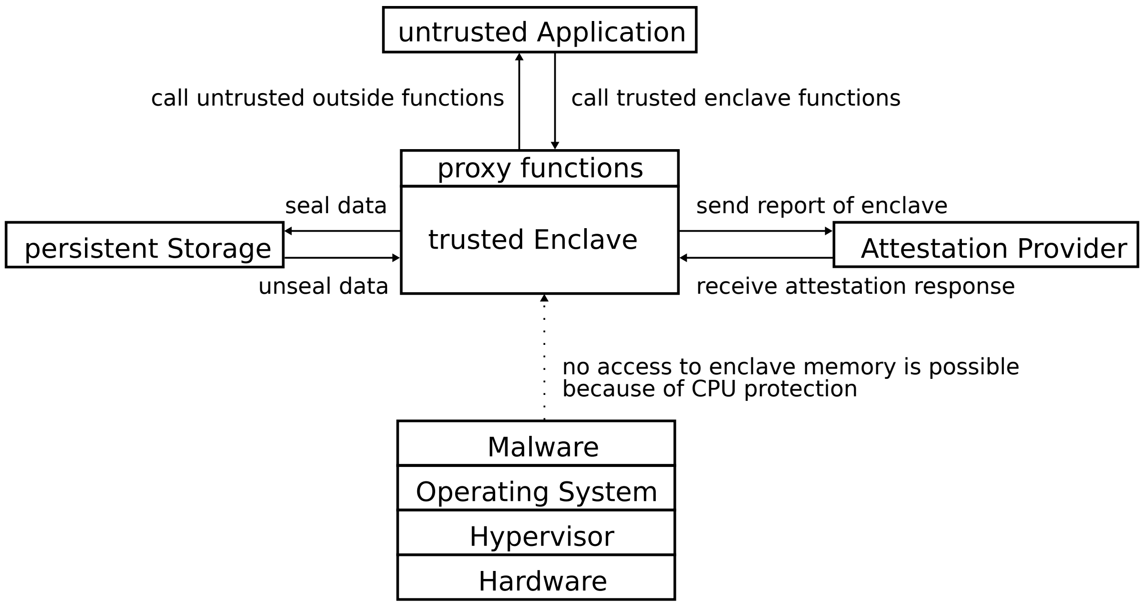

Figure 2.3.: The virtual address space layout of the enclave container in the host application.

enclaves. This is achieved by encrypting the memory with a in-CPU *Memory Encryption Engine (MME)* [17]. The encryption key changes every boot cycle.

- Via a hard-coded private key the CPU is able to perform an attestation of itself against a challenger and to sign via public-key cryptography a measurement of an enclave. That can be used to guarantee the integrity of an enclave and for enclave attestation.
- Function calls into the enclave are provided via special instruction which perform checks on the callee and the function arguments. The same applies for function calls from the enclave to the outside. Interrupts and unplanned exits will not reveal secure information because an enclave can only be stopped in a secured area.
- SGX allows the usage of multiple enclave instances which are isolated against each other and from the system software.
- Intel SGX does not protect against reverse engineering and side channel attacks. It is the duty of the enclave developer to withstand these attack vectors.
- The enclave is only debug-able with an special debugger if it is compiled with debugging enabled.

2.1.5. Technology

This section gives an overview about the Intel Software Guard Extensions. Most of the theoretical information can be found in the official documentation [27][28][30] and in papers of independent researchers [11].

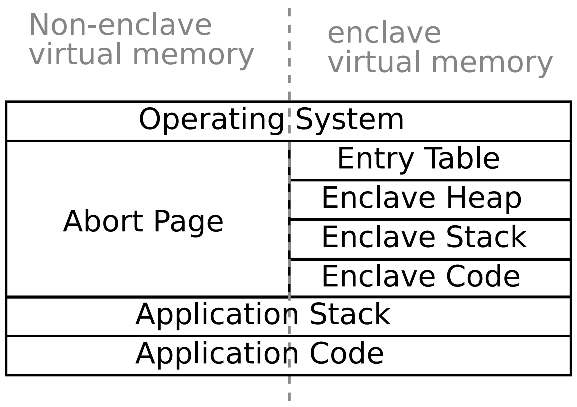

Figure 2.4.: The virtual address space layout of a non-enclave view and as seen from the enclave. Only the enclave can access its protected memory. It is also possible to call untrusted outside libraries from the enclave.

As shown in Figure 2.4 the enclave is seen as a protected container in the address space of the application. SGX guarantees the previously defined security properties to this enclave container.

In the following the architecture of SGX and how the parts interact with each other to achieve the security characteristics will be described. Furthermore, an enclave lifecycle and the building work flow characterised. This section gives an example that helps to understand the measures and actions taken in the SGX implementation of Tresor.

Architecture

The Intel Software Guard Extensions consist of multiple parts. The basis builds the Intel Skylake CPU with its extended instruction set and memory access mechanisms. These instructions are used to create, launch, enter and exit an enclave, as described in 2.1.5. The protected memory, the *Enclave Page Cache (EPC)*, for the enclave is allocated in the *Processor Reserved Memory (PRM)* and secured with a *Memory Encryption Engine* as described in section 2.1.5.

The SGX architecture is shown in Figure 2.5.

The host application is called the untrusted part. The untrusted application can call trusted functions inside the enclave. Neither the input to the enclave, nor the output of the enclave can be fully trusted because a malicious OS can modify these channels. The enclave author has to take this into consideration developing security critical applications.

To initiate the enclave a launch token is needed which can be retrieved with the help of the Intel Launch Enclave. The access to the Launch Enclave and other architectural enclaves (Quoting, Provisioning, etc) is provided by the AESM service in user space. SGX libraries provide the necessary methods to communicate with the AESM Service.

Enclaves can only be entered in user space. However, creating and initiating an enclave

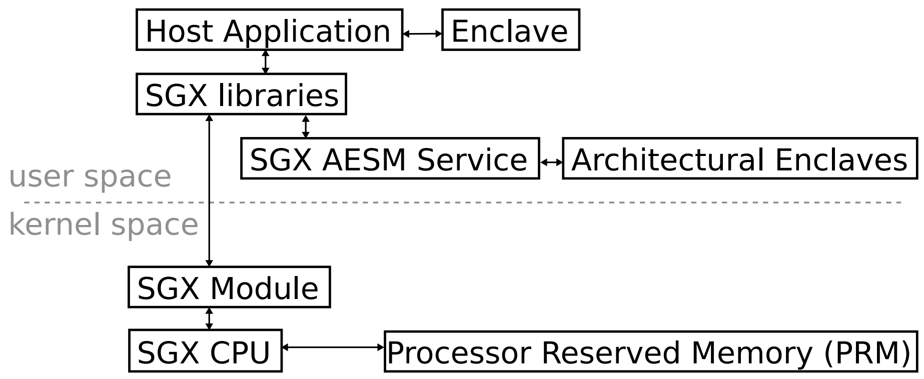

Figure 2.5.: High level hardware and software architecture of SGX.

is only possible in kernel space. Therefore, a privileged SGX module or driver must be installed in kernel space to manage the enclave page cache and calling the specific SGX instructions. The detailed lifecycle of an enclave is described in Section 2.1.5.

The launched enclave can only be entered from an unprivileged user-mode application via special SGX instructions. When the enclave is running, any application which mapped the enclaves page cache into its virtual address space can call enclave functions. When entering the enclave the CPU is switched into *enclave mode* which still runs at user-mode with ring-3 privileges.

Memory Usage

The essential security feature of SGX is that the enclaves data and code is stored, protected and isolated in the *Processor Reserved Memory(PRM)*. The PRM is a range in the DRAM which is protected by the *Memory Encryption Engine (MME)*[17]. This engine rejects *Direct Memory Access (DMA)* on the PRM. The datastructure in the PRM consists of the *Enclave Page Cache Map (EPCM)* and the *Enclave Page Cache (EPC)* itself. The EPC is split into 4 kilo byte pages which are managed in the EPCM and can be assigned to different enclaves. The layout of the physical address space can be seen in Figure 2.6.

The *SGX Enclave Control Structure (SECS)* is required for each enclave and is saved in the EPC. The SECS represents exactly one enclave and contains enclave information e.g. Enclave ID, Enclave HASH, Enclave Size.

In the EPCM access control and security information for these pages are saved. With these information it is evaluated whether to allow the access to the page or not. The EPCM is managed by the CPU and not accessible by system software. Each EPCM entry contains the following information: the validity of the EPC page, the enclave instance that owns the page (links the the enclave SECS page) and the type of the page (regular Data, Thread Control Structure, Version Array, SECS).

When accessing memory numerous checks are executed. A control flow for in-enclave memory access can be seen in Figure 2.7. External access to PRM is blocked with a

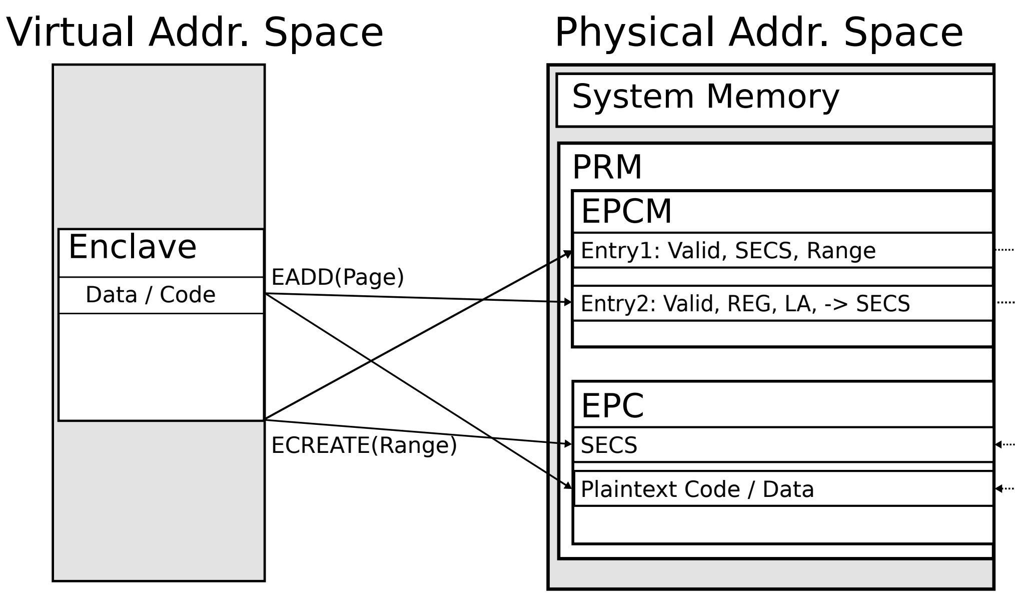

Figure 2.6.: Memory mechanisms during ECREATE and EADD.

reference to non existing memory. The following conditions have to be met to allow the memory access. The logical processor must be in *enclave mode*. The page must belong to current executed enclave. Accessed pages need to have a correct virtual address. If the address is not in the enclaves virtual address space and not inside the PRM the access is allowed.

If the CPU is in enclave mode, the address of the page is not in that enclaves memory range but the address is in the EPC, the access is treated like non-enclave to EPC access. In that case nonexisting memory will be referenced. If the CPU is in enclave mode, the address is in the enclave range but the adress is not in the EPC or EPCM checks fail, a page fault will be signaled.

The system software, e.g. the OS or an hypervisor, manages the EPC like normal memory. The os is able to swap out unused pages from the PRM with SGX instructions. The pages on non-PRM DRAM can be swapped with standard mechanisms. Furthermore, the SGX programming model expects the system software to include enclave management functionality which allows the usage of user-mode SGX applications without the need for privileged drivers.

The confidentiality, integrity and freshness of the DRAM data is guaranteed by the *Memory Encryption Engine (MME)*[17]. A normal CPU reads and writes data from its internal cache. If the CPU needs data that is not loaded in the cache, the Memory Controller loads the data from DRAM. The MME extends the Memory Controller and steps in its place when data from a protected region (in SGX the PRM) is accessed. The MME encrypts data before writing, and decrypts data after reading. Additionally, it verifies the data with the help of an integrity tree. The keys used for encryption and authentication are generated at boot time.

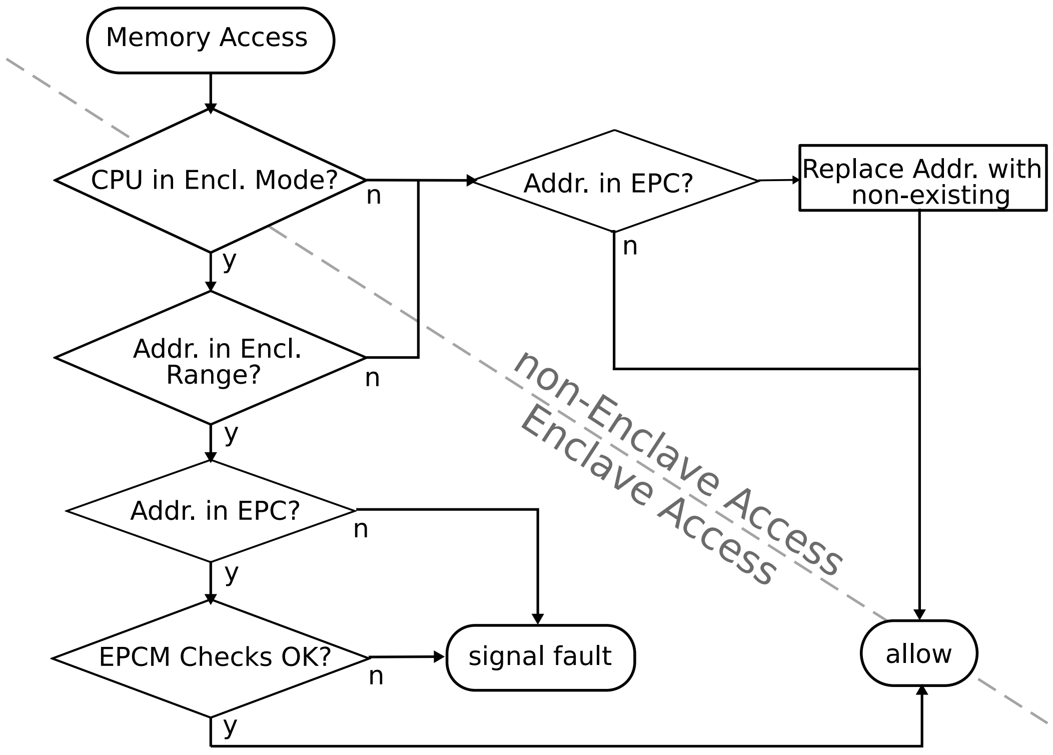

Figure 2.7.: SGX memory access control scheme.

Enclave Lifecycle

The new SGX instructions can be separated in supervisor and user mode instructions. A detailed specification of the instructions and the required parameters can be found in the SGX Programming Reference [25]. For further clarification the paper by [11] is recommended.

In the following, an overview of the CPU leaf functions is given. These instructions will be discussed in the next sections. An excerpt of instructions is shown in Table 2.1. To execute the instructions explicit data is needed to perform the security checks and memory allocations.

To create and initialize the enclave, the host application hands over the enclave content to a privileged service running in supervisor / ring-0 mode. The Intel SGX SDK provides a SGX kernel module / driver for that purpose. Furthermore, a Launch Enclave is required, because EGETKEY is only callable from an enclave with launch capabilities [25]. In the following a lifecycle of an enclave is described and illustrated in Figure 2.8. The dashed Launch Enclave function calls are simplified for better clarity.

Mode	Instruction	Description	Explicit Data Structures
Supervisor	ECREATE	Create an enclave	PAGEINFO, EPCPAGE
Supervisor	EADD	Adds a page to the EPC	PAGEINFO, EPCPAGE
Supervisor	EEXTEND	EPC page measurement	EPCPAGE
Supervisor	EINIT	Initialize an enclave	SIGSTRUCT, SECS, EINITTOKEN
Supervisor	EREMOVE	Removes a page from EPC	EPCPAGE
User mode	EENTER	Enter an enclave	TCS, SSA
User mode	EEXIT	Exit an enclave	
User mode	ERESUME	Re-enter an enclave	TCS, SSA
User mode	EGETKEY	Create cryptographic key	KEYREQUEST, KEY
User mode	EREPORT	Create cryptographic report	TARGETINFO, REPORT-DATA

Table 2.1.: Subset of SGX Enclave Instructions

Creating the enclave An enclave resides in the execution context of a normal host application. These host application must use a privileged party to create the enclave. ECREATE is the first enclave build instruction by that privileged party. It creates a *SGX Enclave Control Structure (SECS)* in the EPC and marks this page it as valid. The SECS->INIT Attribute is set to false because the enclave is not initialised yet. Therefore the enclave can not be executed until the enclave is initialised.

The explicit attribute PAGEINFO of the ECREATE instruction contains information about the enclave (e.g. base address, range). The PAGEINFO will be moved to the page referenced by the SGX Enclave Control Structure (SECS).

Adding enclave pages In a second step EADD is used to add more EPC pages to the enclave. EADD can initialize *Thread Control Structure (TCS)* pages or regular code / data pages. The EPCM is updated with the new page entry and a cryptographic log is saved into SECS. Afterwards EEXTEND is executed which measures the added page and adds the result to the cryptographic log in SECS. EEXTEND is also used in software-attestation scenarios.

Initialising the enclave The third step is to call EINIT with SIGSTRUCT, SECS and an EINITTOKEN. EINIT finalizes the initialization process and finishes the cryptographic log in SECS. Therefore, no page can be added past EINIT. A hash of the cryptographic log is generated and compared to the hash provided in the SIGSTRUCT. That allows the attestation of the enclave.

The EINITTOKEN is generated by EGETKEY which can only be called from a Launch Enclave. The policy is to allow the execution of the measured enclave (defined via SIGSTRUCT) only on that CPU where the EINITTOKEN was generated. That means it is not possible to execute an enclave without the prior execution of the Launch Enclave. The Intel Launch Enclave is included in the SGX SDK described in Section 2.1.6.

EINIT is successful if SIGSTRUCT is correctly signed with the public key which is included in SIGSTRUCT, the measurement of the enclave matches the measurement in SIGSTRUCT and the enclaves attributes matches the defined in SIGSTRUCT. Then the sealing identity and enclave identity is saved in the SECS. Finally the SECS->INIT Attribute is set to true and the enclave can be entered.

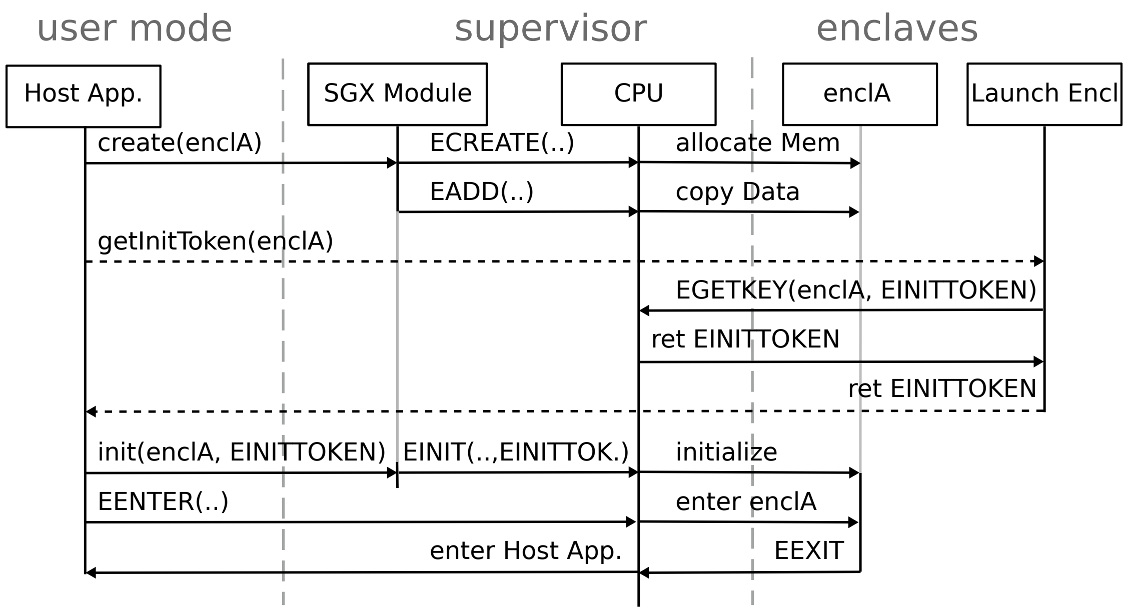

Figure 2.8.: The lifecycle of an enclave. Dashed calls are simplified for better clarity.

Entering the enclave The enclave is initialised and can be entered by a user mode application. EENTER uses a pointer from the *Thread Control Structure (TCS)* to an address inside the enclave to transfer control. The CPU switches into enclave mode, saves RSP, RBP for later restore and modifies architectural features registers with enclave values. When entering the enclave the TCS is marked as busy, preventing any other logical CPU to enter the enclave simultaneously.

During the enclave mode an exception, interrupt or VM exit may occur. Instead of transferring the control directly, the enclave state is saved and a synthetic processor state is loaded to prevent data leakage. Then a *Asynchronous Enclave Exit (AEX)* is called, the *Asynchronous Exit Pointer (AEP)* is pushed to IRET and the processor leaves the enclave mode.

When calling ERESUME the AEP will be retrieved from IRET, the enclave registers will be restored and the CPU switches to enclave mode. An exception handler should be implemented in the enclave to prevent ERESUME - AEX loops which are caused by exceptions triggered by the enclave. The host application could call ERESUME with a different AEP to resolve the exception.

Secure data in the CPU registers must be erased during enclave mode because EEXIT does not modify the most of the registers. Furthermore, should the saved RSP and RBP restored to prevent faults.

Destroying the enclave An enclave can only be destroyed with the supervisor EREMOVE instruction. It deallocates the EPC pages by marking the EPCM entry as invalid if no logical processor is executing the page-owning enclave. After all EPC pages have been set to invalid the SECS page can be removed.

Attestation

Attestation is a core feature of the trusted computing model. The attestation guarantees the integrity and authenticity of the trusted components by signing an enclave measurement. The measurement of an enclave is the same every time the enclave is initialised. In Intel SGX the EGETKEY and EREPORT instruction are used for attestation [1]. Both instructions can only be executed from inside of an enclave.

To perform an attestation an enclave must be measured and signed. Each enclave contains two signing identities, MRENCLAVE and MRSIGNER.

MRENCLAVE is the identity of the enclave. It is created by using a hash of the internal build log, which was written during ECREATE, EADD, EEXTEND and finalized using EINIT. Therefore MRENCLAVE contains information about the content of the pages, the position of the pages and the security flags. The MRENCLAVE will be the same for every build cycle, if the enclave is not modified.

MRSIGNER is also called the *Sealing Identity*, because it can be used to seal data (see section 2.1.5). The public Key is retrieved of the *Enclave Signature Structure (SIGSTRUCT)*, which consists of information about the Enclave, the value of the expected MRENCLAVE and a public Key of the *Sealing Authority*. Furthermore, is SIGSTRUCT signed with the Sealing Authority's public key, which can be verified. When the value of MRENCLAVE in SIGSTRUCT and the calculated MRENCLAVE are equal the hash of the public Key of the Sealing Authority is saved in MRSIGNER. MRSIGNER is the same for all enclaves, build by the same Sealing Authority. This for example allows updating of the enclave content while using the same encryption key for sealing operations.

Intra-Platform Attestation By using MRENCLAVE and MRSIGNER it is possible to certify the identity of two enclaves against each other to guarantee the integrity and authenticity of the enclaves before negotiating a secret channel.

An Intra-Platform attestation workflow is shown in Figure 2.9. The sequence diagram is minimised to the enclaves only, in reality the communication will be managed by a host application.

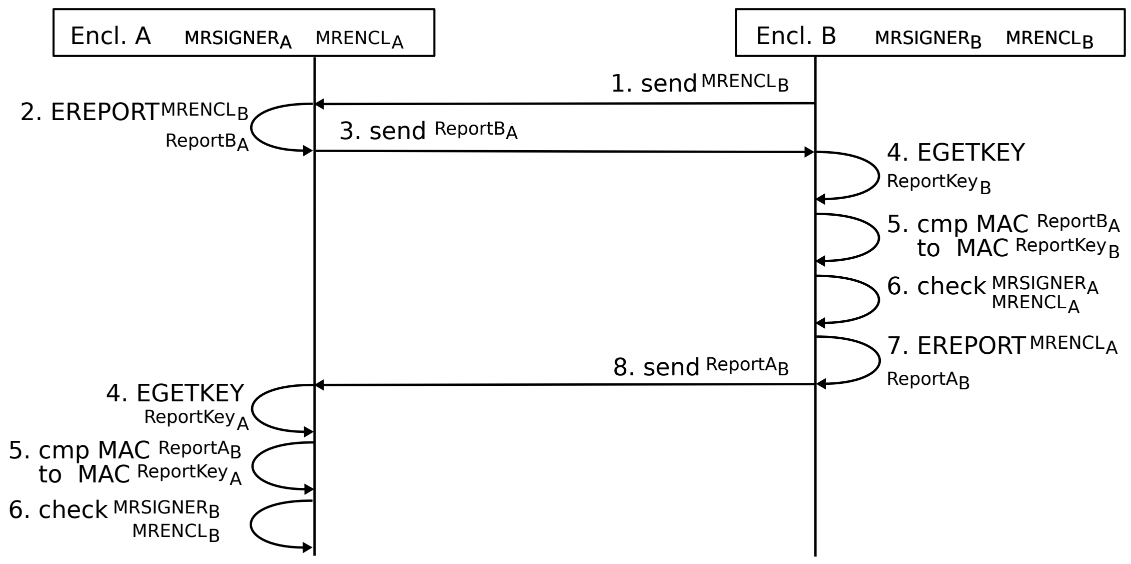
Encl. A MRSIGNER_A MRENCL_A
Encl. B MRSIGNER_B MRENCL_B
1. send MRENCL_B
2. EREPORT MRENCL_B ReportB_A
3. send ReportB_A
4. EGETKEY ReportKey_B
5. cmp MAC ReportB_A to MAC ReportKey_B
6. check MRSIGNER_A MRENCL_A
7. EREPORT MRENCL_A ReportA_B
8. send ReportA_B
4. EGETKEY ReportKey_A
5. cmp MAC ReportA_B to MAC ReportKey_A
6. check MRSIGNER_B MRENCL_B

Figure 2.9.: Intra Plattform Attestation between two enclaves, communication is simplified.

First the EnclaveB sends its identity (MRENCLAVE$_B$) to EnclaveA. EnclaveA calls EGETREPORT with MRENCLAVE$_B$ to generate ReportB$_A$, a report of EnclaveB which is signed by EnclaveA. The essence of the ReportB$_A$ is the signed MAC of the target enclave B, a secret Intel SGX master key and and the SECS attributes of Enclave A. Detailed information about the data flow can be found in [11].

In the third step this is report send to EnclaveB. EnclaveB retrieves its ReportKey to compare its MAC with the MAC in the Report. The MAC guarantees that both Enclaves are running on the same CPU (same secret Intel SGX master key in the CPU) in enclave mode.

In step six the MRENCLAVEA is evaluated with MRSIGNERA. That proofs the authenticity and integrity of EnclaveA. EnclaveB has now sucessfull been attestated EnclaveA. EnclaveB can now send its ReportAB to EnclaveA. As a result both Enclaves performed the attestation of each other.

Remote Attestation Intel proposes a model [1] for remote attestation. Additional Enclaves are required because instead of attesting 2 enclaves at one system via symmetric cryptography, asymmetric cryptography is used for remote attestation. A *Quoting Enclave* is used which verifies an enclaves report. It replaces the MAC of the report with a signature over the report. The signature is generated with a CPU specific private key. The report, also called *Quote*, can send to a remote party for verification.

Intel uses an *Intel Enhanced Privacy ID(EPID)* [7] as anonymous attestation scheme. The EPID bases on a fusekey which is build into the CPU during manufacturing. A database of these fusekeys is maintained by Intel. With the help of a *Provisioning Enclave* an attestation key can be retrieved by Intel's provisioning service [26]. The EPID can be revoked if the private key of the CPU is considered insecure via the back-end infrastructure.

Costan and Devadas [11] analysed the Remote Attestation feature in great detail. Remote attestation is not in the scope of this thesis and therefore will not be analysed any further.

Provisioning / Sealing

During enclave execution the data of the enclave is secured with measures described in 2.1.5. When exiting the enclave the pages will be cleared and no enclave state is saved. As described in section 2.1.3 a trusted computing feature is the sealing of data. The integrity and confidentiality of this data must be guaranteed.

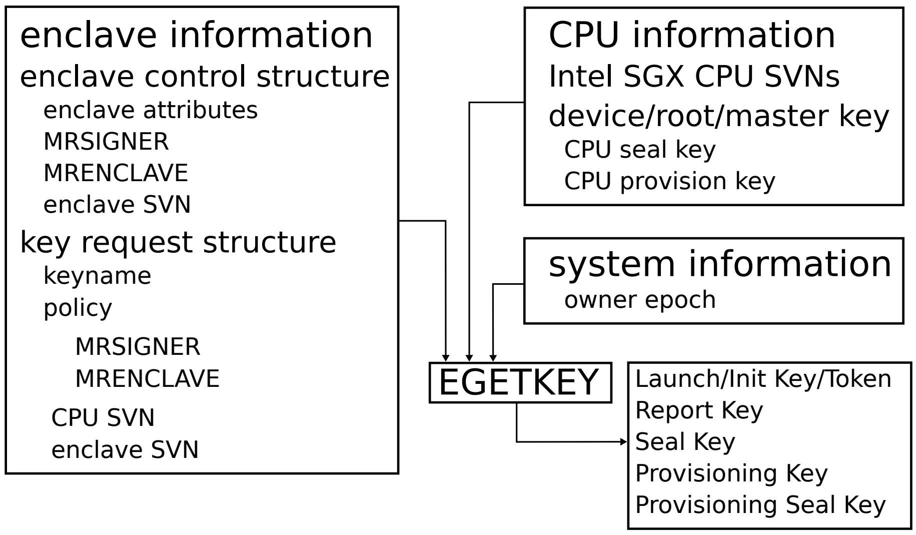

Figure 2.10.: Key hierarchy and generation as described in the Intel SGX manual [25].

Figure 2.10 shows the components which influence the SGX EGETKEY function. It is only possible to generate the same key again if all of the required parameters are the same as in a prior key generation. The key generation process and its consequences for sealed data will be described in the following.

The Intel Skylake Processor contains two different keys for provisioning and sealing. The provisioning key is known to Intel and can be used to encrypt secrets. These secrets which can then be send to the enclave and encrypted during its execution. By using a sealing key, only known to the the enclave itself, data can be sealed and unsealed [1]. With EGETKEY the seal and provision key can be retrieved. Intel provides different policies which are bound to the keys usage. These policies are based on MRENCLAVE and MRSIGNER, discussed in 2.1.5.

If EGETKEY is called with the policy MRENCLAVE, it will seal the data specific to the enclave identity. Any change in the enclave will lead to a different seal key. Only this specific enclave instance is able to generate the same seal key again. Therefore, data can be encrypted and decrypted only by this enclave on this single CPU.

MRSIGNER will allow the sealing to the Sealing Identity, the author of the enclave. The generated seal key can be used by multiple enclaves of that enclave signer to access the same data. An applications product ID and a *Security Version Number (SVN)* are included in the sealing identity. The SVN can be used for invalidate old sealed data if a security flaw is patched in a newer enclave version. This sealing model is usefull to allow the usage of data by multiple enclaves.

Personal entropy can be added to the key derivation via the OwnerEpoch value. If the value is changed, the previously generated seal key cannot be generated with the same seal policies again. Therefore, a modification of OwnerEpoch can used to make sealed data on the system inaccessable. OwnerEpoch is loaded into the MSR when SGX is booted [25].

The official documentation of SGX [25] describes the key hierarchy briefly. A in-CPU key is used as unique root key of the key hierarchy. The keys which are returned by the EGETKEY instructions derive from that key. Costan and Devadas [11] describe the root key as SGX Master Derivation Key, which is used in all key derivation processes. That leads to the assumption that a sealed secret can not be encrypted by the same enclave on another CPU because the root key differs.

2.1.6. SGX SDK

The Intel SGX SDK provides tools to build and execute enclaves. As described in 2.1.5 an Intel Launch Enclave is needed to generate the launch key for a custom enclave. The Launch Enclave is an Intel Architectural Enclave. The following enclaves are included in the SDK.

- Launch Enclave - measures custom enclave and provides launch key / EINIT token for it. Used for licensing purposes in the future.
- Quoting Enclave - signs a report of an enclave with the platform specific Intel EPID key which can only be accessed by the quoting enclave. Used during remote attestation.
- Provisioning Enclave - Proofs its identity to the Intel Provisioning Service to receive attestation key. That key is send to the Quoting Enclave for remote attestation.
- Platform Service Enclaves - are used for pairing and protected session management. Also used for remote attestation, only briefly described in [27]

The architectural enclaves can only be entered from the user mode. Therefore, the SDK includes the AESM service / daemon which manages the enclaves. In addition, the AESM service includes a database for the Platform Service Enclave. This database contains tables with information about nodes, MRSIGNER and other data. This includes the service a public key for provisioning and an white list certificate. No further explanations regarding these files could be found.

The communication between the custom host application, which enters the enclave and the AESM service is established by the SGX libraries. A subset of the libraries that are included in the SDK.

- sgx_trts - SGX internals
- sgx_tstdc - standard C library
- sgx_tservice - sealing, architectural enclave support
- sgx_tcrypto - cryptographic library
- sgx_urts - enclave management library, creation and entering from host application

To provide SGX functionality in the kernel space the SDK also contains a kernel module / driver. That module manages the enclave page cache (EPC) management.

Enclave Building

Intel published the Intel SGX Evaluation SDK tools to build the encrypted enclave containers and the matching application. The SDK consists of multiple tools which allows the building, measuring, debugging and configuration of enclaves.

- Edger8r Tool - analyses the enclave's EDL file and generates interfaces and proxies between the trusted enclave and untrusted application
- Enclave Signing Tool - generates metadata such as the enclaves signature and adds the metadata to the enclave
- Debugger - supports the analysis of enclaves with active debug flag
- Memory Measurement Tool - analyses the memory usage of the enclave
- CPUSVN Configuration Tool - used to configure the security version of the CPU which affects its key derivation process

Building Workflow Enclave Generation is the building of the enclave container and the application with the SGX SDK. The application is the untrusted non-encrypted part which starts the enclave. Enclave creation is the process in the application itself which launches the enclave.

To generate a running enclave with the Intel SGX SDK following files are required.

- Intel SGX SDK include files - especially the sgx_urts which is used by the application to create and launch the enclave
- Intel SGX SDK library files - especially the sgx_edger8r and sgx_sign
- Intel isgx Kernel Module - for executing the new SGX instructions and management of the EPC

- Application source file
- Enclave source and EDL file

In Figure 2.11 the building workflow is visualised. The enclave EDL file is parsed by the Edger8r which generates the Enclave Interface files. The generated interface definitions and implementations are used by the enclave and the host application for communication. The SGX Libraries are required during the linking. The generated library is called an unsigned library. By using the SGX signer a certificate of the enclave and its configuration is created. The generated enclave library is signed and can be used in an host application.

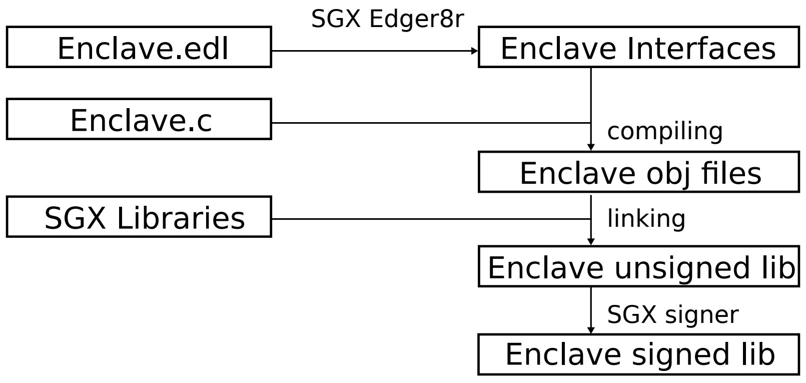

Figure 2.11.: The building workflow with the Intel SGX SDK.

In the following, an example demonstrate the building of an Enclave and its application. The enclave offers a trusted function which changes a *char array*.

EDL file describes the interfaces between the enclave and the untrusted application. It is read by the edger8r tool which generates the edge routines for the interfaces.

The EDL file is divided in an trusted (enclave) and untrusted (application) part. The return value, function name and parameters are user defined. If another return value than void is defined, the value can be retrieved in the function call as 2nd parameter. An example EDL file is shown in Listing 2.12.

The following parameter definitions are available:

```
enclave {
        trusted { // secured enclave calls
                public void ecall_changeBuf(
                        [in,out, size=len] char* buf,
                        size_t len);
        };
        untrusted { // insecure outside calls
        };
};
```

Figure 2.12.: Enclave.edl interface definitions

- user_check - the pointer wont be verified by SGX. Neither is the content of the buffer copied into the enclave. The user is in charge of the check and memory operations.
- in - During ECALL, the buffer and content will be located inside the enclave. During OCALL, the buffer will be copied from the enclave to the application.
- out - During ECALL, buffer will be allocated and can be used by enclave. On the function return the buffer will be copied to the outside application. During OCALL, the untrusted buffer from the application will be checked and copied into the enclave.
- in, out - Combines [in] and [out] functionality. Like [in] the buffer and content will be located in the enclave. But on the function return the buffer will be copied back to the source location. It works the same for ECALL and OCALL.
- isaray - only the pointer is copied into the enclave

Edger8r tool analyses the given enclave EDL file and generates the routines for the outside and enclave calls. The generated files end with _t and _u, declaring trusted and untrusted proxies and bridges. The trusted part is used by the enclave, the untrusted by the application. The generated proxy files can be found in Appendix A.1.

As shown in listing A.2 the edger8r expands the EDL file to the proxys which are used during compilation of the untrusted application and trusted enclave.

In contrast to a simple untrusted proxy definition, multiple checks are performed when calling a trusted function. The checks analyse if ECALL parameters point to untrusted memory and OCALL parameters point to trusted memory. If these checks fail, the corresponding CALL will not be executed and an error will be returned. If these checks were ignored the CPU itself would handle these invalid calls with faults or pointers to non valid addresses. During an ECALL the checks CHECK_REF_POINTER and CHECK_UNIQUE_POINTER ensure that the structures do not overlap the enclave memory. In this example the buffer is passed as [in] attribute, so the trusted bridge allocates memory in the enclave and copies the memory from the outside pointer to the inside. These functionality is explained in detail in the Intel SGX SDK Users Guide [27].

Intel gives the notice that it is crucial to the integrity of the enclave that the Edger8r tool is run in a malware free environment. The correctness of the checks guarantee a secure usage of the enclave functions, therefore they should be reviewed by the enclave developer.

Enclave trusted function and main application The Enclave.c (Listing 2.13) only contains the trusted enclave function. In the `ecall_changeBuf` function is buffer referenced. A `secret` is copied to the address of `buf` if the `len` is greater then the length of the string in `secret`. This is a dummy function without the claim to be secure. Hardcoded secrets must not be included in real enclave applications because the enclave

```
#include "enclave.h"
#include "tlibc/string.h"
#include "tlibc/stdio.h"

void ecall_changeBuf(char *buf, int len)
{
        const char *secret = "Hello␣from␣Enclave!";
        if (len > strlen(secret))
        {
                memcpy(buf, secret, strlen(secret) + 1);
        }
}
```

Figure 2.13.: Enclave.c trusted function which copies a in-enclave saved string to the buffer.

```
#include <../include/sgx_urts.h> // sgx structs func
#include "Enclave/enclave_u.h" // enclave functions
...
        ret = sgx_create_enclave(
                "./Enclave/enclave.so",
                DEBUG_ENCLAVE,
                &token, &updated, &eid, NULL );

        // Change the buffer in the enclave
        char buffer[100] = "Hello␣World!";
        printf("App:␣Buffer␣before␣change:␣%s\n", buffer);
        ret = ecall_changeBuf(eid, buffer, 100);
...
```

Figure 2.14.: App.c untrusted code

library file is not encrypted. It is possible to optain these secrets by reverse engineering. Secrets must provided the way it is described in Intel's secure provisioning schemes.

The App.c (Listing 2.14) uses the generated _u header files and creates the SGX enclave using the `sgx_urts` library. It is not necessary for the host application to be aware of the source code of the enclave. It just refers to the untrusted proxy files. To create the enclave the `sgx_create_enclave` function of the SGX library is called with the enclave file path as parameter beside others. The enclave file is created by the *Signing Tool*. When calling the trusted enclave function the first parameter must be the enclave ID which was returned during the enclave creation. The return value of the trusted function is a SGX error code.

Signing Tool The signing tool guarantees the enclave integrity by creating a signature of the properties and the enclave measurement. Intel states that modifications on the enclave code, its properties or the signature can be detected that way. The signing tool also checks for errors and security related problems.

The signing tool uses a RSA 3072-bit key pair to sign the raw signature properties. A requirement of creating enclaves in release mode is to sign them with white-listed enclave signing keys. These keys must be stored in a hardware security module. Today it is unknown how one can obtain such white-listed signing key from Intel and what licensing model is applied to them. A white list certificate is included in the Intel SGX SDK AESM Service.

The signing tool also analyses the linked libraries and used functions for possible security vulnerabilities and informs the user about them.

2.1.7. SGX Availability

As previously summarised in the SGX history section 2.1.2 Software Guard Extensions capable Intel Skylake Processors[14] are available since August 2015.

The Product Change Notification [29], published on October 1, 2015 declared that the available CPUs are not able to execute SGX. A minor change in the manufacturing configuration allows the usage of SGX on so called post-conversion CPUs distributed past October 26, 2015. The post-conversion CPUs can be identified by the S-Spec number (printed on CPU) or the material Master Number (MM#). If the CPU is already installed SGX support can only be tested via executing a SGX Enclave application. There is no possibility to retrieve information about the conversion type.

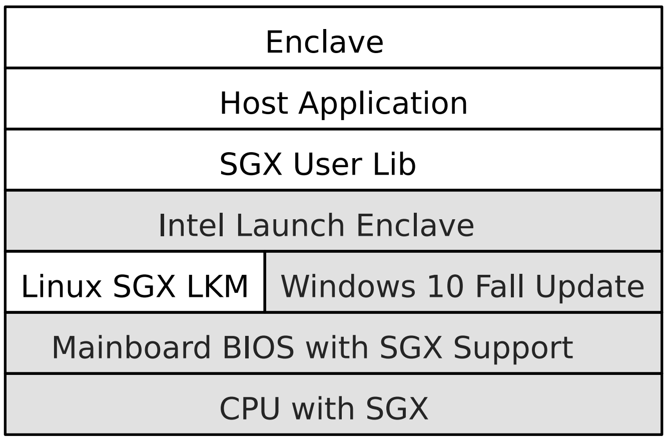

Figure 2.15.: The components needed for executing an enclave. Gray components must be delivered by a third party and must be trusted without the possibility for reviewing.

Before using SGX it must be enabled via the SGX_ENABLE field IA32_FEATURE_CONTROL MSR [25]. SGX is defined as opt-in, so MSR.IA32_FEATURE_CONTROL.SGX_ENABLE is default 0. However the modification of the IA32_FEATURE_CONTROL is prevented by its first lock bit, if it is set to 1. In the manual [18] the lock bit is described as:

[14]http://ark.intel.com/products/codename/37572/Skylake

> When set, locks this MSR from being written, writes to this bit will result in GP(0). Therefore the lock bit must be set after configuring support for Intel Virtualization Technology and prior to transferring control to an option ROM or the OS.

Therefore, SGX can only be used if it is supported by the BIOS where it must be activated. If SGX enable is set, the BIOS will reserve the SGX related memory. That memory is the the processor reserved memory (PRM) [31].

Because of the opt-in functionality of SGX it must be activated in the BIOS. Hardware vendors showing different interest in enabling SGX support in their BIOS. Today, 9 month after the initial launch of SGX, only a few Notebook BIOS Drivers and no Mainboard BIOS Driver with SGX support is available[15]. Dell began shipping its newer devices with BIOS SGX support in the second quarter of 2016.

However, no desktop mainboard BIOS with SGX support is available at this time. The open source BIOS *libreboot* does not support newer Intel CPUs because of security and freedom issues[16]. Another free BIOS alternative *coreboot* is unable to run on devices with activated Intel Boot Guard [24] which secures the UEFI firmware to protect against malicious modifications[17].

Because of the new instructions and the complex lifecycle of an enclave (section 2.1.5) it is not possible to execute SGX without further software components. An Intel signed Launch Enclave is needed to create a EINITTOKEN. Only an Intel-signed enclave, can be initiated without the EINITTOKEN because of public-key crypto which uses a in-CPU saved key to verify the authenticity of the enclave [25].

In Figure 2.15 the required components for enclave execution are shown. At the base is the CPU with SGX support. The SGX support must be enabled by using the SGX_ENABLE bit in the MSR. That is only possible in the BIOS because the MSR will be locked then.

In Linux it is possible to patch the kernel with SGX functionality[18]. The Linux SGX SDK includes the SGX kernel module in text format. On Windows the Windows 10 Fall Update must be installed for managing enclaves. The Intel Architectural Enclaves are the only ones who can be executed without a prior Launch Enclave. Any other enclave depends on the proprietary Intel Launch Enclave. The Architectural Enclaves are managed by a Daemon / Service running in user-mode. Intel includes this service in its SDK. For executing own enclaves a similar service can be developed[19].

[15]incomplete list of hardware with SGX support: https://github.com/ayeks/SGX-hardware
[16]https://libreboot.org/faq/#intel
[17]https://blogs.intel.com/evangelists/2015/02/20/tricky-world-securing-firmware/
[18]open linux sgx kernel module: https://github.com/jethrogb/sgx-utils/tree/master/linux-driver
[19]open source SGX user-mode service: https://github.com/jethrogb/sgx-utils/tree/master/sgxs-tools

2.1.8. Use Cases

In its initial whitepapers Intel demonstrated use-cases for the Software Guard Extensions [1] [20]. In this chapter the published use-cases for SGX will be characterised and assigned into groups. To our knowledge no approach was made of using of SGX in kernel as described in chapter 3.

The use-cases of SGX can be differentiated in shielded execution, policy based configuration, deployment / provisioning, identification / attestation and Digital Rights Management (DRM). Some categories are not clear separable from others, this will be discussed in the following

The patent for the Intel SGX technology was first filed on Dec. 17, 2010 as *Technique for supporting multiple secure enclaves*[34]. That patent describes a system which consists of multiple processors and platform keys to secure the execution of enclaves. Another patent by Intel filed on Jun. 19, 2012 describes the provisioning and sealing process [43]. Both patents also describe methods to enforce a business model which will discussed in Section 2.1.11.

Shielded execution

In 2014, a year prior the public release of SGX, Microsoft proposed a prototype, *Haven* [3], which uses the SGX hardware protection mechanisms. Haven allows the execution of unmodified legacy applications like SQL Server and Apache. This defends against a malicious host in cloud-computing scenarios. Their objective is to execute applications on a cloud platform with the same trust level of a own datacenter. The legacy applications are not aware of SGX, they can allocate memory, throw exceptions and execute instructions which are forbidden in an enclave. Haven is using an in-enclave OS, based on their Drawbridge ABI [50], to implement the Windows 8 API for the legacy application in the enclave. Additional remote-attestation can be used to verify the remote enclave at a cloud computing vendor.

SGX falls short when the syscalls to the host OS are manipulated to defeat the enclave security mechanisms. Sophisticated manipulation can be used to alter OpenSSL certificate checks to establish an untrusted connection. These attacks must be crafted and can be used only for a single vulnerability of the enclave. Tople et al. [59] discussed these attack and proposed *LEVEEFS* as a approach to abstract filesystem management which ensures secure file I/O inside an enclave. The TCB of LEVEEFS is small, compared to Haven where the complete filesystem management is included in the enclave. A small TCB statically guarantees fewer vulnerabilities, which are ported into the enclave. LEVEEFS is achieving that by modifying the host OS, which was avoided in Haven.

Another Microsoft Research group developed the system *Verifiable Confidential Cloud Computing (VC3)* [54] to secure MapReduce computations. VC3 runs on an unmodified Hadoop and shields the MapReduce from the OS or Hypervisor. The TCB of VC3 only consists of the map and reduce functions which are executed in an enclave environment. The MapReduce key-value pairs are encrypted outside the enclave. The deployed enclaves can be tested by standard SGX remote-attestation mechanisms (2.1.5), which also allows the user to securely initiate a secret communciation channel to distribute the encryption key. A patent which covers the VC3 system has been filed on Feb. 7, 2014 by Microsoft [10]. Ohrimenko et al. [48] analysed VC3 regarding information leakage via Side-Channels, which where out of scope in the original VC3 paper. They were able to recover sensitive information although VC3 encrypts all its data.

The researchers behind OpenSGX [37] implemented two use-cases on top of their SGX emulator. A first implementation hardens Inter-domain Controllers which calculate routing paths. In a second model they extended the Tor network with remote-attestation for its nodes to build trust between them. Modified nodes can be detected and excluded from the network.

A patent filed on Sep. 20, 2014 by Horovitz et al. [21] disposes the user-mode limitation of SGX enclaves and proposes an architecture to execute a full-system emulator within a hardware-protected enclave. That enclave can host a virtual machine, which again can host other enclaves (which are managed by the host OS, but used from the enclave VM OS). Nevertheless, there are multiple difficulties which must be overcome to execute a full system in an enclave.

Policy based configuration

Intel and other companies registered patents for technology, mechanisms or use-cases to protect their rights. Often these patents differentiate only in a few aspects. As Costan and Devadas [11] stated the patents of Intel are interesting because they complete the SGX reference papers with some aspects on how to use SGX in the future.

In an Intel patent [55] filed Sep. 10, 2015 is a sensor privacy mode described. That privacy mode allows the device to grant sensor data access rights based on a policy. The privacy mode, along with SGX, allows to regain some control over the behaviour of ubiquitous computing devices, like smartphones.

McAffee patent [49] filed Dec. 4, 2014 extends the previously described Intel sensor privacy patent with location aware configuration control. That allows the setting of a configuration in a mobile device via tracking stations. That can be used by companies to deactivate the microphone of smartphones during a meeting, or disabling the camera in an art gallery or cinema.

Deployment / Provisioning

The Microsoft researchers, who published the Drawbridge ABI [50] and Haven [3], also filed a patent on Jun. 30, 2014 regarding the securely storing of data in public clouds [23]. In essence they describe a model to provision encrypted data for specific CPUs which can only be decrypted with a in-CPU saved secret key. It is similar to the Intel EPID architecture [7] which is used in the SGX provisioning. Another Microsoft patent [14], filed on Oct. 01, 2014, is using that provisioning model to secure management operations in the cloud computing environment.

The Symantec Patent [52] filed on Nov. 20, 2014 describes a deployment model of application containers with remote attestation. It differentiates from known deployment schemes because it not only analyses the application itself, but the deployment environment. If a set of requirements match, the application is transferred to the computer. That could be used to provision SGX enclaves to SGX capable devices.

Identification / Attestation

The attestation scheme of SGX can be used for identification purposes. Intel filed a patent on Sep. 24, 2014 [12] which establishes attestation (2.1.5) between two enclaves via the front cameras and QR-codes which were shown on the display. The attestation can be used to initialise a secret channel between the two devices.

Another authentication patent was filed on Sep. 23, 2014 by Intel [51]. SGX is used to authenticate hardware components of modular device, like 2-in-1 laptops. The standalone display works as tablet and can be authenticated to the bottom (keyboard) part of the device. That authentication can take place via network communication. A secured channel can be established. In consequence, users can use the tablet part of the laptop to access the data, stored on the hard drive in the bottom part of the device.

A McAfee patent, filed on Jun. 27, 2014, [47] describes the usage of remote attestation to securely identify a user against a caller. During the call additional identification data is send to the caller, which allows the identification. This authentication could replace the security questions, asked by financial institutions, which are easy to defeat by social engineering. Another application field is a DRM purpose in which a device can authenticate its identity or licence against the content provider.

Digital Rights Management (DRM)

Digital Rights Management covers the protection of copyright content. DRM systems control how to use, access and distribute digital content. DRM technologies are for example the usage of product keys, limited online registration, persistent online authentication, content scrambling system[20], protected media path[21] or the Advanced Access Content System (AACS)[6]. None of these systems is able to provide perfect protection of the content. The main weakness of the prior DRM approaches, like AACS, were that the decryption keys could be extracted[22] by debugging the memory range of the blu-ray player program.

[20]https://www.cs.cmu.edu/ dst/DeCSS/Kesden/

[21]https://msdn.microsoft.com/en-gb/library/aa376846.aspx

[22]http://forum.doom9.org/showthread.php?t=119871

Intel describes a DRM system based on a patent filed on Dec. 19, 2013 [41]. Two enclaves are used for there DRM system. One enclave inspects a policy and authenticates a second enclave, if the infrastructure meets the policy requirements. The second enclave decrypts the encrypted content. Via the SGX provisioning model can authenticate the enclave its identity against a content provider. The content provider can add the enclave to a whitelist and distribute a whitelist ID to the enclave. This whitelist ID is send to the content provider and checked each time before entering a secured session. The content provider can revoke the whitelist ID when he suspects misuse of data or a media licence expired. In consequence, the secure session will not be initialised and the content will not be decrypted. Via using remote-attestation and provisioning content owners can build a system where the decryption keys cannot be obtained.

2.1.9. Attacks on Enclaved Systems / Untrusted OS

By design it is not possible to access the SGX enclaves memory from the outside. Therefore attacks concentrate on side channels to gain information about the enclave's data.

Intel reminds in a blog post[23] that it is not enough to develop cryptographic libraries which are side-channel resistant. Furthermore, the application must guarantee that no information can be inferred by the observation of access and timing patterns.

Xu et al. [63] introduced controlled-channel attacks which allow the extraction of sensitive applications. They analyse the control flow inside the trusted application by triggering page faults on defined memory regions. That allows the recognition of control flow based on the input. They were able to modify the untrusted part of Microsofts *Haven*[3] to execute this attack on different programs. They were also able to recover text and image data from the enclave.

Tople and Saxena [58] examined different applications regarding input oblivious executions. These execution guarantees that an observer is not able to gather relevant information of the input of the applications. If an application is not input oblivious a *logic-reuse attack* could be carried out that leaks encrypted data.

Checkoway and Shacham [9] developed *Iago attacks* which modify the kernel to return malicious return values on system calls. These can lead to an exploitation of application behaviour to gain information about its input. The Iago attacks make it clear that it is complicated to secure an application against an untrusted operating system. Although Iago attacks are available on other untrusted OS architectures, syscalls are not available in a SGX enclave and therefore forbidden. Furthermore, the available instructions and C functions are limited that the proposed Iago attack on OpenSSL wont be possible.

Ohrimenko et al. [48] observed MapReduce jobs on encrypted data which were decrypted on the nodes inside SGX enclaves. They were able observe and exploit the traffic between the nodes, which leaked sensitive information although the data itself was encrypted.

[23]https://software.intel.com/en-us/blogs/2015/05/19/look-both-ways-and-watch-out-for-side-channels

Instruction	Result	Description
CPUID, GETSEC, RDPMC, SGDT, SIDT, SLDT, STR, VMCALL, VM-FUNC	#UD	Can cause VM exit because VM tries to emulate instr.
IN, INS/INSB/INSW/INSD, OUT, OUT-S/OUTSB/OUTSW/OUTSD	#UD	I/O faults may not safely recover, resulting in VM emulation attempt and possible VM exit
Far call, Far jump, Far Ret, INT n/INTO, IRET, LDS/LES/LFS/LGS/LSS, MOV to DS/ES/SS/FS/GS, POP DS/ES/SS/F-S/GS, SYSCALL, SYSENTER	#UD	Instr. load descriptors from GDT/LDT which change privilege levels. Enclave execution must be CPL=3 and must not be changed.
LAR, VERR, VERW	#UD	Access kernel information which can be used for kernel exploits.
ENCLU[EENTER], EN-CLU[ERESUME]	#GP	An enclave cannot enter another en-clave.

Table 2.2.: Illegal instructions inside an enclave [25].

They proposed an additional shuffle and balance layer to avoid these side-channel information leakage.

2.1.10. Limitations of SGX

The design of Intel SGX leads to different limitations. It heavily depends on the usage of SGX where limitations matter to an enclave developer. The main restriction for developers is the reduced instruction set [25]. The instructions which cannot be executed from inside an enclave can be seen in Table 2.2. These instructions are forbidden because they can be used to weaken the SGX security properties by causing a VM exit or by changing privilege levels.

Another limitation that can cause additional effort in the design and implementation of TresorSGX is that an enclave can only be entered from ring-3, the user space, and not from kernel space. The designated use-case for SGX enclaves is the user-space. That is a big difference to other approaches like the *Arm Trustzone*[2] which enables a secure privileged world.

Intel lists a number of standard C functions which can not be used in an enclave [27]. If these functions are used, the compilation of an enclave will fail because the functions are not implemented in the SGX SDK header files. Some unsupported functions are for example used for complex math, file input output, jumps, signals, string operations and time operations. The trusted SGX SDK functions are hardened to withstand string and buffer overflow attacks.

The current version is SGX1. With SGX2 an enclave developer is able to add memory to an enclave when the enclave has been initialised and is running. Also new threads can be added to a running enclave. With SGX1 the enclave execution is limited in its fixed memory range and thread number.

2.1.11. Future of SGX

As in Section 2.1.8 described numerous use-cases for Intel SGX exist. The reasons to use the SGX technology range from protecting the user against malicious hosts, to enforcing of corporate device policies, provisioning of secrets and digital rights management.

As previously analysed it is not possible to use the Intel SGX without an Intel Launch Enclave. During analysis of SGX it became apparent that the Launch Enclave is an additional mandatory step which is designed to enforce policies. Costan and Devadas [11] mark the Launch Enclave as unnecessary approval instance which allows Intel to force itself as mediator between the enclave developer and the end-user. This intention is not written in any official documentation, but in the Intel patents of 2010 and 2012.

> Enclave Authentication also provides a foundation to outsource Enclave microcode flows, Flexible Sealing & Reporting, as well an enforcement point for a number of new business models. [...] The enclave license indicates who the source/accountable entity for the enclave is, any special capabilities the enclave requires, and any additional information needed for identifying the particular business model/agreement that enabled this enclave. [...] For example, [Vendor] A could purchase a license authorizing them to produce enclaves for use in A's video player. To do this, Intel would create a license for the Vendor A's video player Root Key, along with capabilities that Intel permits Vendor A to use in video player enclaves. Vendor A will then use the video player Root Key to sign individual license files for each video player revision they release. [34] [43]

The licensing scheme for SGX is currently unknown. On an enquiry regarding the licensing scheme, Intel asked for additional information regarding the company and the use case where SGX should be applied. That indicates that Intel will not open the licencing scheme to the public but rather negotiate that in private with each business partner. Such business partners have to perform a set of actions to secure the key material:[24]

- **Secure Software Development** - the enclave must be developed following good programming guidelines described in the Enclave Writers Guide[28]. Futhermore, the developer must notify Intel when vulnerabilities appear in the application. The vulnerabilities must be fixed in a pre defined time. The newest version of the Intel SDK Plattform software, that is required for executing the enclave, must be included and distributed with the application. It is forbidden to write enclaves that behave like malware or spyware. The enclave must not consume all available EPC memory or influence the system stability.

[24] https://software.intel.com/en-us/articles/intel-sgx-product-licensing

An interesting point is that the user-experience should not be influenced if an enclave can not be executed. That conflicts with the DRM use-cases where a media stream can only be decrypted if an enclave can be executed. It becomes apparent that Intel decides problem specific about the granting of licences.

- **Enclave Signing Key Management** - the developers must demonstrate their protection of the key material which follows the industry best practices for key management[25]. These practices contain requirements like multi-factor authentication for access, infrastructure security guidelines and usage of hardware security modules for certificates. Also the developer agrees to inform Intel if any data loss regarding the enclave, keys and certificates happens.

- **Relying Party Functions** - the developed SGX application will be managed and delivered to SGX capable systems using a self hosted distribution system. This systems depends on the Intel Attestation Service. The distribution system must guarantee the ability to withstand attacks which aim to prevent the distribution of patched enclaves. It must be able to process SGX Quotes and deliver SGX Plattform updates. Therefore, it must match the security properties of the Intel Attestations Service (in terms of service licence agreements, DDoS prevention, etc.).

It is currently not possible to highlight the future usage of SGX in detail. Companies must decide if their confidential data, their intellectual property or the rights of their users are worth the cost of implementing a technology which is very prone to vendor lock-in difficulties. Furthermore, an secure development environment and a managed distribution system must be installed which has to match the guidelines by Intel for licensing purposes.

2.2. OpenSGX

OpenSGX is an open source platform for developing and researching the software guard extensions architecture[26]. OpenSGX, developed by Jain et al. [32], provides an emulator for SGX instructions, tools to build enclaves and libraries for interactions with custom enclaves. Their emulator is based on QEMU[27].

During the first part of preparation for this thesis SGX capable hardware was not available or could not be used. Therefore, OpenSGX was evaluated for its usability in the scope of this thesis. Because of the later described limitations it was no further used with the release of the Intel SGX SDK.

[25]https://www.thawte.com/code-signing/whitepaper/best-practices-for-code-signing-certificates.pdf
[26]https://github.com/sslab-gatech/opensgx
[27]http://qemu.org

2.2.1. Motivation

The Intel Software Guard Extensions are an addition to the instruction set of the processor. QEMU emulates the processor by using dynamic binary translation. OpenSGX extends the functionality of QEMU with support for SGX instructions.

However, because of the complexity of SGX it is not sufficient enough to emulate the single user mode instructions. The management of the enclave memory must be performed in a privileged area.

OpenSGX is a complete self-contained software platform. This eases the development and debugging because, in contrast to Intel SGX, a developer can analyse the state of the program and system at any point. No privileged instructions are hidden, or closed source Intel enclaves are involved.

2.2.2. Architecure

As described in the Intel SGX architecture Section 2.1.5, the SGX architecture consists of different components. The components provide functionality for EPC-PRM management, enclave initialisation and creation. The main components and their connections are shown in Figure 2.16. OpenSGX is build on top of QEMU's user space emulation which allows the execution of the host and enclave application. The SGX syscalls are handled by a SGX OS emulation module. This module manages the EPC and PRM via privileged SGX instrucitions.

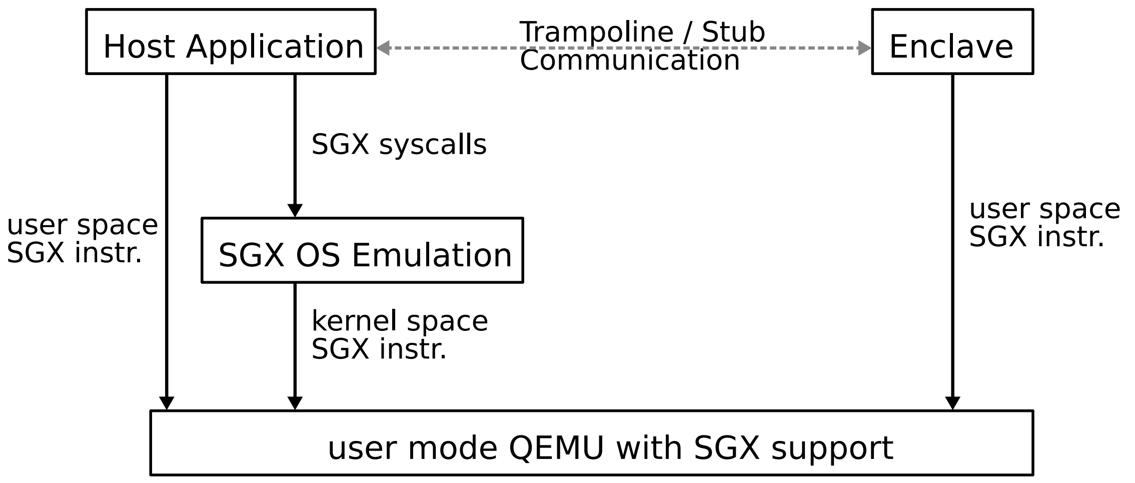

Figure 2.16.: OpenSGX components for executing SGX enclaves in the QEMU context.

SGX instruction emulation The core of OpenSGX is the implementation of the SGX instructions in *opensgx/qemu/target-i386/sgx_helper.c*. Functions are implemented to be conform with the information provided by the Intel manual[25]. When using Intel SGX, the functionality for EPC management is included in a priviledged driver or kernel module. At OpenSGX the memory management / encryption and the new SGX instructions are all included in the helper file.

The OpenSGX team analysed the official documentation in detail to mimic the original SGX behaviour as close as possible. Furthermore, provides the software implementation of the SGX hardware functionality an entry point for researchers who are trying to understand SGX in detail.

OS layer simulation Because Intel differentiates in calls from user space and kernel space, an OS layer must be emulated in OpenSGX. As shown in Table 2.1, an enclave can only be initiated from kernel space and entered from user space.

OpenSGX introduces system calls through an OS layer which provides the privileged functionality. Syscalls are by design illegal in Intel SGX enclaves. OpenSGX provides *trampoline*-functions for in-enclave syscall usage. The state of the enclave will be saved, the context switched to the kernel, the syscall executed in kernel space and then resumed into the enclave. The return value from the kernel will be evaluated inside the enclave. These mechanisms can be implemented on Intel SGX too because the syscalls are simple untrusted function calls and must be implemented in the host application.

SGX toolchain Similar to the build and load tools in Intel SGX tools for OpenSGX exist. A compiling tool generates an OpenSGX binary which can be moved to the EPC by using the EADD instructions. A loader moves the data and text section of an enclave binary into the host program, which can call a syscall to create the enclave. This syscalls loads allocates and creates the required pages in EPC.

SGX user library Like the Intel SGX libraries user libraries for OpenSGX exist too. These libraries can be used in the host application for creating, initialising, entering, resuming and quoting an enclave. Also the tramponline functions are implemented for calling enclave functions.

The libraries for in-enclave use were modified. Like in the Intel SGX libraries functions were removed which would break the enclaves security or are forbidden by design (e.g. syscalls, unsupported C functions).

Calls from the enclave to the outside, for example syscalls, can be implemented using so called *trampolines and stubs*. These functions allow the usage of shared memory for communication purposes. By using this architecture an enclave could setup a networking socket at the host application. In a first step the enclave declares the socket parameters in the *stub*, invokes the socket via a *trampoline* which calls an *EEXIT*. The host application will execute the *trampoline* handler with the *stub* parameters. The results will also be saved in the *stub* and the host application calls an *ERESUME*. This system is very generic and can serve multiple purposes.

A similar implementation could be useful for Intel SGX. However, the outside functions must be implemented in the host application which calls the enclave. This computing model would ease the import of complete programs which rely on certain syscalls. Nevertheless are these syscalls, calls to an untrusted component of the system. A security critical application which relies on, for example a correct time which is retreived from a syscall can be tricked by the host application. This gives a small insight into the difficulty of porting applications into an enclave environment because standard applications trust these syscalls. A secure enclave application must never trust the outside functions and encrypt all its data which is send over outside functions.

2.2.3. Limitations

The overall programming model of OpenSGX is equal to Intel SGX. The enclave lifecycle, the build process and the signing works like they were performed on real hardware. However, OpenSGX's implementation differs in a few details.

The EINITOKEN, which is required to initialise an enclave and is not signed by Intel, can only be retrieved inside an enclave on the Intel SGX platform. OpenSGX does not have such limitations, therefore the key can be generated with an included keytool.

Finally OpenSGX is based on the user mode emulation of QEMU. QEMU supports two modes of operation. In *full system emulation* QEMU emulates the complete computer with all its components. This allows the usage of full operating systems inside of QEMU. The *user mode emulation* is a virtualisation modus for applications. This emulates the same operating system on multiple CPUs for the application. OpenSGX is based on user mode emulation and it is therefore not possible to execute OpenSGX in an emulated Linux kernel. This detail makes its usage to implement TresorSGX impossible because of the missing kernel. However, the TresorSGX enclave itself can be executed and debugged in OpenSGX.

2.3. Linux Kernel

The open source operating system Linux was created in 1991 by Linux Torvalds. Since then the OS evolved and expanded to support many different platforms. Unlike other operating systems the source code of Linux is open and receives patches from thousands of volunteers around the world[28]. The kernel can be extended via loadable-kernel-modules. These modules are executed in kernel space and do not require a recompiling and restart of the kernel itself. They can be inserted during runtime.

In this thesis a implementation of a cold-boot resistant drive encryption module which is initiated by the Crypto API will be described. The Crypto API is located in the Linux Kernel but can be used by user space processes. The implementation uses SGX for the

[28]www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2015

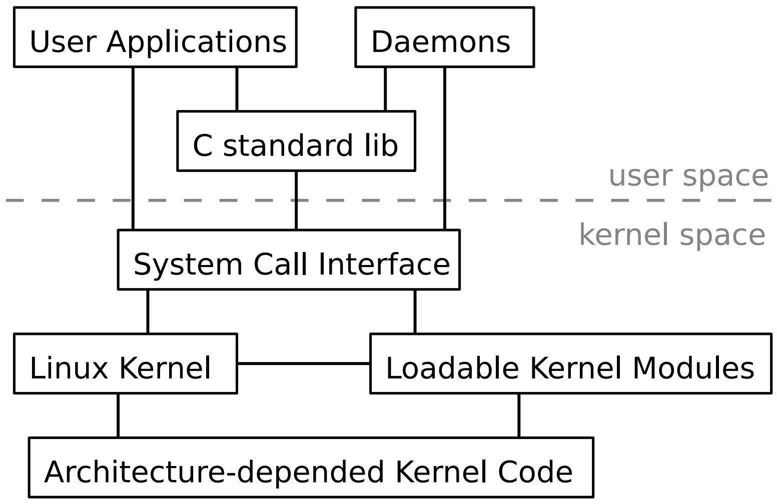

Figure 2.17.: An overview about the Linux architecture.

secure protection of the cryptographic keys. A limitation of SGX is that an enclave can only be entered from user space - ring 3 (see section 2.1.10). That makes it impossible to build the crypto module Tresor completely in kernel space. In the following, a brief overview about the Linux Kernel architecture, the Crypto API and methods to interact with user space processes will be given.

2.3.1. Architecture

The main differentiation in Linux can be made between user and kernel space. As shown in Figure 2.17 are user applications and the standard C library are executed in user space, whereas the systemcall interfaces and the Kernel itself resides in Kernel space.

The User and Kernel space is clearly separated. User space processes allocate memory in a different virtual address space than the Kernel, which allocates a single address space. The kernel is executed on CPU ring 0, which is the level with the most privileges and direct access to the CPU, RAM and the devices. The user space processes are executed on ring 3 with reduced privileges.

Although the Linux kernel is a monolithic kernel, kernel modules can be loaded and unloaded when the system is running. The kernel-internal application programming interface is not stable - in contrast to the userspace-kernel-API which is stable and maintains backwards compatibility[29]. If an in-kernel interface must be modified because of a security or bug fix all instances where this interface is used are modified. This leads to a very high development speed of the Linux kernel itself. Another positive effect of this approach is that once a device driver is in the main kernel tree, the original developer of the driver must not modify the driver on every interface change. That is done by the developer, who introduces the interface modification.

[29]https://www.kernel.org/doc/Documentation/stable_api_nonsense.txt

2.3.2. Interaction with User space

There are multiple ways to interact from the user space with the kernel space. As previously described the kernel-user API is stable. User space processes can interact with the kernel via calls to the *System Call Interface* or by using the GNU C library, which is a layer on top of the System Call Interface. For the implementation bidirectional communication between a kernel module and an user process is needed.

In Linux three different types of device modules which enable bidirectional communication exist[30].

Char / block devices

Char devices are accessed via byte streams. They are typically used in open, close, read and write system calls. Known char devices are the */dev/tty* ports, the */dev/console* or */dev/null*. These devices can be accessed like files, but can only be read sequentially.

Block devices are accessed in Linux like char devices, but are managed by the kernel in data blocks. Data operations transfer or manipulate whole blocks instead of a byte stream.

Procfs is a virtual filesystem which can be used to access kernel information.[31] Many kernel information and configurations can be retreived via this interface. */proc/crypto* for example reveals information about available cryptographic ciphers. The information is dynamically generated when a read attempt on the files is initiated, so no persistent data is stored in a file. Via different functions data can be written to the user space and back to the kernel space[32].

Sysfs is another virtual filesystem which allows the exchange of data via the `/sys/` directory[33]. It is meant to export kernel data structures to the user space. Sysfs files can be read or written. These actions trigger specified callbacks in kernel space.

Network devices

Network devices can be physical network interfaces or software interfaces, like Netlink sockets or a loop back interface. A network device exchanges message packets with another entity, based on addresses and policies.

Netlink sockets can be used for full-duplex communication between kernel and user space [34]. The prior listed methods involve modification of existing kernel modules and in consequence an obligatory recompile of the kernel. That leads to polluting the kernel space

[30]http://www.makelinux.net/ldd3/

[31]https://www.kernel.org/doc/Documentation/filesystems/proc.txt

[32]https://www.ibm.com/developerworks/library/l-proc/

[33]https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt

[34]http://www.linuxfoundation.org/collaborate/workgroups/networking/generic$_{n}etlink_{h}owto$

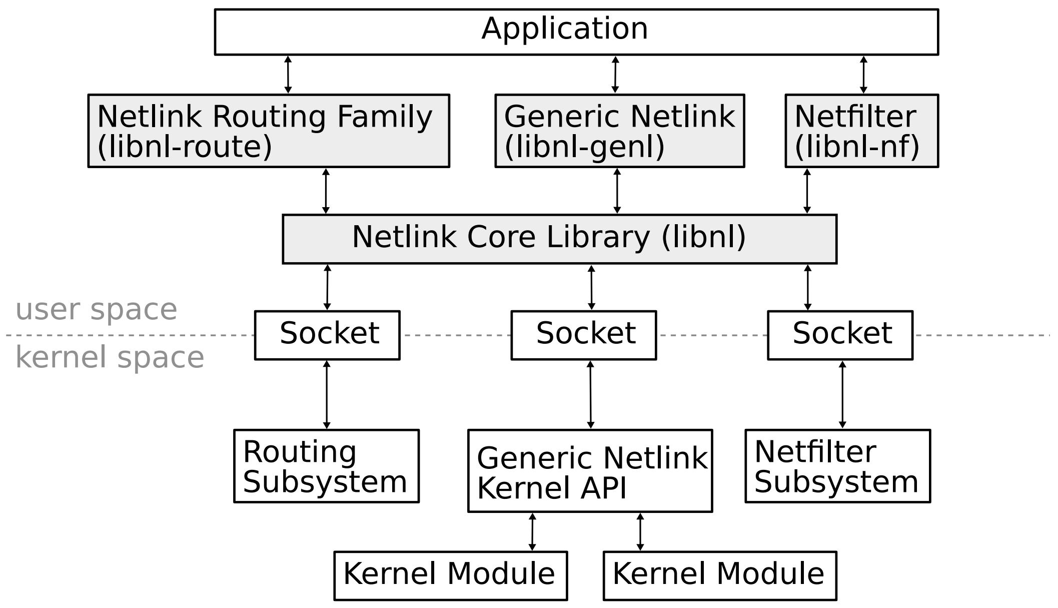

Figure 2.18.: The Linux Netlink Architecture with the user space Netlink library (filled with grey)

with features that lead to possible stability or security problems[35]. By defining a Netlink family and operation, a kernel module and an user process can register at a Netlink socket. If messages are send to this family, the relevant action will be triggered and is able to parse the message.

Unlike syscalls the Netlink model allows an asynchronous handling of the messages. Another benefit of the Netlink is that the user space application must not poll the device periodically for a new message from the kernel. By defining a Netlink operation a specific callback function is set which is directly called when a new message arrives. However, the Netlink protocol is much more complex than operations on procfs or sysfs devices. The user space Netlink library[36] abstracts the fundamentals of the object based kernel socket API to simplify the usage.

Figure 2.18 shows the Netlink architecture with the user space *libnl* library. The routing family is used for network related communication. The netfilter is for network filtering purposes.

2.3.3. Crypto API

The Linux Crypto API[37] is a well defined set of methods to use various cryptographic algorithms with minimum configuration [42]. By using the Crypto API the cryptography can be easily separated from other program logic. Furthermore, it is possible to use the

[35]https://www.linuxjournal.com/article/7356
[36]http://www.infradead.org/ tgr/libnl/
[37]https://www.kernel.org/doc/htmldocs/crypto-API/Intro.html

Crypto API transformations from user space processes. The keys are saved by the Crypto API which ensures that the keys cannot be retrieved in user space. The generic Netlink library allows the communication with the Generic Netlink API. This library is used in the later proposed implementation for TresorSGX.

Cipher Management

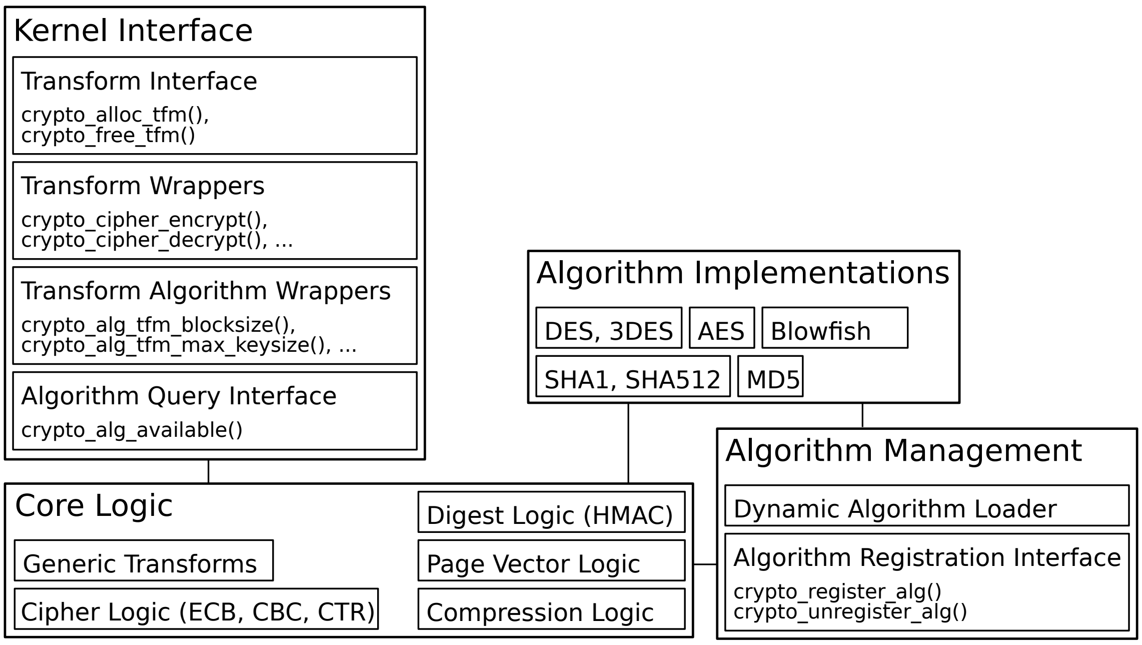

Figure 2.19.: Crypto API architecture. Crypto interfaces are kernel-wide available.

The crypto API is layered to hide the core logic from cryptographic user and algorithm implementers[38] as described in Figure 2.19. The Kernel API provides generic interface to initiate the cryptography, to encrypt and decrypt data and to gain information about available cryptographic algorithms and their specification (e.g. blocksize). The user is able to use a wide variety of algorithms with minimal programming effort[39].

The Kernel Interfaces include methods for allocating and releasing a cipher. The cipher is identified by a name string which must be registered first. Different transform wrappers are available for encrypting single data blocks or bigger data. With the help of transform algorithm wrappers an user is able to gain knowledge about the supported or required block- and keysizes. The Crypto API kernel interfaces trigger actions in the core logic of the Crypto API.

The core logic provides different templates that can be used with single block ciphers. That allows the implementation of a single block cipher which can be used with multiple

[38]https://www.linuxjournal.com/article/6451

[39]https://events.linuxfoundation.org/sites/events/files/slides/lcj-2014-crypto-kernel.pdf

```
name         : aes
driver       : aes-aesni
module       : aesni_intel
priority     : 300
refcnt       : 7
selftest     : passed
internal     : no
type         : cipher
blocksize    : 16
min keysize  : 16
max keysize  : 32
```

Figure 2.20.: /proc/crypto excerpt of the aesni aes crypto cipher

modes like *counter mode (CTR)*, *cipher block chaining (CBC)*, *electronic codebook(ECB)* or *XEX-based tweaked-codebook mode with ciphertext stealing (XTS)*.

Usage

A cipher must first be registered at the Crypto API before it can be used. The cipher must be provided in kernel space. A loadable kernel module can register the new cipher during runtime. The cipher is registered with information about its name, cipher type, blocksize, keysizes and the callback functions for the setkey, encrypt and decrypt functions. Cryptographic information about the registered cipher can be retrieved by reading the device */proc/crypto*. Listing 2.20 shows the output of the *aesni aes cipher*. The properties of a registered cipher will be discussed in the TresorSGX cipher section 3.3.2.

When the new cipher is registered it can be used by allocating a crypto transformation and providing a key via the generic crypto API function. The allocated cipher can be used with the generic crypto API encrypt and decrypt functions.

The described interfaces are primarily for the usage in kernel space. However, the Crypto API can be used from user space, too. The kernel exports message digest ciphers, symmetric ciphers, AEAD ciphers and random number generators via a Netlink family/type. A user space application must generate a Netlink socket with family/type *AF_ALG* with the cipher parameters. By invoking the send and reveive methods on that socket data can be transferred. The user space library *libkcapi*[40] wraps around the Netlink Crypto API interface. Developers can use its API which abstracts from the underlying socket handling.

A more specific usage of the Linux Crypto API is *dm-crypt*[41]. *dm* is the device mapper which allows the usage of physical block devices via virtual block devices. dm-crypt applies encryption to these devices. It allows the usage of block devices by encrypting writes and decrypting reads to that device. The user defines which cipher and parameters

[40]http://www.chronox.de/libkcapi.html

[41]https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt

he wants to use and sets an password in a setup. dm-crypt will encrypt all blocks with the specified cipher. The Crypto API is used for these transformations. An user can mount the encrypted device and use it as any other mapped device.

3 TresorSGX

Based on the foundation build in the last chapter, a SGX enabled cold-boot attack resistant cryptographic module TresorSGX for the Linux Kernel will be proposed. TresorSGX implements a secure Intel SGX enclave which is used to save the cryptographic key material and methods for encryption and decryption. As discussed in section 2.1.10 a limitation of Intel SGX is that the enclave must be entered in user space (ring 3) and is also executed in this mode. This forbids the usage of the enclave in kernel space only. However, a new cryptographic cipher in the Crypto API can only be registered by using a loadable kernel module, which must include the cryptographic functions. Therefore a three layer approach is described in the next sections.

3.1. Motivation

Disk encryption is a standard procedure to secure the confidentiality and integrity of computer systems. Especially on mobile devices it is recommended to use this feature to protect the users data. When the device is turned off the data is at rest and no cryptographic keys are saved on the device. When it is turned on the user must insert a passphrase to initiate the cryptographic cipher. The key material is saved in RAM when using standard disk encryption tools.

3.1.1. Original Tresor

The original Tresor, developed by Müller et al. [46], uses the processor registers instead of RAM to store the encryption states and the cryptographic keys throughout the runtime of the system. This increases the security because the key resides not in RAM which is prone to multiple attacks.

Tresor uses the CPU debug registers as secure key storage. The encryption / decryption of one block is handled in an atomic session to prevent context switches. This guarantees that no sensitive data can be accessed during context switches. Tresor registers its symmetric block cipher at the Linux Crypto API, which provides multiple cryptographic modes for the cipher. Tresor applies patches to different kernel modules to prevent the modification of the debug registers. Therefore, the debug register can not be used when Tresor is configured.

Multiple attacks on cryptographic material in RAM exist and some can be used against Tresor, too. *Cold boot attacks* on DRAM allow the retrieval of encryptions keys after the system is turned off [19]. Müller and Spreitzenbarth [45] developed a method to gather encryption keys from Android phones by using cold boot attacks. *Direct Memory Access (DMA) attacks* , by using ports like Firewire[5] , Thunderbolt[40], ExpressCard[22], USB OTG[33] allow the extraction of key material in RAM. Tresor stores the encryption keys not in RAM but in CPU registers, therefore it should be save against cold-boot attacks or DMA attacks. However, Blass and Robertson [4] developed a DMA attack to extract the Tresor keys from the registers by changing the kernel control flow. *Bus Monitoring Attacks* enable sidechannel attacks on encryption algorithms. By monitoring the access on public pre computed value tables it is able to break AES, even if the cryptographic keys are not saved in RAM [61]. Also the original Tresor is vulnerable against that side channel attack.

During the next sections TresorSGX will be described. References to the original Tresor will be declared as such. References to just Tresor imply references to the TresorSGX system.

3.1.2. Benefits of SGX

TresorSGX extends the original characteristics by using SGX enclave and sealing techniques. The usage of TresorSGX makes the system cold-boot resistant against an attacker. The usage of a sealed salt (as described in section 2.1.5) adds another factor to the encryption key generation. The salt is sealed with a passphrase that is specific to the used processor and enclave. No other enclave and processor combination is able to unseal the salt.That hardens Tresor because the password alone is not enough to decrypt the data. This two-factor approach can be used to encrypt data on a removable storage, which can only be accessed on a specific computer. If the sealed salt is saved on a removable device, it will require its physical presence for encryption and decryption.

3.2. Design

To implement a cold boot and DRM attack resistant cipher for the Crypto API different requirements must be considered.

- **Enclave Management** Which application launches the enclave and calls its trusted functions.
- **Communication** How is the communication between the enclave's host application in user space and the crypto API in kernel space established.
- **Enclave Crypto** Which methods are implemented as trusted functions? What are the available cipher and cryptographic modes?

3.2.1. Enclave Management

As described in the SGX architecture section 2.1.5 it is not possible to enter an enclave from kernel space. The enclave's code is executed in ring-3 with a reduced set of instructions (see SGX limitations 2.1.10) and a limited amount of available memory in the *Processor Reserved Memory*. Furthermore, it is not possible to initiate the enclave on its own, an Intel Launch enclave must be used to generate the enclaves launch token (see 2.1.5).

The initial approach was to maintain and enter the enclave in kernel space. A kernel module should register the new cipher at the Crypto API and manage the SGX enclave. However, with the release of more and more Intel SGX documentation it became apparent that this is not possible.

These difficulties lead to a different approach. The enclave is implemented in an user space service or daemon. The user space host application is developed in Linux and is therefore referred to as daemon in the later description. That daemon calls the Intel Launch enclave for initialisation, which provides a launch key for the enclave. Once the enclave is running, function calls to the enclave can be made for saving the cryptographic material and encrypting and decrypting data.

For faster development speed the Intel SGX SDK is used. These SDK's provides libraries and tools for enclave creation and communication. It is possible to interact without these libraries with the enclave. However, the libraries and tools are rather complex and a self implementation would lead to unneccessary overhead on top of the TresorSGX development. Furthermore, the SDK usage guarantees that the implementation is conform to the SGX specifications because the enclave is analysed during the building process.

The daemon can be started[1] by the kernel with the help of the usermode-helper API[2]. After initialisation the daemon can inform the kernel module about its state and data can be transferred.

[1]https://www.ibm.com/developerworks/library/l-user-space-apps/index.html

[2]https://www.kernel.org/doc/htmldocs/kernel-api/API-call-usermodehelper-setup.html

3.2.2. Communication with Crypto API

Different communication protocols for exchanging data between kernel and user space were analysed in section 2.3.2. The Network devices had the advantage that an implementation is not accompanied with modifying the Linux kernel. In contrast to the original Tresor it is not necessary to patch the Linux kernel itself, recompile it or reboot the system to use TresorSGX. Further, it is possible to implement callback functions on incoming Netlink messages in kernel and in user space. The other approaches had the disadvantage that an user space application must poll for new messages from the kernel.

The drawback of the Netlink communication is the reduced throughput. Nevertheless Netlink interfaces are used in TresorSGX because the scope of this thesis is to design and implement a proof of concept for moving a kernel functionality into an enclave. The performance is of secondary importance.

3.2.3. Enclave Cryptography

To provide cryptographic functions that encrypt and decrypt data via the Crypto API a module must provide at least an implementation of `setkey`, `encrypt` and `decrypt`. As in section 2.1.10 described some instructions in an enclave are forbidden. Executing these instructions lead to a fault and the enclave execution stops.

Intel provides a cryptography library in their SGX SDK, primarily for the use in SGX functions. Therefore, only a limited number of ciphers are implemented in the `sgx_tcryto` library [27]. The library includes functions for *sha256* creation, *AES 128bit GCM, CMAC, CTR* cryptography, *256 bit Elliptic Curve* cryptography, signing and verifying of signatures based on the *ECDSA scheme*. The SGX crypto library is developed for in-enclave usage. Therefore it is usable without any restrictions. However, the source code is only provided as binary.

To provide a new cipher for the kernel Crypto API, a simple block cipher must be implemented. When searching for crypto libraries, which contain multiple ciphers and encryption modes, *OpenSSL*[3] stands out as open source fully featured library for multiple protocolls. An enclave which contains the full OpenSSL library could be used by numerous user mode applications which depend on OpenSSL. To benefit of the Intel AESNI instructions the OpenSSL EVP interface is used.

Different problems were encountered during the porting of OpenSSL into the enclave. In an enclave it is not possible to use filepointers to read or write files [27]. However, OpenSSL implements these functions (beside other in-enclave forbidden calls) in numerous functions. Neither setting options like `OPENSSL_NO_STDIO` nor including only relevant modules would work. It was not possible to build OpenSSL with only the EVP relevant modules because of to many cross dependencies. A modified C library which

[3]https://www.openssl.org/

provides dummy functions or outside calls for forbidden calls could make the usage of OpenSSL in an enclave possible. However, such library is not yet available.

mbedTLS (PolarSSL) advertises itself as easy to use, modular and loosely coupled crypto library. During development it became clear that the configuration possibilities of mbedTLS are superior to OpenSSL. Compiling and linking the enclave against the pre-built mbedTLS library worked. However, during signing an error happened which could be traced with the SGX Evaluation SDK User's Guide [27] to a problem regarding the use of trusted libraries inside an enclave. The included libraries are not allowed forbidden instructions. Furthermore, the called functions must be statically linked to the enclave. By including the mbedTLS source directly the enclave was able to use the mbedTLS cryptography modules. However, problems occured with an invalid opcode trap during executing AESNI instructions and a stack segment trap when encrypting with software AES. These problems must be analysed and debugged in order to make mbedTLS usable in an enclave. In general mbedTLS looks more maintainable for this kind of application field than OpenSSL. These difficulties give an idea of the effort that is required to port an existing library into an enclave.

Because of the prior discussed difficulties it was decided that taking a minimal approach of using the small Intel AESNI library[4] was the best solution. This library provides AESNI cryptography for multiple keysizes (128, 192, 256bit) on multiple modes *block, CBC, CTR*. The library can be successfully included in the enclave and is able to run AESNI cryptography from inside of the enclave.

To generate an encryption key in the enclave which is based on the user password and a sealed salt the Password-Based Key Derivation Function 2 (PBKDF2) is used [36]. This function is standardised in the RSA Laboratories' Public-Key Cryptography Standards (PKCS)[5]. It is recommended for new applications which implement password based cryptography. The function is defined as $DK = PBKDF2(PRF, PWD, SALT, ITER, DKlen)$. The derived key DK with length $DKlen$ is generated by a pseudorandom function PRF e.g. a HMAC [38]. The specific parameters for the function are the user password PWD, the salt $SALT$ and the number of iterations $ITER$ of the pseudorandom function. The iterations, the salt and a version number are saved in a SGX sealed container for later use.

3.3. Implementation

The obstacles that had to be overcome to develop the TresorSGX cryptography system are described in the prior section. In the following, a high level summary of the workflow is given and the used components are described in detail.

An overview about the TresorSGX architecture can be seen in Figure 3.1. The following components were developed during this work. The basis is the *Tresor Enclave* which

[4]https://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library/

[5]https://tools.ietf.org/html/rfc2898

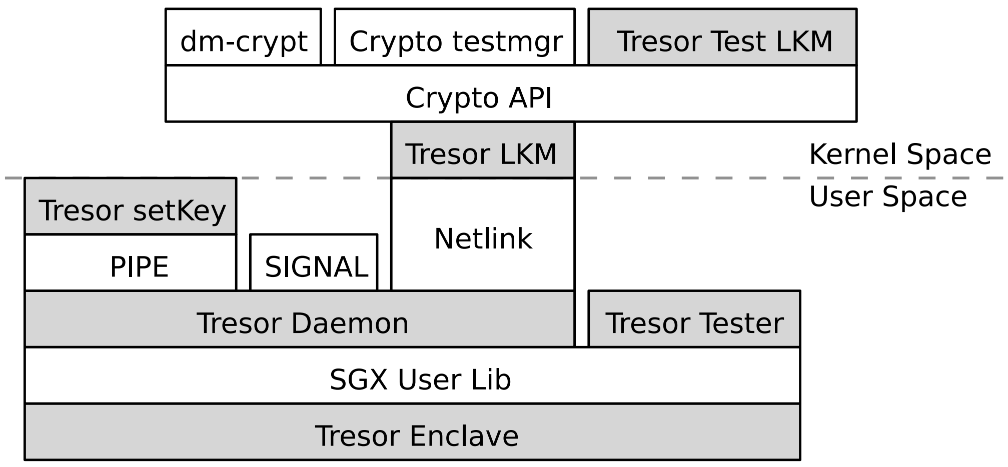

Figure 3.1.: Overview of TresorSGX architecture. Grey components were designed and developed during this work.

provides the cryptographic methods and stores the encryption key. The communication interfaces between the enclave and its host application, the *Tresor Daemon*, are generated by the SGX SDK library. For testing purposes the *Tresor Tester* can call the enclave's functions, too. In general, every other application could use the features provided by the enclave.

The Daemon creates the enclave. When the daemon process is started it forks itself to a daemon. It can be shutdown with signals. It communicates via Netlink with the *Tresor LKM*. This Loadable kernel module registers the cryptographic cipher at the Crypto API. The callback functions for this crypto are implemented in the Tresor LKM. The *Tresor Test LKM* can be used for testing purposes. Other Tools and Modules that use the Crypto API, like the Crypto API testmgr or dm-crypt, can use the Tresor Enclave with that architecture.

The Tresor *setKey program* is used to set the user key independent of the Crypto API to harden the system against cold boot attacks. This program communicates over a pipe with the daemon if it is configured to do so. The user keys from the Crypto API will be ignored in that scenario.

3.3.1. Lifecycle

In contrast to the original Tresor it is not able to include the whole functionality in a single kernel module because the enclave must be entered from user space. In Figure 3.2 the sequence of the initialisation, the key setting process and encryption is shown. When the kernel module is initialised, it registers a Netlink family for communicating with the Tresor daemon. If the Netlink socket is created it starts the daemon via the user-mode helper API. The daemon starts and creates the enclave.

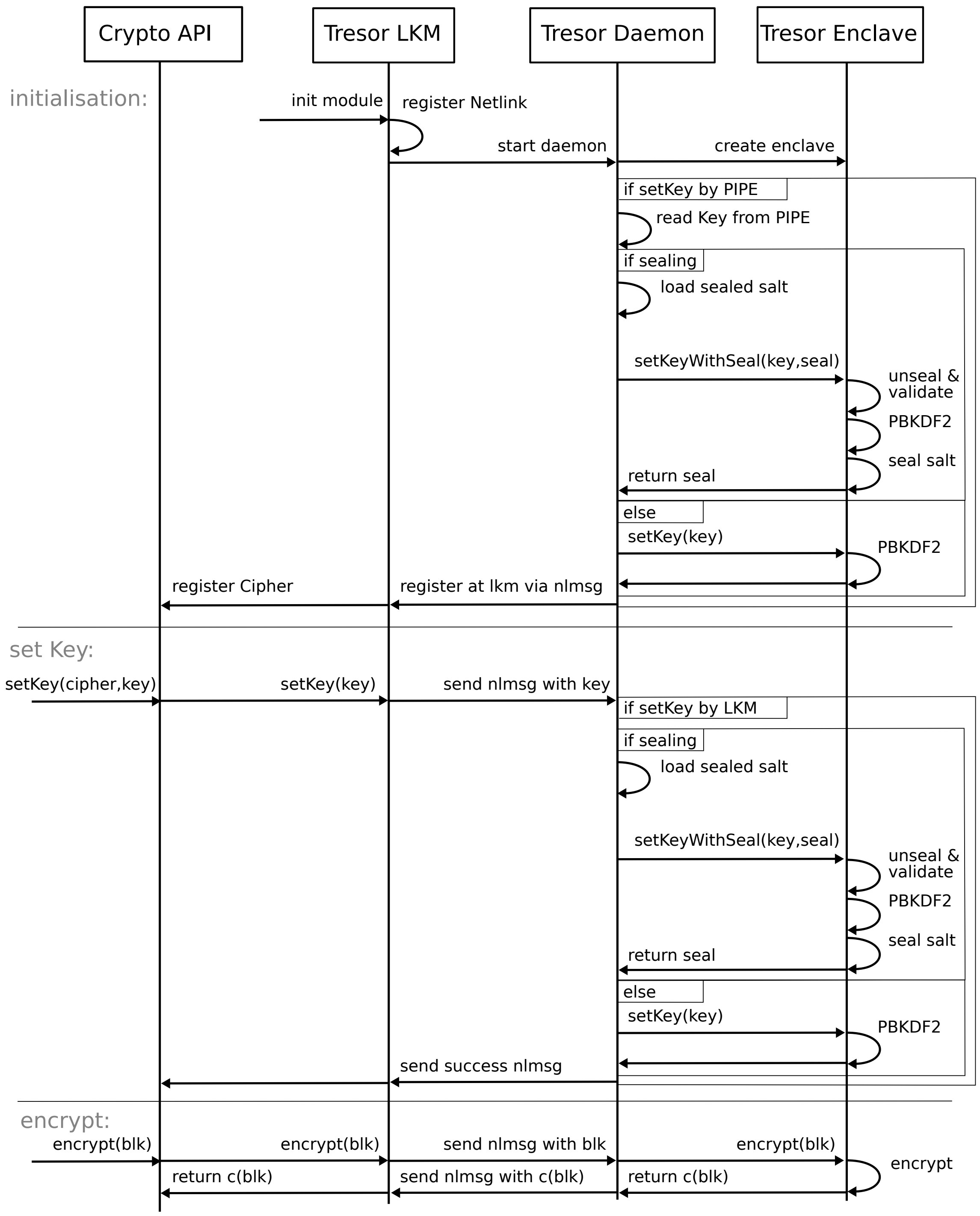

Figure 3.2.: Sequence of the initialisation of TresorSGX, a setKey and encrypt call by the Crypto API.

If the daemon is configured to use user keys from the a pipe instead of the Crypto API it opens a named pipe on a predefined location. The daemon then waits until a user key is written to that pipe. The advantage is that the user key is not copied in multiple modules where it can be accessed by different programs, or optained in a cold boot attack. When the key is read, the daemon sets the key based on the sealing configuration.

The following flow depends on the daemon's configuration. If sealing is enabled, it will load a predefined file from disk. If that fails for any reason, the daemon will generate a sealed data block filled with zeros. It then calls the key setting function with the sealed data at the enclave. The enclave checks if the sealed data is valid and unseals it. This process is described in detail in section 3.3.4. If the sealed data is not valid, it will generate a new salt and seals it. A *Password-Based Key Derivation Function 2 (PBKDF2)*[35] is used to generate a hash-based key on the salt and the password. This key is the AES key for in-enclave cryptography. The enclave function returns with the sealed data, which can be saved by the daemon.

If no sealing is configured the daemon just calls the key setting function in the enclave with the key as parameter. The enclave generates the AES key with the PBKDF2 function and returns.

After the key setting the daemon creates the same Netlink interface as the kernel module and sends a message, that the initialisation succeeded, to the kernel. The kernel receives the message and registers the new cipher at the crypto API.

The Crypto API key setting is triggered by a process which allocates the *tresorsgx* cipher and calls the `setKey` function with the allocated cipher. The Tresor LKM registered an own function as callback for `setKey`, which will be called by the process. This function sends a message with the key and the operation *setKey* to the Tresor Daemon.

The Daemon parses the message and determines if sealing is configured or not. It then sets the key, as described during the initialisation phase.

The daemon finishes the set key routine by sending a success Netlink message to the kernel. The Kernel module blocks its CryptoAPI-setKey call until it receives this netlink message as described in section 3.3.2. It then de-blocks and the crypto API function call returns.

The encryption or decryption process is straight forward. The Tresor LKM encrypt / decrypt callback function is called by the user of the Crypto API. The LKM sends a Netlink message to the daemon, which calls the encrypt / decrypt function at the enclave with the data block. The enclave performs the cryptographic routine and returns the encrypted or decrypted block. That block is send to the kernel module via Netlink. The kernel module copies the block to the destination given in the function parameters and returns.

If the LKM is not able to send a message to the daemon, the module will print an error message to the syslog and return the function. Blocking states in kernel space are difficult because no entity is able to remove the blocked module and the whole system will halt eventually. Therefore, a timeout is applied to the blocking function to avoid such situation.

```
static struct crypto_alg tresor_alg = {
    .cra_name           = "tresorsgx",
    .cra_driver_name    = "tresorsgx-driver",
    .cra_priority       = 100,
    .cra_flags          = CRYPTO_ALG_TYPE_CIPHER,
    .cra_blocksize      = AES_BLOCK_SIZE,
    .cra_ctxsize        = sizeof(struct crypto_aes_ctx),
    .cra_alignmask      = 3,
    .cra_module         = THIS_MODULE,
    .cra_list           = LIST_HEAD_INIT(tresor_alg.cra_list),
    .cra_u  = {
        .cipher = {
            .cia_min_keysize    = AES_MIN_KEY_SIZE,
            .cia_max_keysize    = AES_MAX_KEY_SIZE,
            .cia_setkey         = tresor_crypto_setkey,
            .cia_encrypt        = tresor_crypto_encrypt,
            .cia_decrypt        = tresor_crypto_decrypt
        }
    }
};
```

Figure 3.3.: TresorSGX loadable kernel module crypto algorithm structure

3.3.2. Kernel Module

The Kernel Module consists of multiple parts. One part handles the Crypto API transformation calls, another part manages the Netlink communication to the Daemon. In the following, the usage of the Crypto API and the Netlink interface are described. Based on these information the detailed program flow is shown.

Crypto API The TresorSGX Kernel Module registers the *tresorsgx* transformation at the Crypto API. When registered, the cipher can be used by cryptography users. The structure shown in Figure 3.3 is used to register and identify the new cipher.

The crypto algorithm identification structure[6] contains multiple fields which identify the cipher and its parameters. `cra_name` is a generic identifier of the algorithm which is used by the kernel to look up the provider of the transformation when requested. The name must not be unique. `cra_driver_name` must be a unique name of the provider of the transformation. The `cra_priority` is used to choose an algorithm if multiple transformations with the same name exist. The priority is not used in the TresorSGX system because only one *tresorsgx* cipher is registered. The `cra_flags` describe the transformation further[7]. They identify the type of the transformation. In this case a single block cipher is defined. `cra_blocksize` defines the minimum blocksize for a transformation. If the cipher user uses a smaller block, an error will be returned by the Crypto API. `cra_ctxsize` defines a memory size of the cipher context for the Crypto API. The

[6]https://www.kernel.org/doc/htmldocs/crypto-API/API-struct-crypto-alg.html
[7]https://www.kernel.org/doc/htmldocs/crypto-API/ch02s06.html

```
name         : tresorsgx
driver       : tresorsgx-driver
module       : tresorlkm
priority     : 100
refcnt       : 1
selftest     : passed
internal     : no
type         : cipher
blocksize    : 16
min keysize  : 16
max keysize  : 32
```

Figure 3.4.: TresorSGX /proc/crypto definition

input and output buffer must be aligned to the `cra_alignmask`. `cra_module` defines the kernel module which provides the functionality for the transformation. `cra_list` contains a pointer to the list itself and is defined for Crypto API internal use. *cra_u* defines the callback functions and parameters which match the flags field. A minimal (128bit) and maximum (256bit) keysize is declared and the callback functions for set Key, decrypting and encrypting are linked. The usage of the transformation is described in the *Tresor Test LKM* section 3.3.2.

The configuration of the available cryptographic ciphers can be retrieved by reading `/proc/crypto`. In Figure 3.4 the configuration for the TresorSGX cipher is shown which is equal to the structure that was used to register the new cipher at the Crypto API.

Netlink As shown in figures 3.1 and 3.2 the kernel module is only used as middleman in the TresorSGX architecture. In the early phase of this thesis (October 2015) it was assumed that enclaves can be executed in kernel space and therefore a kernel-only architecture can be achieved. However, during the detailed analysis of the SGX Programming Reference [25] it became clear that this is not possible. The enclave must be initialized in kernel space, but an enclave enter can only happen in user space. This is achieved by a daemon, described in section 3.3.3. To communicate with the daemon *Netlinks* are used. In our scenario most of interaction is initiated by the kernel module. Other possibilities (2.3.2) to exchange data between user and kernel space are based on the assumption that the interaction is initiated by the usermode application. This is not the case with the TresorSGX architecture, therefore Netlinks are used because it allows bidirectional communication.

The Generic Netlink subsystem manages the Linux Kernel Netlink communication. A *Generic Netlink Family* is defined which acts as server for messages on the Netlink bus. To act on messages on the Netlink bus an operation, a callback function, must be registered.

The Tresor message structure, as shown in Listing 3.5, contains the crypto operation (setkey, encrypt, decrypt), a char array which contains the payload and the length of the payload. The policy defines a single attribute and the length of the message structure. The

```
/* Tresor Netlink message struct */
struct tresor_nl_msg {
    unsigned int operation;
    unsigned int text_len;
    char text[32];
};

/* Tresor Netlink message attributes */
enum {
        TRESOR_NL_ATTR1_MSG,
        __TRESOR_NL_ATTR_MAX,
};

/* Tresor Netlink family definition */
static struct genl_family tresor_nl_gnl_family = {
    .id = GENL_ID_GENERATE, // genetlink generates an id
    .hdrsize = 0,
    .name = "TRESOR_NETLINK",
    .version = 1,
    .maxattr = __TRESOR_NL_ATTR_MAX,
};
```

Figure 3.5.: TresorSGX Netlink Family Definition

Netlink family definition contains the name *TRESOR_NETLINK*, the family version (can be used to invalidate old implementations) and the maximum attributes.

The Netlink operation `TRESOR_NL_CMD` is defined as shown in Listing 3.6. If a message is read on bus for family `TRESOR_NETLINK`, it will be validated with the `tresor_nl_gnl_policy`. Then the callback function `tresor_nl_cmd` is called which parses the incoming message.

When the Tresor daemon writes to the Netlink bus for the Tresor family the *port id* of the daemon is saved. This port id is used for later communication with the daemon.

A main problem during the Netlink implementation was that the documentation and most available usage examples were written for a Linux kernel older then the 4 years old version 3.2. The most promising Netlink example repository was updated[8] with support for up-to-date kernel versions to generate some value for the open source community.

Program flow During the design and implantation of the kernel module different challenges had to be overcome. The main problem is that the cryptography user is a transformation request at the kernel module, but the provider of the cryptographic function is the enclave which is not callable by the kernel module. The kernel module sends a message, which contains the request of the user to the daemon as described in Figure 3.2. The control flow is then broken, because the daemon parses the message, performs the transformation and sends a message back to the kernel which triggers the Netlink callback

[8]https://github.com/mdcb/kernel-howto/commit/750468b56ab7effb0219b3afbbec3c0a2824c8fd

```
/* Tresor Netlink callback function for message parsing */
int tresor_nl_cmd(struct sk_buff *skb_2, struct genl_info *info)
{
        /* message parsing */
}

/* Tresor Netlink message policy */
static struct nla_policy tresor_nl_gnl_policy[TRESOR_NL_ATTR_MAX + 1] =
{
    [TRESOR_NL_ATTR1_MSG] = { .len = sizeof(struct tresor_nl_msg)},
};

/* Tresor Netlink operation definition */
static struct genl_ops doc_exmpl_gnl_ops_echo[] = {
    {
    .cmd = TRESOR_NL_CMD,
    .flags = 0,
    .policy = tresor_nl_gnl_policy,
    .doit = tresor_nl_cmd,
    .dumpit = NULL,
    },
};
```

Figure 3.6.: TresorSGX Netlink operations definition

function in the kernel.

The Netlink callback function is completely unrelated to the Crypto API transformation function of the kernel module. One challenge is to guarantee that the Crypto API callback functions are able to return the payload of the daemons Netlink messages. In Figure 3.7 a lock based approach is shown which describes that functionality.

A mutex is used to guarantee that only one crypto API at a time is using the Netlink interface to exchange data with the daemon. The mutex is locked when a Crypto API function is called. In Figure 3.7 it is shown with an encryption call. A mutex can only be locked when it was previously unlocked. The process will block and sleep until the mutex becomes available for locking. When the mutex is locked a Netlink message is send to the daemon. The kernel module's crypto API callback function will then wait for a completion object to complete.

The daemon will parse the Netlink message, and in this example it will call the enclaves encryption routing with the plaintext data. The enclave returns the encrypted data to the daemon. The daemon will send a Netlink message, containing the encrypted data, to the kernel module were it will trigger the Netlink callback function.

The kernel module's Netlink callback function parses the message, copies the encrypted text to a global variable and completes the completion object. This releases the waiting encryption function. The encrypted data is copied to the destination given in the Crypto API call, the mutex is unlocked and the function returns.

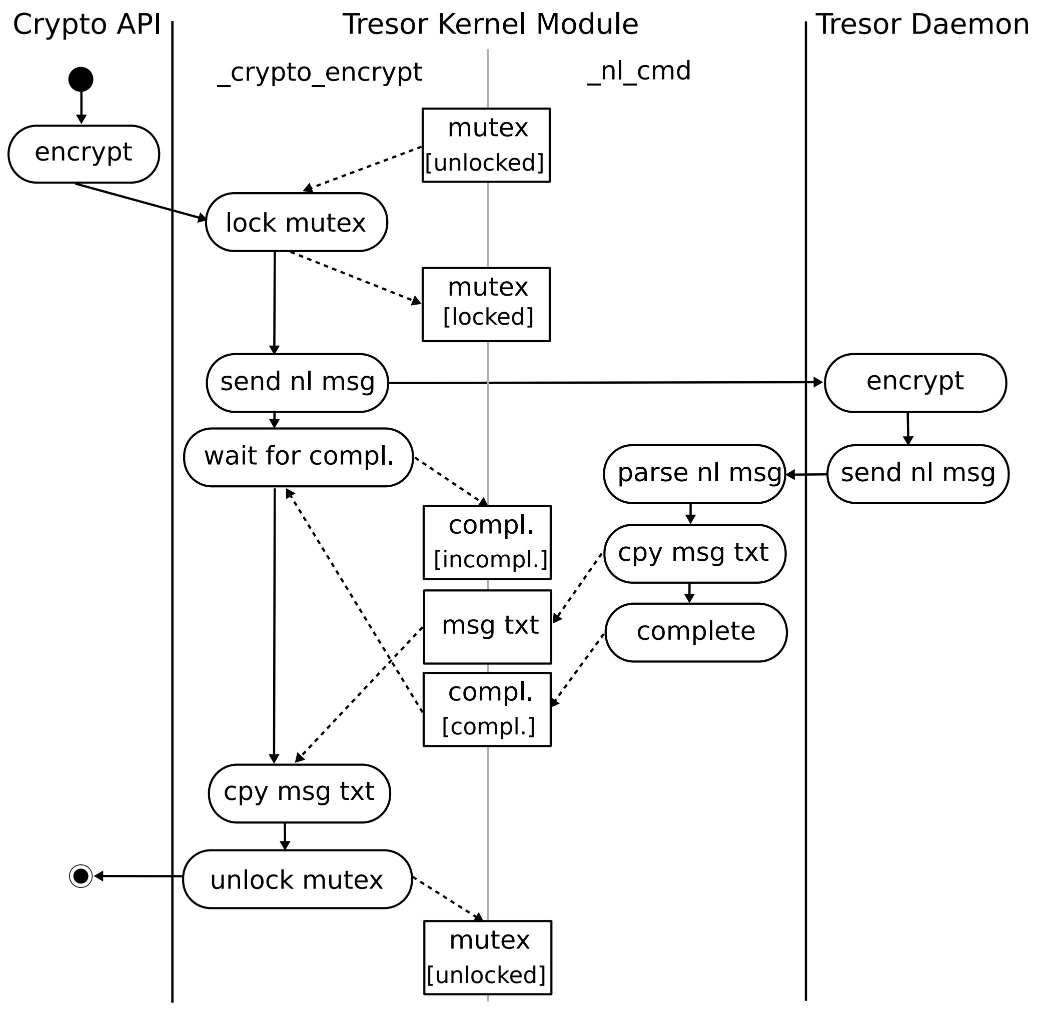

Figure 3.7.: Control flow inside the kernel module during a Crypto API encryption call

This flow guarantees that it is not possible to interrupt a single cryptographic routine via concurrent events.

Crypto API Testmgr

The Crypto API testmanager[9] can be used to perform tests on numerous crypto algorithms. To add a new cipher to the testmanager its information must be added to the `alg_test_desc` structure. The structure for *ecb mode tresor* can be seen in Listing 3.8.

As described earlier it is sufficient enough to just implement the block-cipher. The crypto API manages the logic behind the ECB mode and other modes. The defined vectors and counts are the same as the normal AES cipher, because Tresor depicts the normal AES cryptography.

[9]http://lxr.free-electrons.com/source/crypto/testmgr.c

```
.alg = "ecb(tresorsgx)",
.test = alg_test_skcipher,
.suite = {
        .cipher = {
                .enc = {
                        .vecs = aes_enc_tv_template,
                        .count = AES_ENC_TEST_VECTORS
                },
                .dec = {
                        .vecs = aes_dec_tv_template,
                        .count = AES_DEC_TEST_VECTORS
                }
        }
}
```

Figure 3.8.: Crypto API testmgr TresorSGX test definition

```
#include <linux/crypto.h>
..
alg_test("ecb(tresorsgx)","ecb(tresorsgx)",0,0);

struct crypto_cipher *tfm = NULL;
tfm = crypto_alloc_cipher("tresorsgx", 0, 16);
crypto_cipher_setkey(tfm, test_key_128, key_len)
crypto_cipher_encrypt_one(tfm, testResult, testVector);
```

Figure 3.9.: Tresor Test kernel module function calls to the Crypto API

To use the modified testmanager the kernel must be compiled and the system must boot the modified kernel. For Tresor functionality it is not required to modify the testmanager because it is only used for debugging and evaluation purposes. However, if Tresor is modified to use the sealed salt, the key from the Crypto API set key function will be altered with the salt. Therefore the testmanager will return an error because the encrypted vectors do not match the raw AES encrypted vectors.

Tresor Test LKM

To test the TresorSGX architecture a Linux kernel module was developed. It contains calls to execute the *testmgr* with the TresorSGX cipher as well as methods to call the cipher directly via the Crypto API as shown in Listing 3.9.

The test kernel module calls the test function with the algorithm defined in the testmgr declaration. This function returns either `null` or an error if the test failed. Furthermore, information to the kernel log are written if the test fails.

The Crypto API provides a transformation structure which is allocated with the *tresorsgx* ciphername, the algorithm type and cipher mask[10]. Then the key is set at the Crypto API and the transformation can be used for encryption and decryption. In the example in Listing 3.9 TresorSGX is used to encrypt raw blocks without any special modes like ECB or CBC.

3.3.3. Daemon

The daemon is the middleman between the kernel module and the enclave. The main tasks are the Netlink communication and the enclave management. The daemon is started by the kernel module with the help of the user mode helper API or manually by the user. During the initialisation phase the daemon creates the enclave with the help of the Intel SGX SDK. When the enclave creation has succeeded it sends a register message of type `TRESOR_NETLINK` to the Netlink bus. That initiates the register routine at the kernel module, which saves the daemons port id for later communication. The daemon waits in a loop for new Netlink messages which will be parsed and analysed. It exits the loop if an exit message is send by the kernel module or a signal interrupt is send.

Netlink Because of the improved reliability and usability the *Netlink Protocol Library Suite* is used to exchange Netlink messages. The library consists of multiple modules which handle the socket creation, message parsing, sending and receiving. Furthermore, it is possible to target network interfaces and route TCP/IP packets, but that was not the scope in this work. The Netlink library provides multiple additional features like sequence checks or auto-acknowledge messages which can be used to improve the stability of the protocol and for purposefull error handling. In this scenario these features were disabled, because they produce an overhead which reduces performance.

Enclave Management As described in the SGX technology section 2.1.5 a custom enclave can only be started with a launch key from the Intel launch enclave. In the SGX SDK section 2.1.6 the enclave building process is characterised which allows the execution of the custom enclave. Although, it is possible to execute the launch enclave, retrieve the custom launch key and launch the custom enclave without the help of the SDK tools[11], but that was not the scope of this work either. The *sgx_urts library* was used to create the enclave. This triggers the Intel SGX *AESM Service* which manages the architectural enclaves, for example the launch enclave which returns the launch key for Tresor enclave. By using the procedure described in section 2.1.5 the enclave is created and initiated. The daemon retrieves from the `sgx_create_enclave` function the enclave ID, the launch token and the information if the launch token was updated (if it was an valid token in the first place). The enclave functions are called by using the files generated by the *edger8r*.

[10]http://lxr.free-electrons.com/source/include/linux/crypto.h#L1440
[11]https://github.com/jethrogb/sgx-utils

When configuring TresorSGX to use the sealed cryptography the daemon loads a sealed file on a set key operation. It reads a file from a defined location and sends its content with the Crypto API key to the enclave. In the enclave that file is unsealed and the salt is used as input for the password based key derivation function. If the file cannot be read a null-file is send to the enclave, which generates a new salt. The sealed file must be saved when the enclave's set key function returns. If the seal file is lost or modified, is will not be possible to restore the sealed salt. That results in a new cryptographic key for the following operations, even if the user password is the same.

Set key by pipe To harden the security of the communication it is possible to set the key directly at the daemon using a FIFO file. The FIFO file is similar to a standard pipe, but not anonymous. If the daemon is configured to use the key setting by pipe it will open a pipe during its initialisation and wait for the key. The user of the cryptographic system must send the key to that opened named pipe. This can be done by `echo "password" » /tmp/tresorsetkey` or more securely via the setkey tool which hides the user input and clears all buffers.

3.3.4. Enclave

The enclave is the trusted computing base in this cryptographic architecture. Whereas by default Crypto API must trust the whole kernel that no one tries to extract the key from memory, the key is save in the enclave. The enclave provides access to 4 trusted functions. These functions can be called by anybody, not only the daemon. In this architecture the enclave provides 2 initialisation functions which allow the key setting, an encrypt and a decrypt function. These functions are trusted and with the help of the *edger8r* are files created which allow the function usage from the outside. The edger8r generates these files based on the *EDL* file shown in Listing 3.10.

The `enclInitCrypto` sets the key when sealed cryptography is disabled. Its parameters are the algorithm identifier (AES 128bit, 192bit, 256bit) and the key itself which must be `key_len` bytes long. The *edger8r* builds functions which check and copies these fields to save memory regions, which can be accessed by the enclave (for example, the pointer cannot point in another enclave area).

The `enclInitSealedCrypto` extends the normal initialisation with a buffer that contains the sealed file. The length of the buffer must be defined with `buf_len` for the input. If the enclave must generate a new seal, it copies it to the location of the buffer and writes length information to `seal_len`. However, the buffer must be at least the same size as the output seal. The user of the enclave must make sure to allocate enough memory for the buffer. The seal which contains as payload 3 32bit integers is 616bytes long.

The encryption and decryption functions have parameters for the input and output blocks as well as parameters for the block lengths. Untrusted functions are calls from the enclave to the host application. During the implementation some debugging functions were

```
enclave {
        trusted {
                public int enclInitCrypto(
                        char algorithm,
                        [in, size=key_len] unsigned char *key,
                        size_t key_len);
                public int enclInitSealedCrypto(
                        char algorithm,
                        [in, size=key_len] unsigned char *key,
                        int key_len,
                        [in,out, size=buf_len] unsigned char* buf,
                        int buf_len,
                        [out] int *seal_len);
                public void enclEncrypt(
                        [in, size=in_len] unsigned char * in,
                        size_t in_len,
                        [out, size=out_len] unsigned char  *out,
                        size_t out_len);
                public void enclDecrypt(
                        [in, size=in_len] unsigned char * in,
                        size_t in_len,
                        [out, size=out_len] unsigned char  *out,
                        size_t out_len);
        };

        untrusted {};
};
```

Figure 3.10.: Tresor enclave EDL file for edge interface creation.

usefull, but they are not essential in the final version.

AESNI As summarised in Section 3.2.3 it was not possible to include a fully featured cryptographic library like *OpenSSL* or *mbetTLS* in the enclave. The limitations of SGX are to strict and a modification of an entire crypto library is out of scope for this work. However, it was possible to link against the compiled assembly of the official Intel *AESNI* library which provides AES encryption and decryption for different key sizes. The usage of the AESNI library is straight forward and works flawless.

PBKDF2 To add some sort of 2-factor authentication the sealing functionality of SGX was used. As described in the SGX sealing Section 2.1.5 it is possible to encrypt data with the identity of the enclave or the identity of the enclave signer. Sealing with the enclave signer is a good model if someone expects the enclave to change because of updates or if multiple enclaves should be able to decrypt the data. TresorSGX uses the enclave identity sealing to guarantee that only this one enclave on this machine is able to decrypt the sealed data. In that sealed data is a salt and configuration parameters are saved. The salt is generated in the enclave using a sgx crypto function.

The *Password Based Key Derivation Function 2 (PBKDF2)*[36] applies a Hash-based message authentication code (HMAC) to the password (key) and the salt in the enclave. The HMAC is iterated many times to harden the pseudo random function against brute-force attacks. Neither the HMAC nor the PBKDF2 algorithm are included in an Intel SGX SDK library. For TresorSGX algorithms from a public domain project were used[12].

To generate the same encryption key with the same salt and password the number of iterations must also be the same. Therefore, the iterations are also sealed with the salt for later use. The iterations of the pseudo random function in TresorSGX are defined to 50.000 which can be modified before compiling.

Tresor Tester A host testing application is used to provide quick test and debug cycles. It implements the same functionality as the daemon and therefore allows direct debugging via the console. It also includes testvectors to analyse the correctness of the encryption and decryption functionality.

3.3.5. Usage

The usage of TresorSGX should be as easy as possible. Difficult cryptography software will be circumvented by an user because the additional effort will not result in visible value. Therefore, a cryptographic system must be embedded as ubiquitous as possible.

[12]https://github.com/ctz/sgx-pwenclave

Setup TresorSGX TresorSGX consists of multiple components which must be compiled and configured. A standard configuration is set in the `tresorcommon.h`. This configuration should be adequate for most users. For increased usability the set-key-by-pipe is disabled by default. This allows the usage of TresorSGX without further user interaction with the cryptographic system. In the default configuration sealing is activated. For increased security it is highly recommended to change the path of the seal to some removable or mounted network storage. This guarantees the confidentiality of the encrypted data even if an attacker is able to recover the user-key and the computer itself.

The following steps are needed to install TresorSGX:

1. (optional) modify the configuration of TresorSGX
2. (mandatory) build the TresorSGX LKM, Daemon, Enclave
3. (optional) copy files to the location defined in the configuration
4. (optional) add the TresorSGX LKM to `/etc/modules` to load the module on system boot
5. (mandatory) load the TresorSGX LKM into the kernel
6. (optional) build and load the Test TresorSGX LKM to test the cryptographic system

Setup Encrypted Partition To use TresorSGX to encrypt a whole partition the commandline interface *cryptsetup* for *dm-crypt*[13] is used. A device mapper provides a virtual layer on top of block devices. The device mapper dm-crypt provides an encryption layer which uses the Crypto API. All writes to the block device through this layer will be encrypted and the reads will be decrypted with a symmetric cipher.

Such symmetric cipher is TresorSGX which is registered in the Crypto API after initialisation. To use dm-crypt its kernel module must be loaded with `modprobe dm_mod`. The following line will setup the cryptographic cipher *tresorsgx* for the partition `/dev/sdb1` with a keysize of 128bit.

```
cryptsetup create tresor /dev/sdb1 –cipher tresorsgx –key-size
128
```

The user is asked to insert a password by cryptsetup. This password will be hashed by the cryptsetup and then send to the Crypto API set key function. This function forwards the key to the registered set key function by the TresorSGX LKM.

The key is retrievable from RAM during this time, for additional security the key set by pipe tool should be used. In that case the user-key must be set during the launch of the daemon. The key which is passed to the cryptsetup will be discharged in that case.

The cryptsetup maps the layer to the directory `/dev/mapper/tresor`. This layer works like the raw `/dev/sdb1`. When mapping the encrypted partition the first time a filesystem must be initialised:

[13] https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt

```
mkfs.ext2 /dev/mapper/tresor
```

Afterwards the layer can be mounted with:

```
mount /dev/mapper/tresor /media/tresor/
```

All writes to `/media/tresor` will be encrypted and all reads will be decrypted using TresorSGX. When using the sealed cryptography it is only possible to successfully use the partition when the seal is loaded into the enclave, the user-key is set and the enclave is running on the same machine which were used to create the seal in the first place.

4 Evaluation

The evaluation of the implemented TresorSGX architecture consists of an usage, correctness, performance and security analysis. The correctness of the AES cryptography of TresorSGX is verfied with standard tools and custom test cases. The performance analysis compares TresorSGX to standard Linux AES cryptography and direct unencrypted storage access. The security analysis gives an overview about possible threats and vulnerabilities.

4.1. TresorSGX usage

The Tresor enclave is accessed via the interface build by the Intel SGX SDK library. This helper functions provide the *edge* functions to enter the enclave. As described in Section 2.1.6 these interfaces are build by the *edger8r* Tool. The generated interfaces are open for examination. The edger8r Tool fulfilled its use in the development of TresorSGX without any restriction. It was not necessary to modify the generated interfaces at any time.

The Tresor enclave can be used for other applications than the Tresor daemon, too. It provides a trusted base for encryption and decryption operations. By using the standard interfaces, which were modelled following existing cryptographic libraries, the enclave can be included without much modification.

The usage of the daemon over the Netlink bus is reserved for the daemon at this state of SGX. Nevertheless, it is possible to modify the daemon for exchanging data with other user-space applications, like e.g. the Intel AESM Service.

The Tresor kernel module registers the TresorSGX cipher at the Crypto API. That allows the usage by different components of the Linux kernel and user space. A very user friendly approach is to setup a new partition, using cryptsetup and the TresorSGX cipher. The cryptsetup will map a device to the physical partition. When accessing the created device the data will be encrypted on the fly with TresorSGX. To secure the user-key additionally it is recommended to set the key by using the pipe with help of a small key-set tool. The security characteristics are discussed in Section 4.4.

4.2. Correctness

The correctness of the TresorSGX architecture regarding the encryption and decryption can be tested with different tools.

The Crypto API *testmgr*(3.3.2) is implemented in kernel space and uses the Crypto API. It initiates the crypto cipher and sets the key using the Crypto API. It iterates different keysizes and blocksizes during the test. Encrypted and decrypted data is compared to predefined testvectors. If all tests pass the testmgr returns zero, otherwise it will print an error message to the syslog and returns an error code. To use the testmgr with TresorSGX the kernel must be recompiled with the patched testmgr. The testmgr is disabled in the default kernel configuration, so it must be enabled. The testmsg is called in kernel space with `ret = al_test("ecb(tresorsgx)","ecb(tresorsgx)",0,0);`.

However the testmsg returns only zero if the encrypted data is equal to the predefined testvectors. When using the PBKDF2 with the key and the salt as input the data will be encrypted with another key than the key which was send by the Crypto API. The testmgr will return an error in that case. To test the cryptography despite the different keys, the enclave was modified to print the PBKDF2 derived encryption key to the host application. This key was used to decrypt without PBKDF2 the prior PBKDF2 encrypted data which worked successfully.

The same test model applies to the scenario where the key is set by the pipe. As in the original Tresor the Crypto API key works only as dummy key and has no functionality in that case. The testmgr can only be used if the Crypto API key on standard AES encryption is used.

It was possible to decrypt and encrypt partitions over multiple days so TresorSGX is capable of securing long term storage which is bound to the same CPU, seal, enclave and user-key.

Test	Plain	AES	TresorSGX
dd 100mb block write	107	104.5	1.1
hdparm uncached read	110.14	113.7	1.125
hdparm cached read	13289.53	12004.325	1576.69

Table 4.1.: Performance tests of TresorSGX in MB/s

4.3. Performance

TresorSGX was compared to the standard AES implementation in the Linux Kernel and the plain access to the storage system. The testing platform was a *2015 Dell Inspiron 7559* with an *Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz*, *16GiB System memory* and a *Seagate ST1000LM024 HN-M* hard drive. The operating system was an *Ubuntu 15.10* running the Linux kernel version *4.4.7*.

The performance of the original Tresor[46] is close to the standard AES implementation. When designing TresorSGX it became clear that the multi layered approach will result in many context changes which decreases the performance. The tests proved this assumption. Table 4.1 shows the results of three different tests in MB/s. The test script and the full results can be found in Appendix B.

Performance Tests Three different partitions were mounted on the same hard disk for the evaluation. 24 tests were executed before calculating the median of the results. The Linux *dd*[1] tool was used to analyse the write performance and *hdparm*[2] for measuring the read performance.

dd was executed with the parameters:

```
dd if=/dev/zero of=/media/$NAME/tempfile bs=100M count=1
conv=fdatasync,notrunc
```

That results in a physical not truncated write of one file with the size of 104857600 bytes. dd returns the time and the average write speed for that operation.

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

TresorSGX achieves about 1% of the read / write performance on disk and about 10% the read performance from the buffer cache. To write one block encrypted on disk the system must send the data block over the Netlink bus to the daemon. The daemon must enter the enclave, which is another context switch. The enclave encrypts the block using the AESNI instructions. The enclave returns the encrypted block, which is send over the

[1]http://linux.die.net/man/1/dd
[2]http://linux.die.net/man/8/hdparm

Netlink bus back to the Tresor kernel module. However it was not the scope to achieve the same performance like the original Tresor. TresorSGX works as proof of concept and its performance can be improved in future work.

Performance Improvements Other researchers compared the performance of the Linux Netlink to other communication channels like *SYS V Message Queues* and *SYS V Shared Memory*[56]. Compared to the SYS V Message Queues is the Netlink user-to-kernel communication 30% , kernel-to-user communication 55% slower and the startup time 66% slower in terms of processor cycles. SYS V shared memory has an even higher throughput but a longer startup time than the message queues.

In another test the *SYS V IPC*, *UNIX pipe* and *UNIX sockets* were compared[3]. The best latency and throughput were achieved using the UNIX pipe. However increasing the blocksize of the transmitted data lead to better results of the UNIX socket. It heavily depends on the individual configuration. The single block cryptography sends very small chunks of data over the Netlink, increasing the blocksize will result in a better throughput of the system. Alternatively it is possible to exchange the Netlink interface with a better suited communication channel.

4.4. Security

The security properties of the original Tresor[46] were guaranteed by saving the encryption key securely in the CPU debug registers. The Linux kernel is patched to secure these registers from unauthorized access. However, by exploiting the IDT Blass and Robertson [4] were able to recover the saved encryption key using DMA attacks. TresorSGX must guarantee at least the same security principles as Tresor.

In TresorSGX the encryption key is only generated inside the enclave. At no point in the lifecycle the encryption key is known to the outside world. The sealed salt is generated randomly and also unknown to any other component than the enclave. With this architecture it is not possible to recover the encryption key.

However, it is possible to execute attacks on TresorSGX which allows the decryption of encrypted data. An attacker must obtain the following components of the cryptographic system:

- user key / password
- sealed salt
- Tresor Enclave which sealed the salt
- CPU which sealed the salt

[3]https://sites.google.com/site/rikkus/sysv-ipc-vs-unix-pipes-vs-unix-sockets

The user-key and the sealed salt are used as Input for the PBKDF2 to generate the encryption key. Furthermore, an attacker must be able to execute the Tresor enclave on exactly that CPU, where the salt was sealed. That hardens the encryption because it is not sufficient to know the user key on its own (as in the original Tresor or other cryptographic systems). TresorSGX is therefore a multi-factor cryptographic system.

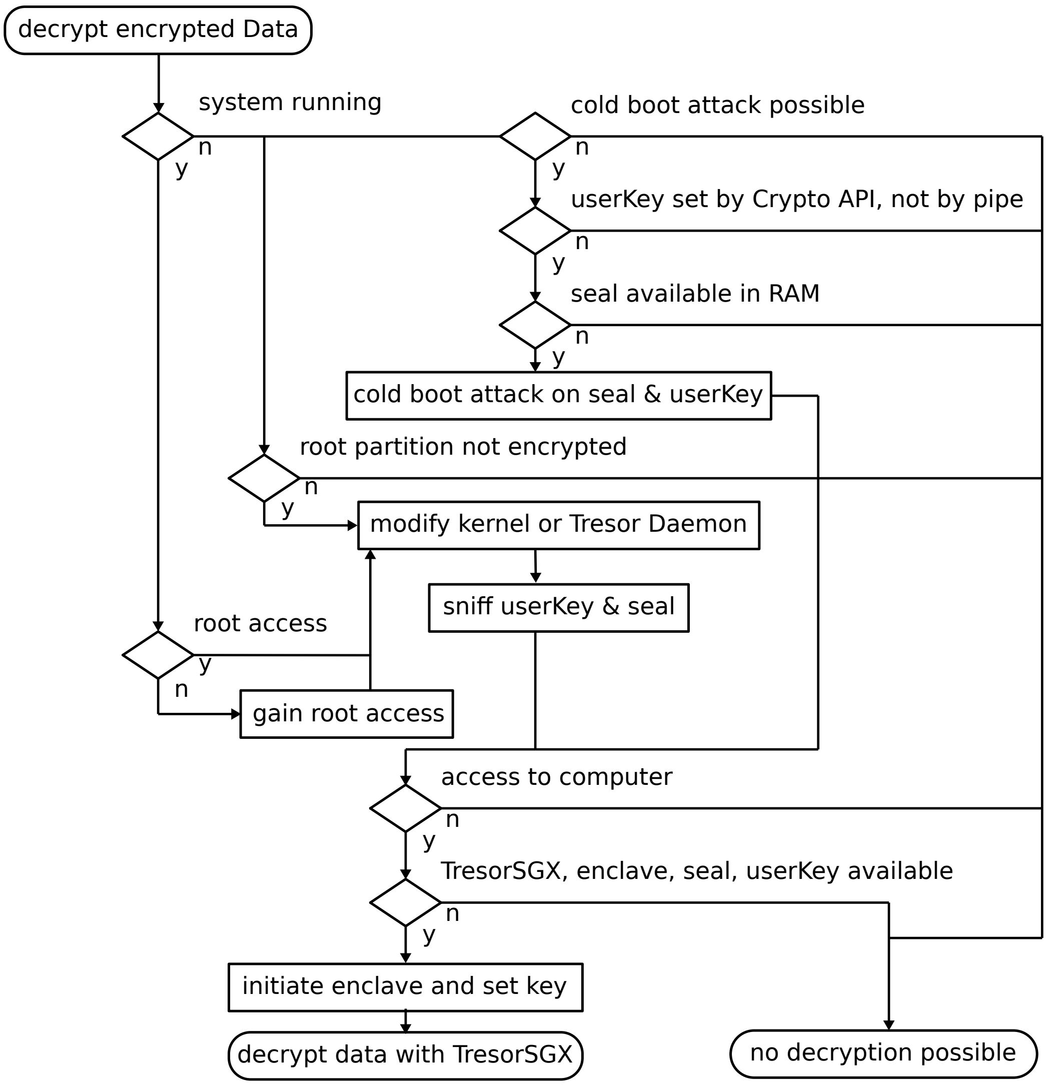
decrypt encrypted Data
system running
n
y
cold boot attack possible
n
y
userKey set by Crypto API, not by pipe
n
y
seal available in RAM
n
y
cold boot attack on seal & userKey
root partition not encrypted
n
y
modify kernel or Tresor Daemon
sniff userKey & seal
root access
y
n
gain root access
access to computer
n
y
TresorSGX, enclave, seal, userKey available
n
y
initiate enclave and set key
decrypt data with TresorSGX
no decryption possible

Figure 4.1.: Flowchart of an attack to decrypt encrypted data.

In Figure 4.1 a possible attack flow is shown. The goal is to encrypt TresorSGX encrypted data. It is assumed that TresorSGX is currently not running and the encrypted partition is not mounted. If the system is not running a cold boot attack could be carried out. The *Linux Memory Extractor (LiME)*[4] was used to analyse the full RAM of the testsystem

[4]https://github.com/504ensicsLabs/LiME

after setting the user-key. When setting the key with the tool *cryptsetup* to initialise an encrypted partition, the user key can be retrieved from multiple locations in RAM.

The original Tresor mitigated this problem by using the *sysfs* virtual file system, to directly send messages to the kernel module. In TresorSGX the daemon is used to send keys to the enclave. A pipe model was used to secure the user key. The user key which was send using the pipe to the daemon could not be retrieved with LiME.

Another component which could be retrieved from RAM is the sealed salt. An enclave can not read a file directly, therefore the daemon must load the file and pass it to the enclave. During that time the sealed data is in unprotected memory and can be retrieved. To increase the security of TresorSGX it is recommended to save the sealed salt on a removable storage which is only connected to the computer when TresorSGX is used.

Another threat model is the modification of the Linux kernel and user applications. If an attacker is able to obtain root rights, it is easy to modify the untrusted parts of TresorSGX. This allows the sniffing of the user-key and the sealed salt. When an attacker has obtained these two components he is able to initialise TresorSGX with the credentials. The access to the CPU and the enclave (which were used to seal the salt must be given). Otherwise it is not possible to unseal the salt.

5 Conclusion and Future Work

In this final chapter the findings are summarised and a conclusion is drawn. During the analysis of Intel SGX and the design / implementation of TresorSGX additional SGX documentation, tools, examples were released. The SGX SDK was made available in January 2016, which allowed the usage of SGX on real hardware.

5.1. SGX Development

The Intel Software Guard Extensions require much more development effort than normal applications because of the strict SGX security properties. As described in section 2.1.10 different limitations which prohibit a direct inclusion of libraries exist. The libraries must not execute operations which can be used to weaken the security of enclave or the host operating system / virtual machine.

The limitations due to the restricted operations and functions were quickly reached during the implementation of TresorSGX. It was neither possible to use the OpenSSL library nor the more modular mbedTLS library. It would ease the development of secure enclaves if fully featured SGX conform cryptographic libraries were available. 40% of the TresorSGX development time were donated to analyse, validate and implement different cryptographic libraries in the enclave.

The main advantage of SGX is that it allows to exclude a defined part of an application into a secure enclave which is save from modification, observation and can be remote attestated. The main use-case is to create trust with the small TCB of the enclave for executions on possible malicious hosts. However, side channel (see section 2.1.9) attacks, which modify the operating system or the host application, are able to recover confidential data. The enclave developer must be aware of the trade-off that is accompanied with a small TCB and large untrusted host application.

The Intel SGX SDK (see section 2.1.6) contains the required Launch enclave as well as additional libraries and enclave building tools. Although these tools and libraries are not required to execute an enclave (see section 2.1.7 for more information) they increase the development speed by generating edge proxies between the enclave and the host application, as well as checking the enclave for errors or security risks during the signing process. However, the Intel Launch enclave is still mandatory for initialising a custom enclave.

The requirements for obtaining a production license, described in section 2.1.11, demand a complex and secure development environment and key management. Furthermore, an enclave developer must create a distribution system which also distributes the Intel SGX platform software beside the actual application. It remains to be seen if proposed use-cases in papers and patents(2.1.8) will be brought the the consumers under these conditions.

5.2. SGX Enabled Tresor

The scope of this thesis was to provide a Linux kernel functionality in an Intel SGX enclave. As a proof of concept the cold-boot resistant cryptographic system Tresor by Müller et al. [46] in a SGX context was developed. The kernel Crypto API is able to perform cryptographic operations using this new cipher. The security guarantees of the original Tresor are kept. Furthermore the encryption is hardened by multiple factors (CPU, enclave, seal) which must be present to generate the same encryption key again. Also it is not possible to gain access to the encryption key or the salt if the enclave is not modified.

In the early phase of this system should be an example for the isolation of Linux kernel components completely in kernel space. However, as described in the SGX lifecycle section 2.1.5 it is by design of SGX not possible to enter an enclave from kernel space.

Therefore, the three layered architecture, proposed in section 3.3 is introduced. Like the original Intel SGX SDK a kernel module is used for the privileged instructions and for registering the cipher at the Crypto API. A daemon is used to manage the enclave and call trusted enclave function. The communication between the kernel module and the daemon is achieved by using the Netlink interface. Netlink provides bidirectional messaging but suffers from a low message throughput as evaluated in section 4.3.

By using sealing techniques to save a salt which can be used in password based key derivation function (PBKDF2), it is possible to generate an encryption key which only exists in the enclave. This key cannot be optained or generated by the outside world. With

TresorSGX encrypted data can only be decrypted if the user inserts its user password to the enclave and the enclave unseals successfull the salt. The salt is sealed using the identity of the enclave. That means that a modified version of the enclave is not able to unseal the data. Furthermore, be the enclave must executed on the same CPU as the data was sealed because the CPU specific key is also used for sealing.

In the original Tresor it was possible to obtain the key by using sophisticated attacks like *TRESOR-HUNT* by Blass and Robertson [4] or modifying the Linux kernel directly. With TresorSGX such an attack would not lead to the extraction of the encryption keys, because these are not known to the untrusted operating system.

5.3. Isolation of OS Components with SGX

As previously described it is not possible to execute the enclaves in kernel space. Therefore, the main approach of isolating operating system components with the Intel Software Guard Extensions is not viable in kernel space. The approach must be adapted to move functions worth protecting into an enclave running in user space.

Today no other approach is known to use SGX to secure kernel components in user space enclaves. The TresorSGX model is a first approach which analyses the conditions, difficulties and consequences of moving a kernel component into an user space enclave. Although the security of the AES cipher was increased by providing in the TresorSGX architecture, the usability and performance suffered.

5.4. Future Work

The Intel Software Guard Extensions were only recently made available to the research community. That opens different research topics. As described in section 2.1.8 multiple use-cases for SGX exist. Companies declared patents for shielded execution techniques, policy based configuration, identification, attestation and digital rights management purposes. These fields of application imply chances for increased security as well as reduced freedom for the device owners. It has to be decided how far a company is allowed to go to secure its content and devices against the user who paid for these devices or services.

However SGX includes other technical challenges which can be addressed by researchers. As described the enclave system is very limited regarding available C functions. Libraries like OpenSSL and mbedTLS cannot be executed because they are calling unsupported functions or syscalls which are not allowed. It would ease the development of security related tools if a full featured cryptographic library can be used in enclave space.

To port existing libraries into the enclave, a layer around unsupported c functions would be helpfull. The Intel SGX SDK c library could be extended with implementations for these unsupported functions. However some functionality which cannot be wrapped and emulated (e.g. `fork`) will not be possible because of the design of the SGX enclave.

TresorSGX security characteristics are superior to these of the standard cryptographic modules. The system could be improved regarding its performance to be on a par with the original Tresor implementation or other encryption ciphers. With the current implementation it is not possible to encrypt a complete system with TresorSGX because the daemon must be executed in user space. Future work could address this problem and provide solutions where the SGX enabled encryption is used for the complete Linux system.

Appendices

A BACKGROUND

A.1. Intel SGX

```
...
#include "sgx_edger8r.h" /* for sgx_satus_t etc. */
#include <stdlib.h> /* for size_t */

sgx_status_t ecall_changeBuf(
        sgx_enclave_id_t eid,
                char* buf,
                size_t len);
...
```

Figure A.1.: enclave_u.h untrusted proxy declaration

```
#include "enclave_u.h"

typedef struct ms_ecall_changeBuf_t {
        char* ms_buf;
        size_t ms_len;
} ms_ecall_changeBuf_t;

sgx_status_t ecall_changeBuf(sgx_enclave_id_t eid, char* buf,size_t len)
{
        sgx_status_t status;
        ms_ecall_changeBuf_t ms;
        ms.ms_buf = buf;
        ms.ms_len = len;
        status = sgx_ecall(eid, 0, &ocall_table_enclave, &ms);
        return status;
}
```

Figure A.2.: enclave_u.c untrusted proxy definitions

```
#include "sgx_edger8r.h" /* for sgx_ocall etc. */
...
void ecall_changeBuf(char* buf, size_t len);
...
```

Figure A.3.: enclave_t.h trusted proxy declaration

```
/* sgx_ocfree() just restores the original outside stack pointer. */
#define OCALLOC(val, type, len) do {     \
        void* __tmp = sgx_ocalloc(len); \
        if (__tmp == NULL) {     \
                sgx_ocfree();    \
                return SGX_ERROR_UNEXPECTED;\
        }                        \
        (val) = (type)__tmp;     \
} while (0)

typedef struct ms_ecall_changeBuf_t {
        char* ms_buf;
        size_t ms_len;
} ms_ecall_changeBuf_t;

static sgx_status_t SGX_CDECL sgx_ecall_changeBuf(void* pms)
{
        ms_ecall_changeBuf_t* ms = SGX_CAST(ms_ecall_changeBuf_t*, pms);
        sgx_status_t status = SGX_SUCCESS;
        char* _tmp_buf = ms->ms_buf;
        size_t _tmp_len = ms->ms_len;
        size_t _len_buf = _tmp_len;
        char* _in_buf = NULL;

        CHECK_REF_POINTER(pms, sizeof(ms_ecall_changeBuf_t));
        CHECK_UNIQUE_POINTER(_tmp_buf, _len_buf);

        if (_tmp_buf != NULL) {
                _in_buf = (char*)malloc(_len_buf);
                if (_in_buf == NULL) {
                        status = SGX_ERROR_OUT_OF_MEMORY;
                        goto err;
                }

                memcpy(_in_buf, _tmp_buf, _len_buf);
        }
        ecall_changeBuf(_in_buf, _tmp_len);
err:
        if (_in_buf) {
                memcpy(_tmp_buf, _in_buf, _len_buf);
                free(_in_buf);
        }

        return status;
}

SGX_EXTERNC const struct {
        size_t nr_ecall;
        struct {void* ecall_addr; uint8_t is_priv;} ecall_table[1];
} g_ecall_table = {
        1,
        {
                {(void*)(uintptr_t)sgx_ecall_changeBuf, 0},
        }
};
```

Figure A.4.: enclave_t.c trusted proxy definitions

B EVALUATION

B.1. Performance test

```
#!/bin/bash
NAME=testtresor

for i in `seq 1 24`;
do
        sudo dd if=/dev/zero of=/media/$NAME/tempfile bs=100M \
        count=1 conv=fdatasync,notrunc 2>&1 | tail -n 1
done

for i in `seq 1 24`;
do
        sudo hdparm -t /dev/mapper/$NAME | tail -n 1
done

for i in `seq 1 24`;
do
        sudo hdparm -T /dev/mapper/$NAME | tail -n 1
done
```

Figure B.1.: performance test script

```
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.13 MB/sec
Timing buffered disk reads: 330 MB in  3.01 seconds = 109.67 MB/sec
Timing buffered disk reads: 330 MB in  3.01 seconds = 109.66 MB/sec
Timing buffered disk reads: 332 MB in  3.02 seconds = 110.11 MB/sec
Timing buffered disk reads: 332 MB in  3.02 seconds = 110.12 MB/sec
Timing buffered disk reads: 332 MB in  3.02 seconds = 110.11 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.12 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.14 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.13 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.13 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.13 MB/sec
Timing buffered disk reads: 328 MB in  3.00 seconds = 109.33 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.16 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.14 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.14 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.16 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.15 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.16 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.17 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.16 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.16 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.17 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.18 MB/sec
Timing buffered disk reads: 332 MB in  3.01 seconds = 110.18 MB/sec
```

Figure B.2.: plain - hdparm -t uncached read

```
Timing cached reads:   25644 MB in  2.00 seconds = 12836.07 MB/sec
Timing cached reads:   24894 MB in  2.00 seconds = 12460.96 MB/sec
Timing cached reads:   25600 MB in  2.00 seconds = 12814.92 MB/sec
Timing cached reads:   26192 MB in  2.00 seconds = 13112.28 MB/sec
Timing cached reads:   25402 MB in  2.00 seconds = 12715.69 MB/sec
Timing cached reads:   27232 MB in  2.00 seconds = 13633.27 MB/sec
Timing cached reads:   25358 MB in  2.00 seconds = 12702.80 MB/sec
Timing cached reads:   22482 MB in  2.00 seconds = 11252.19 MB/sec
Timing cached reads:   27932 MB in  2.00 seconds = 13983.15 MB/sec
Timing cached reads:   26548 MB in  2.00 seconds = 13289.36 MB/sec
Timing cached reads:   27696 MB in  2.00 seconds = 13864.80 MB/sec
Timing cached reads:   26278 MB in  2.00 seconds = 13154.02 MB/sec
Timing cached reads:   27136 MB in  2.00 seconds = 13589.09 MB/sec
Timing cached reads:   26708 MB in  2.00 seconds = 13369.61 MB/sec
Timing cached reads:   28104 MB in  2.00 seconds = 14070.00 MB/sec
Timing cached reads:   27264 MB in  2.00 seconds = 13648.28 MB/sec
Timing cached reads:   27826 MB in  2.00 seconds = 13929.95 MB/sec
Timing cached reads:   24870 MB in  2.00 seconds = 12448.85 MB/sec
Timing cached reads:   26548 MB in  2.00 seconds = 13289.70 MB/sec
Timing cached reads:   26450 MB in  2.00 seconds = 13240.17 MB/sec
Timing cached reads:   26514 MB in  2.00 seconds = 13271.94 MB/sec
Timing cached reads:   26736 MB in  2.00 seconds = 13383.77 MB/sec
Timing cached reads:   26660 MB in  2.00 seconds = 13345.51 MB/sec
Timing cached reads:   26938 MB in  2.00 seconds = 13485.47 MB/sec
```

Figure B.3.: plain - hdparm -T cached read

```
104857600 bytes (105 MB) copied, 1,18115 s, 88,8 MB/s
104857600 bytes (105 MB) copied, 1,02872 s, 102 MB/s
104857600 bytes (105 MB) copied, 0,966114 s, 109 MB/s
104857600 bytes (105 MB) copied, 0,965683 s, 109 MB/s
104857600 bytes (105 MB) copied, 0,966563 s, 108 MB/s
104857600 bytes (105 MB) copied, 0,997715 s, 105 MB/s
104857600 bytes (105 MB) copied, 0,989554 s, 106 MB/s
104857600 bytes (105 MB) copied, 0,981488 s, 107 MB/s
104857600 bytes (105 MB) copied, 0,994705 s, 105 MB/s
104857600 bytes (105 MB) copied, 0,954451 s, 110 MB/s
104857600 bytes (105 MB) copied, 0,995095 s, 105 MB/s
104857600 bytes (105 MB) copied, 1,02202 s, 103 MB/s
104857600 bytes (105 MB) copied, 1,1182 s, 93,8 MB/s
104857600 bytes (105 MB) copied, 1,03035 s, 102 MB/s
104857600 bytes (105 MB) copied, 1,0205 s, 103 MB/s
104857600 bytes (105 MB) copied, 1,19903 s, 87,5 MB/s
104857600 bytes (105 MB) copied, 1,33448 s, 78,6 MB/s
104857600 bytes (105 MB) copied, 0,985027 s, 106 MB/s
104857600 bytes (105 MB) copied, 1,00159 s, 105 MB/s
104857600 bytes (105 MB) copied, 1,0296 s, 102 MB/s
104857600 bytes (105 MB) copied, 0,987914 s, 106 MB/s
104857600 bytes (105 MB) copied, 1,03266 s, 102 MB/s
104857600 bytes (105 MB) copied, 1,01218 s, 104 MB/s
104857600 bytes (105 MB) copied, 1,01608 s, 103 MB/s
```

Figure B.4.: AES - dd test 100M block

```
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.66 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.65 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.66 MB/sec
 Timing buffered disk reads: 340 MB in  3.01 seconds = 112.80 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.67 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.68 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.68 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.69 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.71 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.68 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.69 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.70 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.71 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.70 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.72 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.70 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.71 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.71 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.72 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.74 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.72 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.74 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.74 MB/sec
 Timing buffered disk reads: 342 MB in  3.01 seconds = 113.74 MB/sec
```

Figure B.5.: AES - hdparm -t uncached read

```
Timing cached reads:   21870 MB in  2.00 seconds = 10946.90 MB/sec
Timing cached reads:   22282 MB in  2.00 seconds = 11151.74 MB/sec
Timing cached reads:   25484 MB in  2.00 seconds = 12759.09 MB/sec
Timing cached reads:   27594 MB in  2.00 seconds = 13814.17 MB/sec
Timing cached reads:   27638 MB in  2.00 seconds = 13837.99 MB/sec
Timing cached reads:   24220 MB in  2.00 seconds = 12123.35 MB/sec
Timing cached reads:   25230 MB in  2.00 seconds = 12628.97 MB/sec
Timing cached reads:   27330 MB in  2.00 seconds = 13682.14 MB/sec
Timing cached reads:   18246 MB in  2.00 seconds = 9131.30 MB/sec
Timing cached reads:   25460 MB in  2.00 seconds = 12744.31 MB/sec
Timing cached reads:   25398 MB in  2.00 seconds = 12714.44 MB/sec
Timing cached reads:   27814 MB in  2.00 seconds = 13924.96 MB/sec
Timing cached reads:   22722 MB in  2.00 seconds = 11372.56 MB/sec
Timing cached reads:   22362 MB in  2.00 seconds = 11192.19 MB/sec
Timing cached reads:   22248 MB in  2.00 seconds = 11138.35 MB/sec
Timing cached reads:   22702 MB in  2.00 seconds = 11365.54 MB/sec
Timing cached reads:   22588 MB in  2.00 seconds = 11309.10 MB/sec
Timing cached reads:   23740 MB in  2.00 seconds = 11885.30 MB/sec
Timing cached reads:   23426 MB in  2.00 seconds = 11725.43 MB/sec
Timing cached reads:   22756 MB in  2.00 seconds = 11393.50 MB/sec
Timing cached reads:   17574 MB in  2.00 seconds = 8799.93 MB/sec
Timing cached reads:   26858 MB in  2.00 seconds = 13448.88 MB/sec
Timing cached reads:   27634 MB in  2.00 seconds = 13834.35 MB/sec
Timing cached reads:   28184 MB in  2.00 seconds = 14110.11 MB/sec
```

Figure B.6.: AES - hdparm -T cached read

```
104857600 bytes (105 MB) copied, 92,3919 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,0696 s, 1,2 MB/s
104857600 bytes (105 MB) copied, 90,8594 s, 1,2 MB/s
104857600 bytes (105 MB) copied, 91,2528 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,1856 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 92,4903 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 92,0558 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,7607 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,2323 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,5721 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,2175 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,2368 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,4442 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,9309 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 90,7654 s, 1,2 MB/s
104857600 bytes (105 MB) copied, 90,7294 s, 1,2 MB/s
104857600 bytes (105 MB) copied, 91,1484 s, 1,2 MB/s
104857600 bytes (105 MB) copied, 91,3997 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,0346 s, 1,2 MB/s
104857600 bytes (105 MB) copied, 91,2326 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 91,1419 s, 1,2 MB/s
104857600 bytes (105 MB) copied, 91,3237 s, 1,1 MB/s
104857600 bytes (105 MB) copied, 90,6785 s, 1,2 MB/s
104857600 bytes (105 MB) copied, 91,6027 s, 1,1 MB/s
```

Figure B.7.: TresorSGX - dd test 100M block

```
Timing buffered disk reads:   4 MB in  3.41 seconds =   1.17 MB/sec
Timing buffered disk reads:   4 MB in  3.58 seconds =   1.12 MB/sec
Timing buffered disk reads:   4 MB in  3.57 seconds =   1.12 MB/sec
Timing buffered disk reads:   4 MB in  3.53 seconds =   1.13 MB/sec
Timing buffered disk reads:   4 MB in  3.71 seconds =   1.08 MB/sec
Timing buffered disk reads:   4 MB in  3.61 seconds =   1.11 MB/sec
Timing buffered disk reads:   4 MB in  3.58 seconds =   1.12 MB/sec
Timing buffered disk reads:   4 MB in  3.61 seconds =   1.11 MB/sec
Timing buffered disk reads:   4 MB in  3.52 seconds =   1.14 MB/sec
Timing buffered disk reads:   4 MB in  3.51 seconds =   1.14 MB/sec
Timing buffered disk reads:   4 MB in  3.68 seconds =   1.09 MB/sec
Timing buffered disk reads:   4 MB in  3.57 seconds =   1.12 MB/sec
Timing buffered disk reads:   4 MB in  3.46 seconds =   1.15 MB/sec
Timing buffered disk reads:   4 MB in  3.55 seconds =   1.13 MB/sec
Timing buffered disk reads:   4 MB in  3.60 seconds =   1.11 MB/sec
Timing buffered disk reads:   4 MB in  3.47 seconds =   1.15 MB/sec
Timing buffered disk reads:   4 MB in  3.51 seconds =   1.14 MB/sec
Timing buffered disk reads:   4 MB in  3.63 seconds =   1.10 MB/sec
Timing buffered disk reads:   4 MB in  3.48 seconds =   1.15 MB/sec
Timing buffered disk reads:   4 MB in  3.72 seconds =   1.08 MB/sec
Timing buffered disk reads:   4 MB in  3.57 seconds =   1.12 MB/sec
Timing buffered disk reads:   4 MB in  3.51 seconds =   1.14 MB/sec
Timing buffered disk reads:   4 MB in  3.52 seconds =   1.14 MB/sec
Timing buffered disk reads:   4 MB in  3.55 seconds =   1.13 MB/sec
```

Figure B.8.: TresorSGX - hdparm -t uncached read

```
Timing cached reads:   3542 MB in  2.00 seconds = 1771.19 MB/sec
Timing cached reads:   2056 MB in  2.00 seconds = 1028.09 MB/sec
Timing cached reads:   1890 MB in  2.00 seconds = 945.04 MB/sec
Timing cached reads:   2842 MB in  2.00 seconds = 1421.09 MB/sec
Timing cached reads:   3668 MB in  2.00 seconds = 1834.22 MB/sec
Timing cached reads:   3396 MB in  2.00 seconds = 1698.26 MB/sec
Timing cached reads:   3650 MB in  2.00 seconds = 1825.21 MB/sec
Timing cached reads:   3728 MB in  2.00 seconds = 1864.21 MB/sec
Timing cached reads:   3054 MB in  2.00 seconds = 1527.15 MB/sec
Timing cached reads:   3268 MB in  2.00 seconds = 1634.24 MB/sec
Timing cached reads:   1310 MB in  2.00 seconds = 655.00 MB/sec
Timing cached reads:   3446 MB in  2.00 seconds = 1723.21 MB/sec
Timing cached reads:   3008 MB in  2.00 seconds = 1504.13 MB/sec
Timing cached reads:   3090 MB in  2.00 seconds = 1545.20 MB/sec
Timing cached reads:   3328 MB in  2.00 seconds = 1664.23 MB/sec
Timing cached reads:   3204 MB in  2.00 seconds = 1602.15 MB/sec
Timing cached reads:   3168 MB in  2.00 seconds = 1584.13 MB/sec
Timing cached reads:   2618 MB in  2.00 seconds = 1309.13 MB/sec
Timing cached reads:   2808 MB in  2.00 seconds = 1404.13 MB/sec
Timing cached reads:   3314 MB in  2.00 seconds = 1657.19 MB/sec
Timing cached reads:   3024 MB in  2.00 seconds = 1512.10 MB/sec
Timing cached reads:   3138 MB in  2.00 seconds = 1569.25 MB/sec
Timing cached reads:   4088 MB in  2.00 seconds = 2044.29 MB/sec
Timing cached reads:   2164 MB in  2.00 seconds = 1082.08 MB/sec
```

Figure B.9.: TresorSGX - hdparm -T cached read

Bibliography

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013.

[2] ARM ARM. Security technology building a secure system using trustzone technology (white paper). *ARM Limited*, 2009.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[4] Erik-Oliver Blass and William Robertson. Tresor-hunt: attacking cpu-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 71–78. ACM, 2012.

[5] Benjamin Böck and Secure Business Austria. Firewire-based physical security attacks on windows 7, efs and bitlocker. *Secure Business Austria Research Lab*, 2009.

[6] Blu-ray Disc Pre-recorded Book. Advanced access content system (aacs). 2006.

[7] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing for hardware authentication and attestation. *International Journal of Information Privacy, Security and Integrity 2*, 1(1):3–33, 2011.

[8] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[9] Stephen Checkoway and Hovav Shacham. *Iago attacks: Why the system call api is a bad untrusted rpc interface*, volume 41. ACM, 2013.

[10] Manuel Costa, Felix Schuster, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, and Antony Ian Taylor Rowstron. Trusted execution within a distributed computing system, February 7 2014. US Patent App. 14/175,692.

[11] Victor Costan and Srinivas Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 20 16. http://eprint. iacr. org.

[12] William C Deleeuw and Ned M Smith. Establishing secure digital relationship using symbology, March 24 2016. US Patent 20,160,087,949.

[13] Loïc Duflot, Daniel Etiemble, and Olivier Grumelard. Using cpu system management mode to circumvent operating system security functions. *CanSecWest/core06*, 2006.

[14] Niels T Ferguson, Yevgeniy Anatolievich Samsonov, Kinshuman Kinshumann, Samartha Chandrashekar, John Anthony Messec, Mark Fishel Novak, Christopher Mccarron, Amitabh Prakash Tamhane, Qiang Wang, David Matthew Kruse, et al. Secure management of operations on protected virtual machines, November 5 2015. US Patent 20,150,319,160.

[15] KAIST Gatech. Opensgx, 2015. URL `https://github.com/sslab-gatech/opensgx`. https://github.com/sslab-gatech/opensgx.

[16] James Greene. Intel trusted execution technology. *Intel Technology Whitepaper*, 2012.

[17] Shay Gueron. A memory encryption engine suitable for general purpose processors.

[18] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. 2015.

[19] 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. *Communications of the ACM*, 52(5):91–98, 2009.

[20] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 11. ACM, 2013.

[21] Oded Horovitz, Stephen A Weis, Sahil Rihan, and Carl A Waldspurger. Method and system for providing secure system execution on hardware supporting secure application execution, September 25 2014. US Patent App. 14/497,111.

[22] David Hulton. Cardbus bus-mastering: 0wning the laptop. *Proceedings of ShmooCon*, 6, 2006.

[23] Galen Clyde Hunt and Mark Eugene Russinovich. Securely storing content within public clouds, 12/31/2015.

[24] Intel. Intel hardware-based security technologies for intelligent retail devices. *Intel Technology Whitepaper*, 2013. URL `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/white-papers/security-technologies-4th-gen-core-retail-paper.pdf`.

[25] Intel. *Intel Software Guard Extensions Programming Reference*, 329298-002us edition, October 2015.

[26] Intel. *Intel Software Guard Extensions*, 2015.

[27] Intel. *Intel Software Guard Extensions Evaluation SDK*. Intel Corporation, 2015.

[28] Intel. *Intel Software Guard Extensions Enclave Writer's Guide*. Intel Corporation, 2015.

[29] Intel. Product change notification 114074 - 00, 10 2015. URL `http://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf`. 6th Generation Intel ® Core TM i7 & i5 Desktop and Intel ® Xeon ® E3-1200 v5 Family Processors, PCN 114074-00, Product Design, S-Spec and MM Number Change.

[30] Intel. Intel® 64 and ia-32 architectures software developer's manual, 9 2015. Volume 3 (3A, 3B, 3C & 3D): System Programming Guide.

[31] Alex Ionescu. Intel sgx enclave support in windows 10 fall update (threshold 2), November 11 2015.

[32] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. Opensgx: An open platform for sgx research. In *Proceedings of the Network and Distributed System Security Symposium, San Diego, CA*, 2016.

[33] Moritz Jodeit and Martin Johns. Usb device drivers: A stepping stone into your kernel. In *Computer Network Defense (EC2ND), 2010 European Conference on*, pages 46–52. IEEE, 2010.

[34] Simon P Johnson, Uday R Savagaonkar, Vincent R Scarlata, Francis X McKeen, and Carlos V Rozas. Technique for supporting multiple secure enclaves, March 3 2015. US Patent 8,972,746.

[35] Burt Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. 2000.

[36] Burt Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. 2000.

[37] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. 2015.

[38] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. Hmac: Keyed-hashing for message authentication. 1997.

[39] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 409–420, 2014.

[40] Carsten Maartmann-Moe. Adventures with daisy in thunderbolt-dma-land: Hacking macs through the thunderbolt interface. *U Rl: http://www. breaknenter. org/2012/02/adventures-with-dai sy-in-thunderbolt-dma-land-hacking-macs-through-the-thunderbolt-interface*, 2012.

[41] Jason Martin and Matthew Hoekstra. Policy-based trusted inspection of rights managed content, December 3 2015. US Patent 20,150,347,768.

[42] Nikos Mavrogiannopoulos, Miloslav Trmač, and Bart Preneel. A linux kernel cryptographic framework: decoupling cryptographic keys from applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1435–1442. ACM, 2012.

[43] Francis X McKeen, Carlos V Rozas, Uday R Savagaonkar, Simon P Johnson, Vincent Scarlata, Michael A Goldsmith, Ernie Brickell, Jiang Tao Li, Howard C Herbert, Prashant Dewan, et al. Method and apparatus to provide secure application execution, July 21 2015. US Patent 9,087,200.

[44] 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 *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–1. ACM, 2013.

[45] Tilo Müller and Michael Spreitzenbarth. Frost. In *Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.

[46] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, pages 17–17, 2011.

[47] Igor Muttik. Identification of call participants, 2015.

[48] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1570–1581. ACM, 2015.

[49] Rajesh Poornachandran, Shahrokh Shahidzadeh, Sudeep Das, Vincent J. Zimmer, Sumant Vashisth, and Pramod Sharma. Premises-aware security and policy orchestration, 12/31/2015.

[50] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library os from the top down. *ACM SIGPLAN Notices*, 46(3): 291–304, 2011.

[51] Avi Priev, Avishay Sharaga, and Hormuzd Khosravi. Securely pairing computing devices, March 24 2016. US Patent 20,160,085,960.

[52] Sanjay Sawhney, Petros Efstathopoulos, and Daniel Marino. Systems and methods for deploying applications included in application containers, August 25 2015. US Patent 9,116,768.

[53] Seth Schoen. Trusted computing: Promise and risk. *Electronic Frontier Foundation*, 16:26, 2003.

[54] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE, 2015.

[55] Mark E Scott-nash, Scott H Robinson, Howard C Herbert, Geoffrey S Strongin, Stephen J Allen, Tobias M Kohlenberg, and Uttam K Sengupta. Sensor privacy mode, September 3 2015. US Patent 20,150,248,566.

[56] Ryan Slominski. *Fast User/Kernel Data Transfer*. PhD thesis, College of William & Mary, 2007.

[57] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.

[58] Shruti Tople and Prateek Saxena. On the trade-offs in oblivious execution techniques.

[59] Shruti Tople, Ayush Jain, and Prateek Saxena. Leveefs: Securing access to untrusted filesystems in enclaved execution. 2015.

[60] TCG TPM. Main part 1 design principles specification version 1.2, 2003.

[61] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[62] Rafal Wojtczuk and Joanna Rutkowska. Attacking smm memory via intel cpu cache poisoning. *Invisible Things Lab*, 2009.

[63] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. 2015.