Samuel Weiser

Secure I/O with Intel SGX

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Mario Werner

Assessor

Stefan Mangard

Institute of Applied Information Processing and Communications

Graz, April 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Acknowledgements

I would like to thank my supervisor
for his valuable input and support
in writing and shaping this thesis.

Furthermore I thank my family,
especially my parents and grandmothers,
for their continuous and unconditional support
during the last years in many different ways.

Finally, I thank my creator
for being able to study,
for having peace,
for being loved.

Abstract

Unlike mobile phones, modern smart phones are no longer designed for a specific, definite use case. They not only feature a fully-fledged general purpose CPU but are also equipped with a multitudinous and permanently increasing set of application software. It is extremely difficult to have such a big software stack completely free of bugs. Hence, smart phones nowadays face increased susceptibility to malware, comparable to notebooks and personal computers. To tackle this issue, recent work has put a lot of effort into building a secure execution environment where applications are protected by a completely isolated execution container. Thus, bugs in the big software stack do not directly affect the protected application.

However, secure execution alone is not enough. Many applications on a mobile device require secure interaction with the user. Consider a secure application requesting user passwords and credit card information or providing secure chat. No piece of untrusted code outside the secure application shall have access to user input and output. Without a method for secure I/O, secure execution technology might be of no value to the end user at all, since sensitive information cannot be securely communicated. Therefore, secure I/O is needed.

To achieve secure I/O, one can encrypt sensitive content between I/O device and secure application. Thus, no malicious software can sniff on user input/output. However, mobile devices such as a smart phone or notebook are typically restricted to use legacy I/O devices, which do not support encryption at all. Therefore, secure I/O has to be assisted by the secure execution technology.

Apart from ARM TrustZone, there is currently no appropriate secure execution technology available which addresses secure I/O with legacy I/O devices. Intel Software Guard Extensions (SGX), for example, is a major upcoming player in secure execution technology but completely lacks support for secure I/O. Due to its broad dissemination in the near future, we investigate on enabling secure I/O with Intel SGX. In a proof-of-concept, we supplement SGX with a security microkernel, called seL4. We use seL4 to transparently and exclusively bind an I/O device to an SGX-hardened application. This enables to do secure I/O from within that application, even in the presence of malware. By trusting the security kernel, we approve an increased Trusted Computing Base. This is acceptable since seL4 is both, small and verifiable. Finally, we discuss possible modifications to SGX hardware in order to support secure I/O by design. This would eliminate the need for a security kernel, which would be easier to integrate in existing software stacks.

Keywords: secure input, secure output, secure I/O, secure port I/O, mobile device, smart phone, notebook, secure chat, Intel Software Guard Extensions, SGX, secure execution, seL4, security microkernel, RefOS, OpenSGX

Kurzfassung

Im Gegensatz zu klassischen Mobiltelefonen sind Smartphones nicht mehr beschränkt auf einen bestimmten Anwendungsfall. Sie stellen nicht nur gehörig Rechenleistung zur Verfügung, sondern sind auch mit einer ständig wachsenden Fülle an Programmen ausgestattet. Es ist extrem schwer, umfangreiche Software frei von Programmierfehlern zu halten. Daher sind Smartphones heutzutage ebenso von einer erhöhten Anfälligkeit für Schadsoftware betroffen wie Notebooks und der Personal Computer. Um dem zu begegnen, können sichere Ausführungsumgebungen verwendet werden, in denen Anwendungen durch einen komplett isolierten Container geschützt werden. Damit können Sicherheitslücken in fremder Software das geschützte Programm nicht mehr gefährden.

Sichere Ausführung ist jedoch nicht genug. Auf mobilen Endgeräten benötigen viele Anwendungen die Möglichkeit für sichere Interaktion mit dem Benutzer, um zum Beispiel Passwörter oder Kreditkarteninformationen abzufragen oder sichere Kommunikation mit anderen Benutzern zu ermöglichen. Keine andere Software darf Zugriff auf sensible Benutzerein- oder ausgabe (E/A) erlangen. Ohne eine Möglichkeit für sichere E/A ist sichere Ausführungstechnologie ohne großen Mehrwert für den Endbenutzer, da sensible Daten nicht sicher kommuniziert werden können. Daher ist sichere E/A notwendig.

Um sichere Ein- und Ausgabe zu erreichen, können sensible Daten zwischen einem E/A Gerät und der Anwendung verschlüsselt übertragen werden. Dadurch kann keine Schadsoftware diese Daten abgreifen. Jedoch sind Mobilgeräte wie Smartphones oder Notebooks üblicherweise nur mit handelsüblichen E/A Geräten ausgestattet, welche keine Verschlüsselung unterstützen. Daher muss sichere E/A von sicherer Ausführungstechnologie unterstützt werden.

Abgesehen von ARM TrustZone gibt es derzeit keine Technologie für sichere Ausführung, die sichere E/A mit handelsüblichen E/A Geräten erlaubt. Zum Beispiel ermöglichen Intel's Software Guard Extensions (SGX) sichere Ausführung, lassen jedoch sichere E/A vermissen. Wegen der weiten Verbreitung von SGX in naher Zukunft erforscht diese Arbeit die Möglichkeit von sicherer E/A mit SGX. In einer Machbarkeitsstudie wird SGX von einem sicheren Mikrokernel, genannt seL4, ergänzt. Dieser Mikrokernel wird verwendet, um eine transparente und exklusive Bindung zwischen E/A Gerät und einer SGX-geschützten Anwendung herzustellen. Dadurch wird sichere E/A möglich. Da seL4 vertraut werden muss, erhöht sich die Komplexität der Trusted Computing Base (TCB), also die Menge aller vertrauenswürdigen Komponenten. Das ist vertretbar, da seL4 nicht nur klein sondern auch verifizierbar ist. Zum Ende werden mögliche Hardware-Erweiterungen für SGX diskutiert, um sichere E/A per Design zu unterstützen. Damit würde die Notwendigkeit für einen Mikrokernel wegfallen, was die Integration von sicherer E/A in bestehende Software erleichtern würde.

Stichwörter: Sichere Eingabe, Sichere Ausgabe, Sichere E/A, Sichere Anschluss-E/A, Intel Software Guard Extensions, SGX, Sichere Ausführung, seL4, Mikrokernel, RefOS, OpenSGX

Table of Contents

Chapter 1

Introduction

Information and Communications Technology (ICT) is nowadays a fixed part of everyday life. This is underlined by the global smart phone market, which already exceeded 1.4 billion devices in 2014 [54]. Classical ICT addresses information processing and exchange. Its successor, the Internet of Things (IoT), also incorporates real-world physical actors such as self-driving vehicles and medical devices. Following technical forecasts, the evolution of the Internet of Things will reach between 21 and 34 billion devices by 2020 [39,18]. Applications are no longer limited to a specific device but can assemble a whole compound of devices. Having such a broad and emergent technology, security aspects are essential. GlobalPlatform, a main driver of establishing industry standards for IoT, concludes:

> *"As devices and services proliferate, there are increasing privacy and security concerns: personal privacy must be respected, vehicles must remain safe and not endanger the general public, and critical infrastructure (such as water and energy systems) must not be hacked. Because IoT Devices and services impact others – and potentially society as a whole – these security concerns are paramount."* [33]

In the IoT, security is typically addressed by a dedicated and restricted hardware and software architecture. However, for general-purpose mobile devices like smart phones and notebooks, security often fails for reasons of usability and complexity. A majority of users prefers a rich and intuitive feature set over security. This is not only true for consumers but also for business users. The term bring-your-own-device summarizes the desire of employees for self determination with regard to the mobile device. Thus, companies often establish security policies on top of highly diverse and partially insecure devices.

In addition, the software stack of mobile devices increases rapidly. Steady raise of computing power enables more complex and computation-intense applications, even on battery-powered devices. Other catalysts are a rich set of sensors, enabling new use cases, as well as high bandwidth network interfaces with permanent internet connection. Several incidents in the past showed that it is infeasible to keep a big and dynamic software stack free of bugs and backdoors [1,24,31,8]. The gateway for malware is still wide open. Hence, the main attack surface on mobile devices arises from insecure software.

In order to address this issue, research investigates secure code execution technology to defend against both, logical and physical attacks. A Secure Execution Environment (SEE) aims at keeping sensitive information safe within a secure

container. The SEE is enforced by CPU hardware. It isolates an application from the rich and insecure software stack, ideally protecting against all kinds of malware. While secure execution is essential in protecting applications, it does not address the orthogonal challenge of secure I/O. Secure I/O explicitly requires sensitive information to enter or leave the SEE for interacting with its physical environment. Secure I/O is not only relevant for cyber-physical systems, where sensor input and actuator output needs protection, but also for classical ICT with human interaction. Especially end users, frequently entering PINs and passwords, exchanging private messages, accessing medical data or doing electronic payment, would greatly benefit from secure I/O on mobile devices.

Secure I/O requires a secure communication between SEE and the I/O device. For specific use cases like electronic payment, dedicated I/O devices can be used. Secure I/O is achieved by a cryptographic channel between I/O device and application. However, when addressing general-purpose mobile computing devices such as smart phones and notebooks, one is limited to legacy I/O devices which are either built-in or off-the-shelf. These typically do not provide cryptography. Hence, secure I/O can only be provided up to the I/O port where the legacy I/O device is connected. This secure port I/O requires hardware assistance by the CPU and the chipset.

There is little state-of-the-art technology available which covers both, secure execution and secure port I/O. To our knowledge, ARM TrustZone is the only secure execution technology which allows secure I/O by design. Others like Intel Software Guard Extensions (SGX) can protect outsourced computation in the cloud where no local interaction between the SEE and the user is required. However, secure I/O with legacy devices is not possible with SGX at all. This means that SGX cannot significantly improve security of mobile applications such as secure password stores or secure chat, which all require secure I/O. Since SGX will have a wide dissemination among modern x86 CPUs in the near future, its lack of secure port I/O is unpleasant.

1.1 Contributions

This thesis bridges the gap between SGX and secure I/O on general-purpose mobile devices. We develop a secure I/O architecture, which links Intel SGX with secure port I/O features in software.

Our architecture considers an SGX-protected application running on an insecure rich OS. To allow the application to do secure port I/O, we operate a secure I/O driver on top of the trusted microkernel seL4. We use the strong resource management capabilities of seL4 to establish a transparent binding between the driver and the I/O device as follows: First, giving the driver exclusive and non-revocable ownership of the I/O device. Second, giving the driver tools to transparently verify if the I/O binding is indeed exclusive and non-revocable. Third, allowing the driver to identify the I/O device.

We demonstrate our secure I/O architecture in a proof-of-concept. Therefore, we modify seL4 to support transparent verification of the I/O binding. Furthermore, we integrate SGX support in seL4.

Our concept highlights shortcomings in combining SGX's overly strict trust model with secure port I/O. We outline possible modifications to SGX hardware to reduce total complexity of secure port I/O with SGX. We show how SGX could even do I/O binding in hardware, without the need for a security kernel.

The rest of this thesis is structured as follows: Chapter 2 describes related work in the area of secure execution and secure I/O. Chapter 3 gives general background on secure information systems with a focus on how such systems are built securely. Chapter 4 discusses our secure I/O architecture in detail. It is followed by an introduction to Intel SGX (Chapter 5) and the seL4 kernel (Chapter 6). In Chapter 7 we describe our proof-of-concept and give details on how we implemented secure port I/O with standard SGX and seL4. Chapter 8 discusses identified shortcomings and recommends modifications to SGX hardware to mitigate them. Finally, Chapter 9 summarizes our work.

Chapter 2

Related Work

This section gives an overview of research in secure execution technology. We pay special attention to ARM TrustZone and its approach to secure I/O. Finally, some existing proprietary and dedicated secure I/O technologies are mentioned.

2.1 Secure Execution

Secure execution technology ranges from virtualization and architectural isolation to memory encryption and verification of code authenticity. In literature, the terms secure execution, isolated execution, trusted execution, shielded execution, etc. emphasize on different aspects and are often intermixed. In this work, we use the term *secure execution* to refer to a security container, which is isolated from other software, providing verified launch, protected code execution, protection of secrets and means for attestation.

The Trusted Computing Group [75] initiated a lot of research on secure execution. Using their Trusted Platform Module (TPM) [73], a dedicated, passive security co-processor, one can verify the security state of a whole software stack, beginning at the system boot. Having issues with building such a static trust hierarchy over a large software stack, Intel came up with its Trusted Execution Technology (TXT) [43]. Intel TXT dynamically verifies the system state only when needed. However, a security co-processor alone is a rather passive tool. It does not provide strong isolation between applications. Hence, research tends towards stronger protection mechanisms, solely built into the CPU.

Early software solutions like SP3 and Overshadow virtualize an untrusted operating system on top of a security hypervisor [79,21]. The hypervisor does memory cloaking – a context-aware page encryption. A security-critical application, launched by the untrusted OS, is encrypted by the hypervisor as soon as the OS illegitimately accesses it. As such, strong memory isolation between the application and the untrusted OS is provided. Since then, a lot work was done in moving memory cloaking to hardware and building isolation mechanisms into the CPU [72,22]. IBM SecureBlue++ additionally addresses the issue of running legacy software in a secure execution environment [13]. It furthermore reduces the Trusted Computing Base (TCB) to the CPU and the protected application. Secure execution can also be achieved by keeping all sensitive data on-chip. Using Cache-as-RAM, one can create a secure execution container, entirely running in the CPU cache, with interrupts disabled [64,76].

Recently, Intel released the Software Guard Extensions (SGX) for its x86 mainline CPUs. SGX is a comprehensive secure execution technology, combining different methods for process isolation and memory encryption with TPM-like code attestation and sealing features. Intel is shipping SGX with Skylake CPUs since October 2015 [49], being available to the broad community. We explain SGX in more detail later on.

The secure execution methods, discussed so far, mostly isolate an application from an untrusted OS. Even with secure execution in place, the communication with the underlying, potentially untrusted operating system is problematic [20]. Baumann *et al.* therefore shifted most OS functionality into the secure execution environment, when experimenting with SGX [10].

ARM TrustZone. ARM is a main player in designing processing systems for low-power mobile platforms, complementing Intel's prevalence in servers, desktops and notebooks. ARM TrustZone not only supports secure execution but also secure I/O with legacy I/O devices.

TrustZone adds a new, orthogonal protection domain to existing CPU privilege modes [4]. In accordance to GlobalPlatform's specification of Trusted Execution Environments [32], TrustZone partitions software into a secure and an insecure world. Both worlds are isolated from each other. Each world has its own software stack.

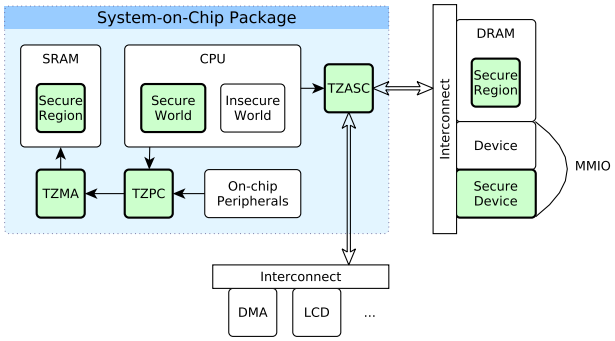

Figure 1: The ARM TrustZone architecture allows protecting on and off-chip memory as well as Memory-Mapped I/O (MMIO) devices, which are bound to the secure world.

TrustZone is complemented by a set of security modules for extending the secure world concept to memory and peripherals. Therefore, a secure world bit is tracked to peripherals via a separate wire. Figure 1 gives an overview of a typical TrustZone architecture. A TrustZone Memory Adapter (TZMA) protects on-chip

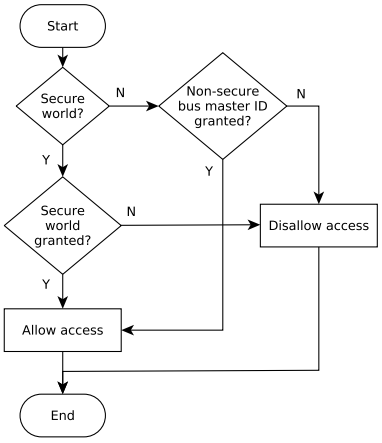

Figure 2: ARM TrustZone Address Space Controller distinguishes between secure and insecure world and does identity filtering for insecure accesses.

memory, a TrustZone Protection Controller (TZPC) secures on-chip peripherals [5,6]. The TrustZone Address Space Controller (TZASC) protects off-chip DRAM and memory-mapped devices [7]. ARM also provides a System MMU for more demanding designs. The System MMU is similar to an x86 IOMMU and can prevent DMA attacks on the main memory.

The TZMAs and the TZASCs implement memory access filtering based on a software-defined security policy. For example, the TZASC does address filtering for multiple non-overlapping address regions. For each address region, one can specify, who is permitted or refused access to it. Therefore, the entity, which issued the memory access, is identified by a bus master ID. Furthermore, an address region can be restricted to the secure or the insecure world. The permission checking is outlined in Figure 2. In addition, read and write access policies are configured independently.

With that, secure-world software can bind memory and devices to secure-world only. Also, TrustZone can route interrupts directly into the secure world, making the implementation of secure I/O drivers feasible. This enables secure I/O as required for mobile devices.

Despite its comprehensive security features, ARM TrustZone requires more effort by software developers than what would be necessary, for the following reasons:

First, it does not directly provide transparent hardware encryption and integrity protection of DRAM as others do [72,22,13,40]. Encryption could be enabled by software [37], leveraging TrustZone's hardware encryption accelerators.

Second, TrustZone does not isolate on a process granularity, as done by SecureBlue++ and Intel SGX [13,40]. Instead, it only uses a single bit of information to isolate the secure and the insecure world. In order to allow multiple secure applications, one needs to run a full software stack in the secure world, including a secure OS with scheduling and process abstraction. This certainly increases the TCB by the secure OS. Also, when deploying apps from multiple vendors to one device, all vendors have to trust the secure OS.

2.2 Proprietary and Dedicated Secure I/O

Secure I/O can be achieved with specialized hardware, either in form of a CPU or chipset extension or as dedicated I/O device. Intel already supports a bunch of secure I/O technologies in its CPUs. However, it is unclear if and how these could actually support SGX in doing secure I/O. Hoekstra *et al.* mention trusted I/O very vaguely when discussing possible applications of SGX in [38]. In their experiment, they claim to have used Protected Audio Video Path (PAVP) to do secure output. However, no further information is provided. Moreover, secure input is left for future work.

Very little information is publicly available about how to actually make use of PAVP. Ruan describes PAVP on a very high-level in [68]. PAVP basically decrypts protected content in GPU hardware such that the host OS cannot access it. Thus, it allows to securely output audio and video content. Key management is assisted by Intel's Management Engine (ME). The successor of PAVP is called Intel Insider, for which details are also scarce [50].

On a higher level, Intel invented High-bandwidth Digital Content Protection (HDCP) [53] to secure the communication between the graphic chip and flat screens. HDCP, in special, underwent a wealth of security breaches in the past, leaving it a rather tame content protection mechanism [29,34,36]. Intel Insider together with Intel Wireless Display (WiDi) extend this concept and protect multimedia content content over WLAN [50,42]. Both, Insider and HDCP are discussed highly controversial, not least because its main beneficiary is the film industry [50]. One can safely conclude that existing secure output technology from Intel mainly focuses on enforcing Digital Rights Management (DRM).

Protected Transaction Display (PTD) is a secure input technology as a part of Intel's Identity Protection Technology (IPT) [19,68]. The PTD makes use of PAVP to display a protected virtual numeric block to the user. The user enters a PIN by clicking the corresponding numerical buttons with the mouse. The numerical buttons are randomly arranged to avoid information leakage via the mouse coordinates. However, PTD is only designed for numerical input. When considering more generic use cases, secure input is typically implemented in motherboard's input controllers, as suggested by some patents [69,25]. Another

example is Intel's Trusted Mobile Keyboard Controller (TMKBC), for which almost no information is publicly available [1].

A big problem of using existing secure I/O technology from Intel is its proprietary, closed nature. Insider, PTD and similar incorporates proprietary firmware, running on the Intel ME. The ME is a separate embedded processor running within the chipset. It has highest privileges, as it runs in System Management Mode (SMM). Hence, bugs in the ME firmware are critical to overall system security. However, Intel does not disclose ME firmware code for transparent security analysis. Moreover, Intel lacks thorough documentation of how its I/O technology can be accessed from the OS. Instead, Intel just provides closed-source drivers and libraries for a limited number of OSes, if publicly available at all. For example, the WiDi software stack is only available for Microsoft Windows 7 and higher as well as for Android since 4.2.2 [42]. This makes existing technology rather worthless for the open-source community.

If existing CPU or chipset extensions cannot be used or are insufficient, dedicated I/O devices are an option. For example, dedicated secure input devices are often used for electronic payment applications like an automated teller machine (ATM). Devices for secure PIN entry and transmission are standardized and certified by the Payment Card Industry Security Standards Council (PCI), which currently hosts a list of over 500 approved secure PIN devices [65].

[1] TMKBC is part of the MEC5025 controller, which has been included in several dated motherboards.

Chapter 3

Secure Information and Communication Systems

Secure Information and Communications Technology (ICT) is both complex and versatile. This section describes secure ICT as a system of cooperating devices.

First, we give some informal but intuitive definitions used throughout this thesis. Next, typical ICT applications are described. Among them are cloud computing, digital rights management, remote-controlled systems and private communication. Our secure I/O architecture will stick to latter scenario, relevant for many mobile device applications. Finally, we show how to combine and protect different building blocks to build a secure ICT system[2]. Special attention is paid to secure execution and secure I/O in defending the system against attacks from software. This lays the foundation for our secure I/O architecture, which implements secure port I/O on mobile devices.

3.1 Definitions

In the context of ICT, the term *security* typically refers to the properties *confidentiality* and *integrity* of sensitive information. Integrity covers both, *data integrity* and *origin integrity*. These security properties need to be fulfilled during the whole lifetime of sensitive information, from input, transport and processing to output. In addition, this requires *execution integrity* for any piece of code, which processes sensitive information. Use cases in mobile computing typically stick to the above properties. However, further properties like *anonymity* and *availability* can be encountered in the more general ICT context. In the following, we define security properties as well as other common terms used throughout this thesis.

Data confidentiality is the main objective of most secure ICT systems. It refers to protection of actual data from unauthorized access. For example, a password needs to be kept secret. Also, chats, emails and business contracts often require confidentiality.

Data integrity means that alteration of data can be detected. A financial transaction, for instance, needs integrity on the amount of money to be transferred. In general, data integrity is especially important for communication protocols,

[2] For a more comprehensive discussion of secure mobile communication systems, we refer to Open Mobile Terminal Platform (OMTP) [63].

signature schemes and data storage where illegal or accidental manipulation of data shall be detected.

Origin integrity refers to the authenticity of the entity from which data originates. For example, in a private communication scenario, users want to be sure that they are indeed talking to each other. If origin integrity is not enforced, an attacker could impersonate users and read and manipulate the communication data on their behalf. A subset of origin integrity is *non-repudiation*. This characterizes the authenticity of a signer. The signer cannot deny having signed some data. Non-repudiation is important for signing contracts and any other sort of legally binding actions.

Execution integrity means that code is indeed executed exactly as specified by the programmer. This is essential in order to entrust code with security-critical tasks.

Anonymity refers to keeping sender and receiver addresses in a communication covert. Thus, anonymity is a subset of confidentiality, where the sensitive information is the sender's and receiver's identity.

Availability typically means high service uptime, which can be threatened by Denial-of-Service (DoS) attacks. In cyber-physical systems unavailability might affect human safety. In contrast, unavailability of ICT systems typically causes financial or reputational damage only.

Entity. Any instance which is part of the ICT system is called an entity. This typically includes computing devices, routers and devices for input and output. Also the end user is an entity in the system.

Secure input is the way an ICT application securely retrieves information from the physical world, and the user in special. For instance, if the user enters a PIN on a keyboard, this PIN shall be only accessible to the application. Secure input is required for any input device acquiring sensitive information. Examples are keyboard, mouse, touchscreen, microphone, camera, fingerprint reader and other sensors.

Secure output is the way of securely transferring processed data back into the physical world. For example, the current bank balance or some private chat messages shall be displayed to the user in a secure way. Secure output is necessary for any device outputting sensitive information. This can be displays, speakers or all kinds of actors, for instance.

Secure Environment. An ideal secure environment is a closed environment, which is tamper resistant and fully side-channel secure. No information in any form can leave or enter it under any circumstance except through a defined interface, see Figure 3. All entities within the secure environment are trusted. Hence no additional security measures are needed and one can safely operate on sensitive information.

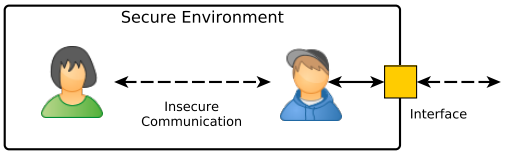

Figure 3: Within a secure environment, no additional protection is needed.

Typically, a secure environment protects security-critical data against different kinds of attacks. Note that from a security perspective one can also speak of a secure environment in the absence of an attacking surface. For example, consider a notebook that is operated within a secure room without network connection. The notebook might be vulnerable to different kinds of software attacks. However, since it is operated within a secure room with no network connection, these attack vectors simply do not apply.

In information and communication technology, there exist two forms of secure environments, a *secure communication channel* for secure data transport, and a *Secure Execution Environment* (SEE) for secure data processing. A secure communication channel is passive, just allowing some entity to forward data in a secure way. In contrast, an SEE is active in the sense that it allows manipulation of and operation on data.

Secure Execution Environment. Secure execution describes the way, a processor securely operates on data. When processing sensitive data, one has to prevent unauthorized inspection and alteration of this data as well as manipulation of the execution flow. An environment which provides this protection, is called Secure Execution Environment (SEE).

Secure Communication Channel. Secure communication is a key asset of many ICT applications. Transport of sensitive information needs to be secured against wire tapping and man-in-the-middle attacks from untrusted entities. Therefore, a secure communication channel is required.

Attestation. In distributed ICT applications it is not enough to provide secure execution along with secure communication. If a malicious instance could manage to trick execution into a simulated execution environment, it could circumvent any of its security mechanisms. To tackle this issue, attestation mechanisms need to be in place. These allow a verifier to assess the security state of the computing device.

Trusted Computing Base. The Trusted Computing Base (TCB) comprises all hardware and software parts which must be trusted in order to guarantee certain security properties. Typically, the TCB covers at least the CPU and some secure application running on it.

General-purpose Mobile Device. A mobile device is general-purpose if it is not restricted to a specific use case. It typically allows installation of applications on the discretion of the user. Thus, general-purpose mobile devices are highly user-centric. Prominent examples are smart phones and notebooks.

3.2 Applications

To have a common notion of security in drawings, we distinguish between solid and dashed lines. Solid lines are used for both, security because of the absence of an attacker and security due to security measures in an insecure environment. In contrast, dashed lines denote both an insecure environment and an insecure communication path, as an attacker is able to infiltrate it. Figure 4 shows an example where Alice and Bob are communicating over a secure communication channel that protects against wiretapping, for instance.

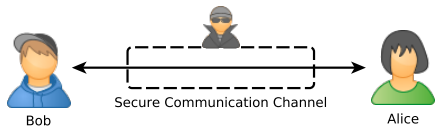

Figure 4: Solid lines indicate secure components while dashed lines are used for insecure components.

3.2.1 Secure Cloud Computing. Cloud computing allows to use remote computing power for executing CPU-intensive tasks. When considering security-critical applications, not only the communication network to the cloud provider has to be considered insecure but also the cloud provider itself. While the communication can be easily secured using any form of secure communication channel, there is special need for protecting remote code execution against a malicious cloud provider. The cloud provider shall only be able to choose between securely executing user code or refusing execution at all.

This requires a secure execution environment at the cloud provider's infrastructure. Security of the SEE needs to be independent of the cloud provider's software framework. Hence, the SEE has to build its security solely on tamper-resistant CPU hardware and/or user's secure code. In addition, the user has to be able to remotely attest the trustworthiness of the SEE. With that it is possible to do cloud computing in a secure way, as demonstrated by Microsoft Haven [10]. Secure cloud computing comes without the need for secure I/O at the cloud provider, since all communication with the user is done via a secure channel, as shown in Figure 5.

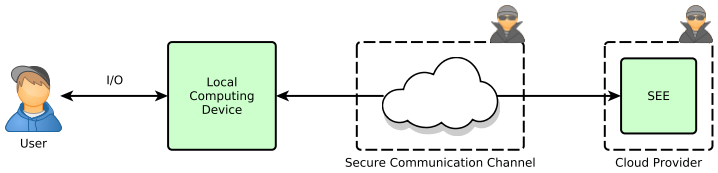

Figure 5: Secure cloud computing allows secure applications to be executed by an untrusted cloud provider.

3.2.2 Digital Rights Management (DRM) aims at protecting content from disallowed usage, copying or modification. The term DRM is also used for other protection mechanisms such as vendor locks. However, we focus on content protection only. Users get access to protected content only via authorized media players and output devices as shown in Figure 6. The media player receives protected content from the content provider, which can be a broadcast as well as a local BluRay disc. This content is then presented to the user via a secure output device.

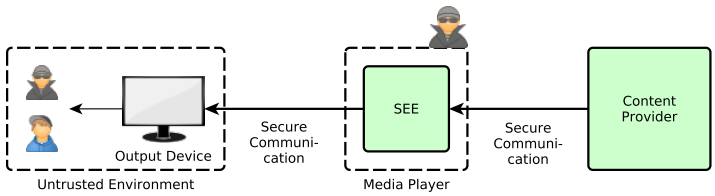

Figure 6: Digital rights management protects content up to the secure output device.

For DRM, the user is considered as malicious, trying to obtain an illegal copy of protected content. Since regular users are able to consume what they purchased legally, they eventually have access to this protected content. A malicious user could legally buy content which he then records during playback. This discrepancy between regular and malicious users poses certain limits on DRM. Thus, the goal for DRM is to push content protection as far into the output device as possible. Ideally, the malicious user can only reveal an analog representation of protected content, which cannot be easily reconstructed in a lossless way.

An example of DRM is High-bandwidth Digital Content Protection (HDCP).

3.2.3 Remote-controlled Systems are typically operated within a hostile environment. A system operator remotely controls the system by querying remote sensors and instructing remote actuators. Autonomous systems also fall into this category. Apart from doing most of the controlling tasks on their own, autonomous systems are also instructed by a system operator via some form of high-level instructions.

On the operator's side there is no special need for secure I/O since the operator has to be trusted anyway. Besides a secure communication channel to the remote-controlled system, there is need for the remote system to run an SEE, see Figure 7. Secure I/O is limited up to the sensors and actuators. At some point, sensors capture physical input and actuators produce physical output. If the physical environment is not secure, an attacker can at least mount physical attacks, manipulating sensor data or actuator output. This can only be prohibited by physical access control mechanisms, if possible at all.

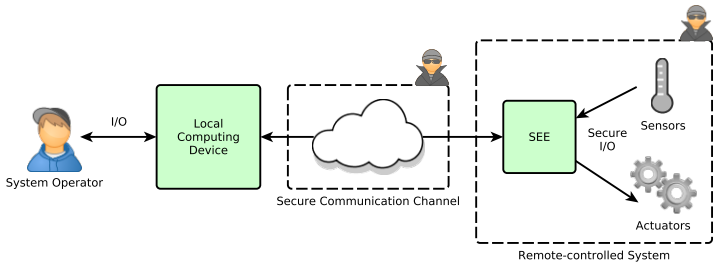

Figure 7: A remote-controlled system is operated in a physically insecure environment.

Especially for critical infrastructure like traffic control systems and industrial control units, a system operator also wants to attest correct behavior of the system. This is also true for any kind of monetary application such as smart meters, for example. Hence, strong attestation methods are required.

Cloud computing can be seen as subset of a remote-controlled system, which only provides computing power but no sensors or actuators. Also, DRM can be seen as an inverse remote-controlled system from the viewpoint of the content provider acting as system operator. DRM output devices have to operate remotely within a malicious user environment. Typically, there are no remote sensors or input devices on the user's side. However in contrast to remote-controlled systems, DRM and cloud computing traditionally suffer from software attacks. Conversely, remote-controlled systems can have a decreased susceptibility to software attacks due to a rather static firmware and the possibility for regular maintenance.

3.2.4 Private Communication. Much societal attention is paid to private communication, not least because of disclosures on governmental mass surveillance in the past. Key goal is to enable a group of humans to communicate over an untrusted network in a secure and privacy-preserving way, see Figure 8.

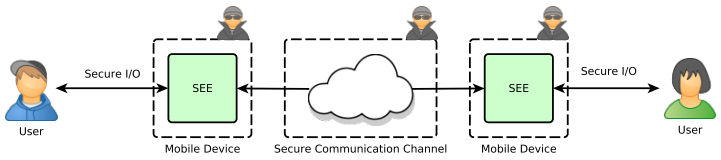

Figure 8: Private communication.

To protect communication data, a secure communication channel is established over the untrusted network. This might involve complex authentication methods, which are not detailed here. To enhance security, users can mutually attest the trustworthiness of their mobile devices. Since secure communication is possible with today's cryptographic protocols, the focus shifts towards securing the mobile communication devices. While physical attacks have to be considered for theft and bugging, for example, the far more dangerous threat is an insecure software stack, running on the device, that might be exploited locally or remotely. The challenge is to combine secure I/O with an SEE such that no malware can sniff on sensitive traffic.

The above attack model does not only support private communication such as secure chat, audio and video calls. It can also be applied to authentication schemes, where a secret password, PIN or TAN code is entered on an insecure device. This is of special interest for e-government, e-payment, e-health and similar applications.

3.2.5 Application Requirements. Summarizing the threats and security requirements for different applications yields Table 1. We distinguish between a local environment in the scope of the user and a remote environment. Latter contains any devices, which interact with the user remotely. Hardware attacks refer to physically tampering of a device. Software attacks are all kinds of attacks which can be carried out by software, either locally or remotely.

Table 1: Application threats and security requirements are marked with an x.

Application	*Local Environment*				*Remote Environment*					
	HW attacks	SW attacks	SEE	Secure I/O	HW attacks	SW attacks	SEE	Secure I/O	Secure Channel	Attestation
Cloud Computing					x	x	x		x	x
Remote-controlled System					x	x	x	x	x	x
DRM	x	x	x	x					x	x
Private Communication		x	x	x		x	x	x	x	x

All of these application require an SEE with the ability for attestation. Likewise, secure communication channels are part of each application scenario. Secure I/O is needed for remote-controlled systems, DRM and private communication. However, hardware attacks on secure I/O only apply to remote-controlled systems and DRM. Private communication requires secure I/O in the presence of software attacks, with a strong focus on mobile devices. Latter is the main scenario addressed by this thesis.

3.3 Building a Secure ICT System

This section describes how a secure ICT system is built from some basic building blocks. In order to understand how security can be enforced, we introduce different dimensions of access control. Then, we give examples of concrete access control mechanisms for each building block. Special attention is paid to securing against software attacks.

3.3.1 Building Blocks. Secure ICT systems rely on building blocks which provide security in an insecure environment. These cover secure input, execution and output as well as secure communication channels, as depicted in Figure 9. In addition, to evaluate the security state of running ICT applications, a secure execution environment has to provide attestation methods.

To get a secure system, these building blocks are arranged to create a gapless secure environment, spanning the whole processing chain from input to output.

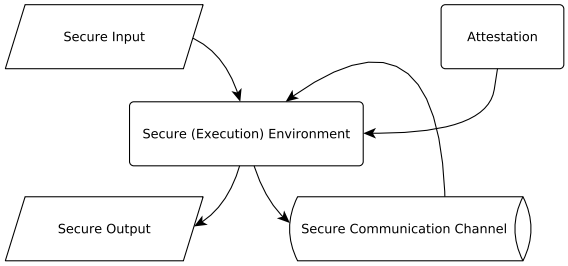

Figure 9: Hierarchy of building blocks in a secure communication system.

This is achieved by a continuous concatenation of SEEs with secure communication channels. The remaining link to the user is done by secure I/O, as shown in Figure 10.

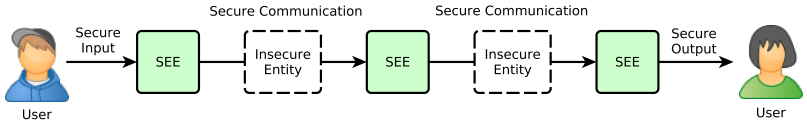

Figure 10: Processing chain: A secure ICT system is built by connecting SEEs with secure communication channels and interacting with the user via secure input and output.

3.3.2 Dimensions of Access Control. In order to enforce certain security properties, access control mechanisms are needed. This comprises a physical and a logical dimension. Unlike in literature, we introduce a semantic dimension to furthermore refine the logical dimension. This distinction is useful in understanding the underlying trust model. In our notion, logical access control is enforced via a piece of hardware or software. Hence, one has to trust exactly those parts of hardware or software to behave well. Semantic methods only rely on the strength of cryptographic algorithms. Hence they relax this spatial restriction to specific secure software or hardware. Semantic methods are useful if some entities in the processing chain are entirely distrusted.

The physical dimension addresses physical attacks that exploit weaknesses in the hardware design. Among them are fault injection and tampering with memory, for example.

The logical dimension covers all kinds of attacks, which exploit implementation bugs or misconceptions of the architecture or the interfaces. Logical access control primarily deals with attacks from software. Thus, it is of particular interest for hardening an SEE against an insecure software stack.

The semantic dimension addresses data directly, letting surmise its strength in building secure environments. Rather than actively enforcing access restrictions, one obfuscates sensitive information to get a ciphertext. The ciphertext might be accessible to anyone, without breaking the confidentiality of data. Access to the original information is only possible for those entities which have certain semantic knowledge. Therefore, Kerckhoffs prepared the way for modern cryptography: A strong cryptographic scheme binds access to sensitive data to a cryptographic key [47]. Since only the encryption key needs to be protected, semantic methods provide kind of a self-contained secure environment, which is independent of the security of the underlying hardware and software. Hence, they can defend against both, physical and logical attacks.

3.3.3 Secure Execution Environment.

Depending on the use case, an SEE might require protection against hardware and software attacks. The SEE can be protected using a combination of physical, logical and semantic measures, as described hereafter.

Physical Access Control. Depending on the level of security, physical countermeasures range from mechanical barriers and fault resistance to intrusion detection sensors and automated erase of sensitive data. A detailed list of physical defense methods is found in [77]. However, with sufficiently big effort an attacker will be able to break any computing device based on CMOS or similar technology. This is not least because CMOS inherently leaks information via power side channels [55]. In the future, quantum technology could yield a new level of side-channel resistance which, is currently unreachable [14].

Logical Access Control. In the past, a lot of effort was put into securing user applications from each other and protecting an operating system kernel from malicious applications. The Memory Management Unit (MMU) provides memory isolation between different applications via the concept of virtual memory. Privilege modes or protection rings separate the OS from applications. Special page protection flags shall inhibit damage on security breaches. Virtualization and sandboxing introduce another layer of isolation between the virtualized app and the host.

In contrast, recent work focuses on the opposite direction: securing a user application against a malicious operating system. This reduces the amount of software in the TCB. Since the OS is responsible for resource handling, one requires stronger protection mechanisms. For example, the application's memory needs to be protected against manipulation by a malicious OS. Hence, new isolation mechanisms for processes and memory need to be enforced solely by CPU hardware. The CPU typically achieves this by identifying a secure application and

tagging all of its memory in CPU caches as well as in RAM with the application's identity [13,40]. Thus, the CPU can deny access if the identity of the currently running application does not match the memory tag.

Furthermore, integrity of code execution is crucial for security. If an insecure piece of software could change code execution to jump right into an SEE, this opens powerful attacks, completely breaking confidentiality of SEE data. The attacker could for example choose to execute an SEE function from within a wrong context, possibly outputting sensitive information from the SEE stack. Return-oriented programming might enable arbitrary code execution, given that the SEE code base is big enough [16]. Thus, executing SEE application code needs to be restricted to defined entry points. This is typically achieved by confining SEE launch to special CPU instructions, which adhere to those defined entry points.

Also, if the insecure OS can dynamically map pages into the SEE, the SEE, needs to have control over which pages shall be mapped to which location. This can be achieved using a cooperative approach where the SEE acknowledges new pages with a special instruction, for example [40].

Whenever an SEE can be interrupted by an external signal, additional protection is necessary. Typically, the application traps into the kernel. If the kernel is considered insecure, it might has access to sensitive information, stored in the CPU registers. Hence, the register file needs to be protected by the SEE. This can be achieved by copying the register file to encrypted SEE memory before clearing it and giving control to the OS [13,40].

Semantic Methods. Semantic methods for secure execution evolved during the last years. Memory cloaking prevents information leakage by transparent memory encryption. This can protect against software attacks as well as hardware attacks [79,21,22,13,40].

Also, integrity of the secure application is decisive as soon as untrusted code is responsible for building and maintaining a secure application process. To verify integrity, the application's state is typically derived as a chained hash over all secure code and data. In literature, such a hash is called measurement [73,40]. If the measurement is correct, the secure application is in an authentic state. This can be used for sealing data. Sealing means that data is encrypted under a key, which is dependent on the application's measurement. Hence, only if the application is loaded correctly, it can decrypt sealed data again.

Furthermore, one can combine the concept of measurements with logical access control to provide verified application launch and attestation. Verified application launch means that the SEE verifies if secure application has been loaded correctly before actually launching it. Therefore, the application vendor typically includes the expected measurement in the application's binary and signs the binary. Only if the vendor's signature is valid and the application state matches the expected measurement, the SEE decides to launch the application.

Attestation is similar to verified application launch. Attestation typically allows to verify integrity of the secure application during runtime. The SEE therefore signs the application's current measurement. By verifying the measurement and the signature, a remote party can gain confidence in the integrity of the secure application. Thus, the remote party could refuse provisioning secrets to an SEE which has been tampered with.

Homomorphic Encryption. A completely different approach to secure execution is homomorphic encryption. In theory, secure execution does not only work on plaintext data. Homomorphic encryption schemes operate on encrypted data. This allows outsourcing computation to an insecure entity, without having special hardware or software protection mechanisms in place. The insecure entity is not able to see plaintext data since it has no access to the decryption key. When computation is finished, the results are transferred back to a local SEE, where they are decrypted, integrity checked and evaluated. Thus, homomorphic encryption schemes can be used to build a secure execution environment, solely based on cryptography.

In order to support arbitrary calculation on encrypted data, Fully Homomorphic Encryption (FHE) is required, This comes, however, with enormous computational costs. For example, recent practical implementations of FHE easily require several gigabytes of RAM, apart from computation times in the range of seconds and minutes [35]. Furthermore, standard FHE only deals with data confidentiality but does not give any guarantee about the integrity of computation results. There exist, however, FHE approaches which provide verifiable computation [23].

FHE is a purely semantic method for providing secure execution. However, due to practical limitations of FHE, one needs to build secure execution using a combination of different access control methods, as described before.

3.3.4 Interaction with Insecure Code.

Whenever the TCB interacts with an insecure entity, it has to ensure that security properties are not violated. This is in fact just a different view of our security definition, requiring security during the whole processing chain. Yet, it is an important principle for designing a secure system, worth mentioning here. We briefly give some examples on the basis of secure execution.

When an insecure OS is able to dynamically map pages into a running SEE, this needs cooperation by the SEE. For example, the SEE verifies external changes in its virtual memory mapping and either refuses or accepts them. Likewise, secure application launch relies on insecure code. Hence, the SEE implements verified application launch where it verifies if the application is in a secure state before running it. Furthermore, when insecure code enters an SEE, this is restricted by the SEE to secure code entry points. Also, when leaving secure execution, the SEE protects sensitive content in the CPU register file before giving control to insecure code.

Thus, the SEE has to verify or restrain every action carried out by insecure code that might threaten security of the SEE.

3.3.5 Secure Communication Channel. A physically secure communication channel is rarely practical, if not impossible for many applications. For example, wired networks are not physically secure as soon as a malicious entity might gain access to it. Wireless communication even increases the attack surface since anyone being in radio range can easily tap the communication. Furthermore, in an internet setting any gateway or internet service provider involved in the communication can do packet inspection and manipulation.

Luckily, semantic methods can be applied here. Cryptographic protocols allow to set up a secure communication channel on top of such untrusted networks. With end-to-end security, no other network entity needs to be trusted. There exist various cryptographic protocols for different security requirements. In any case, security essentially reduces to bootstrapping cryptographic keys among secure ICT entities. A detailed classification of different key bootstrapping methods is given in [59].

3.3.6 Secure I/O. Secure I/O addresses the interaction between I/O device and user. However, if the user is either untrusted or within a physically insecure environment, security might be violated. This is the case for DRM, where protected content is given to a potentially malicious consumer. The same way one cannot guarantee full security for a remote-controlled system, whose sensors and actuators operate within a physically insecure environment. They might be exposed to all kinds of physical attacks, manipulating physical sensor input or actuator output directly. Hence, from the perspective of ICT systems, the scope of secure I/O ends where data enters or leaves the physical world.

Depending on the application scenario, secure I/O is either done with dedicated secure I/O devices or with legacy I/O devices off-the-shelf. Dedicated devices can be specifically designed to achieve secure I/O. As already mentioned, this is rather impractical for many mobile applications. Hence, we have to consider legacy I/O devices, which have no special security built in. This requires security up to the I/O port, where the legacy device is connected.

Dedicated secure I/O devices typically support cryptographic protocols in order to establish a secure communication channel to an SEEs in the system. The secure channel not only protects against a malicious driver stack. It also detects tampering of the physical connection to the mobile device. Hence, having the dedicated secure I/O device as close to the user as possible can significantly decrease the attack surface. Furthermore, a small and dedicated design can be protected much easier than a rich-featured general purpose computing device.

As an example, we briefly discuss card TAN generators. The bank provide its customers a special piece of hardware, the card TAN generator, allowing them

to do online banking in a secure way. When initiating online transactions on an insecure notebook, the user has to authenticate the transaction. First, the user feeds transaction details into the card TAN generator. Next, he enters a secret PIN on the card TAN generator. Finally, the generator outputs a one-time Transaction Authentication Number (TAN) which is then used to authenticate the transaction.

Malware on the notebook cannot comprise secure transactions since it has no access to the generator. Thus, shifting secure I/O from an insecure environment (the notebook) into a dedicated secure I/O device (the card TAN generator), significantly improves security.

Secure Port I/O and Secure MMIO. Most legacy I/O devices such as Human Interface Devices (HID) do not support cryptographic channels. Typically, also I/O controllers are closed legacy devices, incapable of cryptography. This completely eliminates the possibility of using semantic access control for secure I/O. Hence, security can only be established up to the I/O port or the I/O controller, respectively, where the device is connected to. This is not only true for classical x86 I/O ports but also for Memory-Mapped I/O (MMIO) devices.

Note that there is no point in using physical access control either since hardware attacks will easily succeed when directly targeting an unprotected legacy I/O device. Thus, one can only use logical access control to achieve secure I/O on general-purpose mobile devices. The next section shows how to build a secure I/O architecture which achieves secure port I/O with logical access control mechanisms. This essentially establishes an I/O binding between the I/O port and the SEE.

Chapter 4

Secure I/O Architecture for General-Purpose Mobile Devices

In this section we develop an architecture for doing secure I/O on a general-purpose mobile device such as a smart phone or notebook. Since mobile devices typically do not feature dedicated secure I/O devices, our architecture focuses on secure port I/O with legacy I/O devices. We combine Intel SGX with the seL4 security kernel. SGX provides a secure execution environment for applications, called enclave. seL4 enables the implementation of secure port I/O features. The architecture protects against all kinds of malicious software, running on the mobile device. This enables and enhances security of a vast amount of mobile applications doing private communication, authentication, electronic payment, *etc.*

First, we explain our secure I/O architecture. We then discuss security aspects of the architecture. Next, we show how secure port I/O is achieved with seL4. Finally, we address secure I/O from a user's perspective.

4.1 Secure I/O Architecture

In our setting, a big and mostly untrusted software stack is installed on the mobile device. The software stack contains a rich OS like Android or Linux, including a bunch of software libraries, frameworks, drivers and applications. This properly reflects the situation of current general-purpose mobile devices, which are designed to fit a vast amount of use cases.

Our architecture considers a security-critical application, which wants to exchange private messages with a user via a legacy I/O device. The user application might be a front end to a larger private communication network, such as a secure chat, for example. It runs in an SEE on top of the insecure software stack, as shown in Figure 11.

The user application wants to securely communicate with the I/O port. However, secure port I/O is not possible over the insecure software stack due to lack of support by SGX. Therefore, the security kernel seL4 runs below the rich OS stack. We add a secure I/O driver directly on top of the bare-metal seL4 kernel to implement secure port I/O. The user application talks with the I/O driver via secure Inter-Process Communication (IPC) and the I/O driver securely mediates private messages between the application and the I/O port.

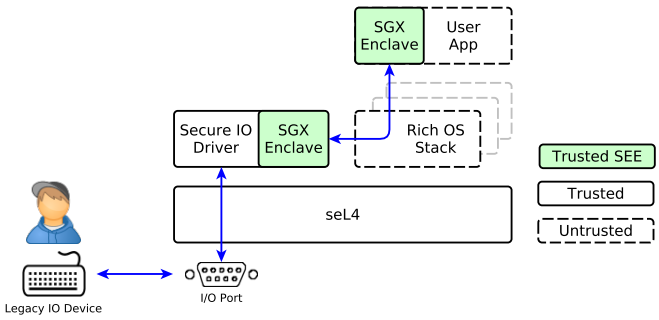

Figure 11: Secure I/O architecture: A user application, running on top of an an untrusted software stack, securely communicates with a secure I/O driver. The I/O driver does secure port I/O to a legacy I/O device for interacting with the user.

Secure Execution. As already discussed, experience in the past showed that a big software stack is likely to have bugs. Hence, the rich OS has to be considered as insecure. We use the term *insecure* equivalently with the term *untrusted*. To protect the user application and the driver from the untrusted rich OS, both are running critical code in an SGX enclave.

Secure IPC. IPC is provided by the untrusted rich OS. Since the rich OS can arbitrarily inspect or alter communication data between the application and the driver, IPC has to be protected. To make IPC secure, we set up a secure communication channel. This involves mutual authentication between the application and the I/O driver. The authentication process typically includes the exchange of a symmetric encryption key. When authenticated, both, the application and the driver can exchange encrypted messages. SGX directly supports authentication between the application's and the driver's enclaves.

Secure Port I/O. SGX lacks methods to protect actual I/O between driver and I/O port. Malware could grab port input by issuing the proper CPU instruction. While it might not be able to directly inspect outgoing port traffic, it can at least subvert it by outputting wrong data. Hence, secure port I/O requires an exclusive binding between I/O port and driver, that cannot be forged by any kind of malware. To implement such an I/O binding, the driver runs directly on top of seL4 and uses seL4's strong resource management capabilities to get exclusive access to the I/O port.

Having secure execution, secure IPC and secure port I/O in place, the application can eventually interact with a user in a secure way, even in the presence of malware. Figure 12 gives an example flow from authentication over establishing an encrypted channel to secure input and output.

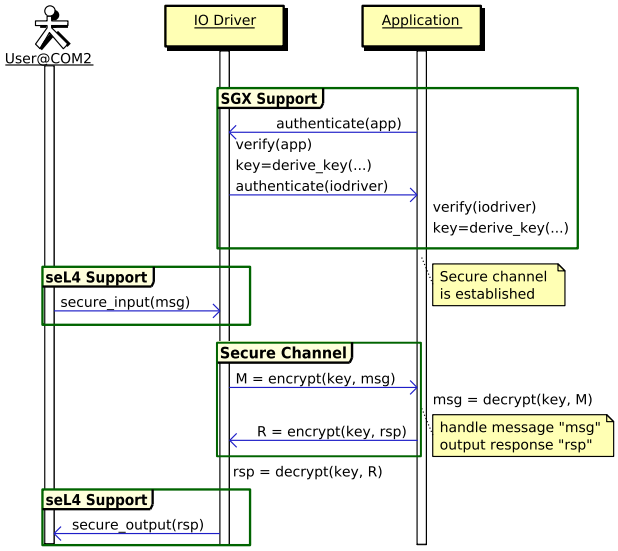

Figure 12: Secure interaction between application, driver and user: SGX supports authentication and key exchange between application and driver to set up a secure channel. Note that the channel might not only encrypt but also integrity-protect messages. seL4 supports secure port I/O with the user.

Bootstrapping. The seL4 kernel is the most trusted software in our architecture. To prevent manipulation of the seL4 kernel by the rich OS, it is the first piece of code that is executed. seL4 protects itself from other software by a strict memory management. The rest of the software stack is executed on top of seL4. To provide a way of checking the integrity of the seL4 boot process, one can leverage Intel TXT and the TPM.

Our architecture is nonrestrictive in the sense that the rich OS shall be able to maintain I/O resources. Thus, the secure I/O driver is bootstrapped dynamically by the insecure rich OS when needed.

4.2 Security Aspects

To assess the security of our architecture, we first describe the Trusted Computing Base (TCB). We then give the threat model and explain why seL4 is secure even in unprotected memory. Finally, we argue why malicious peripherals do not threaten our design.

4.2.1 Trusted Computing Base. SGX can be effectively used for secure cloud computing, in which all code from the cloud provider is untrusted. Hence,

in SGX the TCB only consists of the CPU and the enclave. However, for our scenario, this has to be relaxed. Apart from the CPU, the user application and the driver, the TCB contains other components as well. First of all, the seL4 kernel is essential part of the TCB, providing secure port I/O functionality in software. Second, the chipset needs to be part of the TCB as well, since the mobile device might contain untrusted peripherals, attached to the memory bus. Hence, the chipset needs to protect the communication between CPU and I/O port from these potentially malicious peripherals. The full TCB is shown in Figure 13.

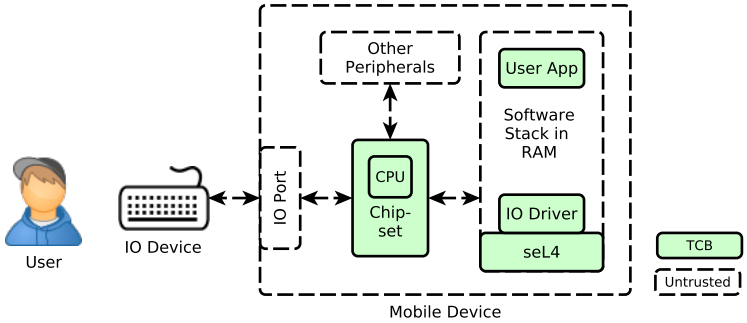

Figure 13: The trusted computing base contains the CPU and the chipset as well as seL4, the user app and the I/O driver.

4.2.2 Threat Model. SGX protects enclaves against both, hardware attacks on memory and software attacks. Memory protection does not only cover memory bus probing and forcing but also direct attacks on the RAM, tampering with memory content. Note that memory is also protected against attacks from malicious peripherals on the memory bus. Software attacks comprise malware at any privilege level. The following list gives an overview of threats against which SGX enclaves are protected:

- Physical threats
  - Bus probing and forcing
  - Memory tampering
  - Malicious peripherals
- Logical threats
  - Malicious applications

- Malicious drivers
- Malicious rich OS

However, since SGX's trust model considers all kernel code as insecure, SGX only allows user applications to leverage enclave protection. Hence, the seL4 kernel does not benefit from SGX protection at all. It remains unprotected in memory. Since seL4 is critical in providing secure port I/O, we have to refine the SGX threat model to our architecture.

In the beginning, we stated that the main attack surface arises from software threats. We argued that the big software stack has an increased susceptibility to bugs. However, this is not the sole reason for sticking to software threats. In the mobile device scenario, users typically have no interest in attacking their own device. Hence, we can safely ignore physical attacks for normal device operation. However, if the device gets lost or stolen, physical attacks eventually apply. When distinguishing between normal operation and device loss, we eventually get the same level of security as with pure SGX for latter case.

Device states. We differentiate between two operating states of the mobile device: an active and a passive state. In active state the device is actively used to do secure I/O by a legitimate user. No physical attacks apply as long as the user is not malicious. The unprotected seL4 kernel is safe in RAM.

When in passive state, the device is locked and cryptographic keys are unloaded[3]. If the device gets lost or stolen, an attacker could actively mount physical attacks on hardware, possibly subverting seL4. Since there is no sensitive communication with the user, sensitive data remains safe within the SGX enclave. Figure 14 shows the physical attack surface in both device states. Note that software threats are not depicted but apply equally in both cases.

Thus, our architecture not only protects against software attacks but in addition defeats hardware attacks on the memory of a lost or stolen mobile device. Furthermore, our architecture protects against malicious peripherals in both device states.

4.2.3 Malicious Peripherals.

Often, a computing device features multiple peripherals of different vendors like mass storage controllers or network interface controllers. Such peripherals might be untrusted, either because of a weak security design, allowing malware to rewrite its firmware, or because of built-in backdoors. If such untrusted peripherals have Direct Memory Access (DMA), they can directly manipulate RAM content. Since the seL4 kernel is unprotected in RAM, it needs special protection against malicious DMA peripherals. Modern chipsets therefore feature an IOMMU. Like a Memory Management Unit (MMU)

[3] Note that such mechanisms are already implemented by iOS data protection classes [3] and BlackBerry two-factor encryption key generation [11].

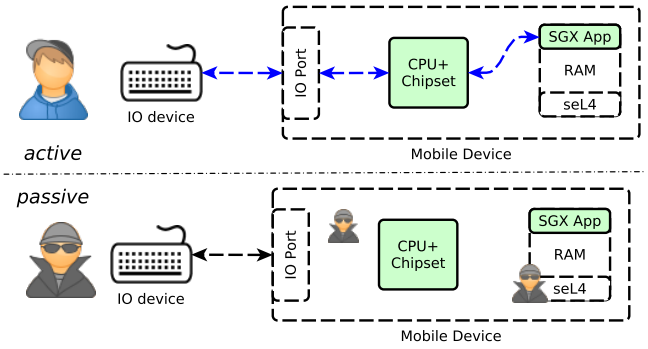

Figure 14: Top: In active state, the application is exchanging sensitive content with the legitimate user (blue line). The user is trusted, no physical attacker is present. Bottom: The device is in locked passive state, doing no secure I/O. If it gets lost or stolen, it might be subject to physical attacks, however secret information is protected by SGX.

translates virtual to physical addresses, the IOMMU does device to physical address translation, restricting the portions of physical memory accessible to the device. When properly configured, each DMA-enabled peripheral has limited access only to those memory regions it is allowed to. Configuration is done by the seL4 kernel. Thus, an IOMMU protects against malicious peripherals.

Recent research suggests direct communication between DMA-capable peripherals [52]. In that case an IOMMU must not only protect the kernel in RAM but also any Memory-Mapped I/O (MMIO) device used for secure user communication, as depicted in Figure 15.

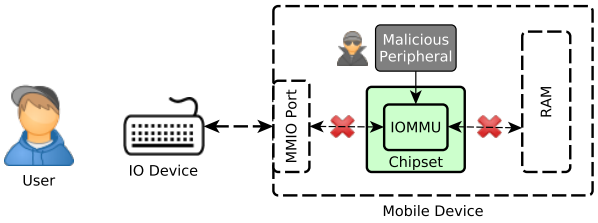

Figure 15: Malicious peripherals might not unauthorizedly access and manipulate RAM or other MMIO devices via DMA. This is enforced by an IOMMU within the chipset.

4.3 Building Secure Port I/O

In order to do secure port I/O, the driver needs to have a transparent binding to the I/O port. In the following, we first give requirements for such an I/O binding. Afterwards we argue why each requirement is necessary, and outline how such an I/O binding can be enforced by seL4.

An I/O binding is characterized by

- Exclusive ownership of the resource
- Non-revocability of this ownership

An I/O binding is transparent to the driver if the driver is capable of doing the following checks:

- Verification of the I/O binding
- Identification of I/O resource

I/O Binding. The first requirement of an I/O binding states that the driver needs to be assigned the I/O port exclusively. This means that only the driver can access the I/O port at a specific moment in time. This is required to prevent malware from simultaneously accessing the same I/O port. However, malware, which has enough privileges, might be able to simply revoke such an exclusive ownership. It might even remain stealthy by just temporarily revoking the ownership while tampering the communication. Hence, the I/O binding needs to be non-revocable.

Transparency is essential because the establishment of the I/O binding involves insecure code. Without transparency checks, a malicious OS could for example violate exclusivity and keep a backdoor open for accessing the I/O port. Therefore, the driver needs a way to verify if its ownership of the I/O port is indeed exclusive. Furthermore, if a malicious OS assigns the driver a wrong resource, all sensitive I/O might be redirected to a different resource. Therefore, the driver needs to be able to identify the acquired resource.

SGX does not directly support such an I/O binding in hardware. Its main objective is to protect software execution in an adverse environment. Hence, we have to entrust privileged software with this task. While standard resource management features are part of every OS, the above I/O binding demands a highly restrictive permission system. To meet this requirements, we use a modified version of seL4.

I/O Binding with seL4. In order to establish an I/O binding, seL4 provides strict but dynamic resource management with an appropriate permission system. This can be used to give the driver exclusive ownership of the I/O port. In seL4, exclusive ownership not only covers exclusive access to the resource but also full control over the ownership. Once established, no other process can revoke this ownership. Furthermore, seL4 implements strong process isolation. Thus, the driver process can be hardened such that no other process has control over it.

To get transparency of the I/O binding, we need some adaptations to seL4. If a process just wants to check ownership of a resource, it can simply try to access the resource. seL4 will refuse access if the process has no access permission. However, verification of *exclusive* ownership is not possible with seL4. Although a process might have exclusive access, it simply cannot know if that is the case. There are two alternatives to still provide this transparency: 1. trust other seL4 processes to behave correctly, or 2. provide methods to dynamically assess the exclusive ownership of caps.

The first option is unacceptable in a generic software architecture, where resource management is delegated to the insecure software stack. Furthermore, the insecure OS shall be able to load and unload the secure I/O driver dynamically.

Therefore, we consider the second option. It requires a dynamic and transparent method for the driver to verify the I/O binding. This method has to be provided by the trusted seL4 kernel itself. Hence, we extend seL4 by a new syscall, named `HasExclusive`, which implements this missing functionality. Using that, the driver can verify exclusivity of a resource.

Identification of the I/O port is easy. When attempting to access the I/O port, the driver provides the port address. If the I/O binding does not match this port address, seL4 will reject access.

Security Considerations. Once the secure driver is loaded and the I/O binding is established and verified, no malware can sniff or manipulate the communication between driver and I/O port. Thus, secure port I/O is possible.

However, bootstrapping the secure I/O driver is also a security critical task. Since the I/O driver is loaded by the insecure rich OS, this OS could manipulate not only driver's code but also the I/O binding. Code manipulation can be detected by SGX. Likewise, manipulation of the I/O binding is detected by the driver by verifying exclusive ownership and identifying the I/O port. However, if the rich OS refuses loading the driver at all, it could load malware instead, which in turn could forge the communication with the user.

One cannot hinder the rich OS to load malware instead of the secure I/O driver. However, even if it does so, secure port I/O can still be protected. In any case, secure output cannot be undermined. If the security application is not able to talk to a genuine, secure I/O driver, which has been loaded correctly, it simply withholds secret information. Protection of secure input is addressed in the following.

4.4 Secure Input from a User's Perspective.

So far we have discussed security from an application's perspective. Now consider an unlock screen of a smart phone, where the user has to enter a secret PIN. How can the user know that he is indeed communicating with the correct unlock

application and not with phishing malware? With secure input alone, a user does not learn anything about the security state of the application or the driver. There is simply no reliable feedback channel to the user. Hence, we combine secure input with secure output to provide this feedback.

In order to notify the user that the application and the driver have been loaded correctly, the user is presented a distinctive, unforgeable feature via secure output. The user shall only provide sensitive input to the smart phone if he observes the correct distinct feature. Note that each secure application requires a different feature. There are two options to make such features unforgeable: First, by specialized hardware and second, by a user-defined secret identifier.

The first option requires additional support from either the output device or the chipset and the CPU. Although our architecture sticks to legacy devices, we explain this option for the sake of completeness. For example, a security LED, could directly reflect the driver state to the user. Likewise, a graphic controller could draw a green frame overlay for indicating a secure driver. The graphic controller prevents software from forging this green frame. Such secure output devices either require a separate wiring to the chipset and the CPU or have to support cryptographic channels.

The second option applies to our architecture. The user provisions a secret identifier to the unlock application beforehand[4]. This secret identifier is protected by an SGX enclave against malware. When attempting to query the PIN from the user, the unlock application first acquires the secure input driver and verifies its security state. It then acquires a secure output driver and displays the secret identifier to the user. When the user sees the correct secret identifier, he knows that he is indeed talking to the unlock application.

Physical Attacks do not apply. One might think that the user-defined secret identifier can be easily forged by a physical attacker. In the above scenario, the attacker could easily grab the smart phone and attempt to unlock it. The unlock app then reveals the secret identifier to the attacker. The attacker can provision this secret identifier to malware on the smart phone, which in turn forges the unlock application.

However, we consider physical attacks only for theft or loss of the mobile device. In both cases, the device might not be returned to the user. If the attacker would return the device to the user after manipulating it, he could also choose to install a wiretap on the unprotected buses. Hence, the secret identifier scheme does not degrade security.

[4] To avoid attacks on provisioning of the secret identifier, the user could do this from a secure remote device via remote attestation.

Chapter 5

Intel Software Guard Extensions

Intel Software Guard Extensions (SGX) add secure execution features to existing Intel x86 mainline CPUs. SGX provides a secure and verifiable execution environment, called *enclave*, that can be leveraged by an application to execute securely on the processor. Security covers privacy and integrity of data. SGX not only protects against hardware attacks on memory but also against attacks from privileged software. Even the operating system cannot access the protected enclave domain. Hence, the trust model of SGX only considers the CPU and the enclave as trusted while the OS is entirely untrusted.

Apart from the SGX programming reference [40], which is superseded by the software developer's manual [41], there exists other useful literature. Explanatory tutorial slides are available at [27]. A comprehensive analysis of SGX is given by Costan and Devadas in [28]. McKeen *et al.* give a quick overview of SGX features in [58]. Sealing and attestation is explained in [2]. Possible applications of SGX are outlined in [38]. A user guide for using the SGX evaluation SDK on Windows is given in [44].

The rest of this section shows how SGX achieves secure execution: First, we discuss SGX's application model and memory protection in more detail. Next, an overview of SGX instructions is given. In the following, we show how enclave startup, secure execution and dynamic memory management is done with these instructions. Furthermore, SGX key derivation is outlined to allow attestation and sealing. In the end, we give a short discussion on certain security aspects and elaborate on the business model of SGX.

5.1 Application Model and Memory Protection

In SGX, the enclave is part of an application process. The application is partitioned into an insecure non-enclave area and a secure enclave area, both sharing the same virtual address space. This partitioning has the advantage that only application's critical code needs to run within the enclave, having a minimal amount of software within the TCB.

Pages mapped within the enclave always belong to the so-called enclave page cache (EPC). The EPC is a contiguous part of RAM, which is protected by a hardware memory encryption engine (MEE). The MEE hardens the EPC against physical memory attacks. To defend bus probing attacks, it encrypts the EPC using AES in counter mode of operation. To defend bus forcing and memory replay attacks, the MEE does integrity protection using a MAC algorithm over

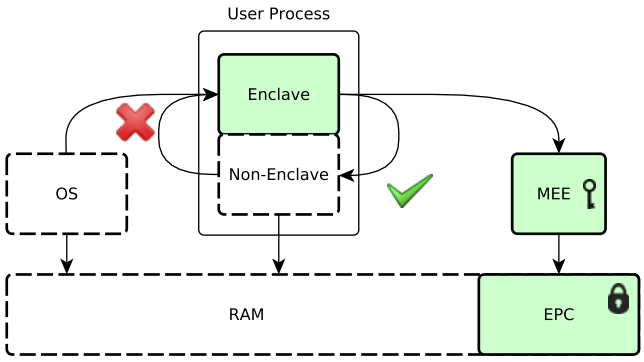

Figure 16: SGX memory protection: The Memory Encryption Engine (MEE) encrypts and integrity-protects the Enclave Page Cache (EPC). Enclave code can access non-enclave area while access into the enclave is prohibited from software running outside the enclave.

a multilinear universal hash function [27]. The MEE works transparently to the last-level CPU cache: memory writes are encrypted on-the-fly. Likewise, memory reads trigger verification and decryption of data from RAM.

Furthermore, EPC pages are protected against unauthorized software access from any privilege level. An application's enclave area can only be accessed from within the enclave. Any access from none-enclave code results in a fault signal. In contrast, enclave code can read and write non-enclave memory. Figure 16 gives an overview of SGX memory protection features.

The CPU does internal book-keeping in an Enclave Page Cache Map (EPCM) to associate EPC pages with enclaves. The EPCM holds a slot for each page in the EPC, as shown in Figure 17. This slot contains an enclave identifier as well as the virtual enclave address of the page, some status information and access permission flags. If the enclave identifier does not match the currently running enclave, access is denied. This restricts enclaves to access their own EPC pages only. Furthermore, by checking the virtual enclave address, SGX can verify if the untrusted OS has mapped the EPC page to the correct location.

5.2 Enclave Instructions

Intel SGX adds new instructions to the x86 architecture. These are divided into privileged enclave system (`ENCLS`) and unprivileged enclave user (`ENCLU`) instructions. `ENCLS` instructions are used for creating and maintaining enclaves and EPC memory, hence they are only accessible from protection ring-0. `ENCLU` instructions cover all necessary operations during enclave runtime: entering and

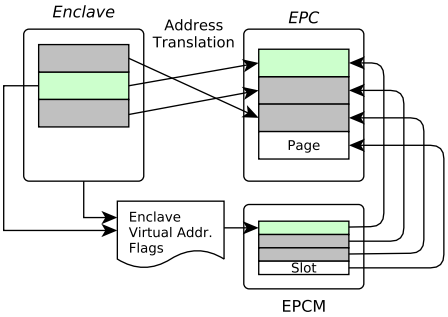

Figure 17: The Enclave Page Cache Map (EPCM) keeps track of all EPC pages. If an enclave page is resolved (green), the corresponding EPCM slot is queried. If the slot matches the currently running enclave and the virtual address, access is granted.

exiting an enclave, accepting new EPC pages from system software, attestation and key generation. `ENCLU` is only accessible to user mode applications (ring-3). Thus, an enclave cannot be run in kernel mode.

Table 2 lists all SGX user instructions. Table 3 gives an overview of SGX system instructions. The following section briefly explain the usage of most important instructions. For a detailed description refer to the SGX manual [40].

5.3 Enclave Startup

A new enclave is initialized with the `ECREATE` instruction. This reserves a contiguous part of the application's virtual address space as enclave region. The enclave

Table 2: SGX `ENCLU` instructions.

ENCLU	*Description*
EENTER	Enter an enclave
EEXIT	Exit an enclave
ERESUME	Resume an enclave from interruption
EACCEPT(COPY)	Accept EPC pages during enclave runtime
EREPORT	Report generation for attestation
EGETKEY	Query keys for report verification, sealing, provisioning

Table 3: SGX `ENCLS` instructions.

ENCLS	*Description*
ECREATE	Create a new enclave
EADD	Add EPC pages to enclave
EEXTEND	Measure added EPC pages
EINIT	Verify measurement and initialize enclave
EAUG	Add new EPC page during enclave runtime
EMODPE/R	Extend/Restrict EPC page permissions
EWB	Evict EPC page to memory
ELDB/U	Reload an EPC page from memory
EREMOVE	Remove an EPC page from enclave
EBLOCK	Block an EPC page for eviction
ETRACK	Check if TLB is ready for page eviction
EPA	Create version array to track evicted EPC pages
EMODT	Prepare for new enclave thread
EDBGRD/WR	Read/write into running debug enclave

region is filled with content by subsequent invocation of the `EADD` instruction. `EADD` maps a protected EPC page into the enclave region and initializes it with content from the non-enclave area. Furthermore, each EPC page is measured in blocks of 256 bytes, using the `EEXTEND` instruction. `EEXTEND` measures not only the content of an enclave page but also its relative position in the virtual address mapping.

Measurements are stored as a chained cryptographic hash in a measurement register, called `MRENCLAVE`. This is comparable to a Platform Configuration Register (PCR) in a Trusted Platform Module (TPM) [73]. If for any reason the enclave is not loaded exactly as intended by the enclave software vendor, `MRENCLAVE` will have a wrong value.

After loading the enclave, the instruction `EINIT` is invoked. This involves a security data struct, called `SIGSTRUCT`, which is signed by the software vendor. This data struct contains the expected enclave measurement and is used to verify the final value in the measurement register `MRENCLAVE`. If the verification process succeeds, enclave startup is completed and the enclave is ready to be used. If verification fails, SGX will refuse to run the enclave. The enclave startup procedure is summarized in Algorithm 1.

Algorithm 1 Enclave startup

```
ECREATE(enclave_region)
for all pages p do
  epc = AllocateEpcPage()
  EADD(p, epc)
  for i = 0 to 16 do
    EEXTEND(epc + i · 256)
EINIT(SIGSTRUCT)
```

5.4 Secure Execution

Integrity of code execution is crucial for security of the enclave. Therefore, SGX distinguishes between enclave mode and non-enclave mode. Switching between these two modes is restricted to certain instructions.

5.4.1 Regularly Entering and Exiting an Enclave. To switch to enclave mode, the instruction `EENTER` has to be executed from application's non-enclave code. Code execution is then started at a pre-defined point within the enclave. This entry point is specified during enclave initialization. Once in enclave mode, any direct jump to non-enclave code yields a fault. The enclave can be regularly left with the `EEXIT` instruction.

`EENTER` passes the caller's return address to the enclave. Enclave code has to manually store the caller's stack in order to set up its own enclave stack. Before invoking `EEXIT`, the enclave needs to restore the caller's stack. `EEXIT` takes an arbitrary non-enclave code address as argument. In order to resume non-enclave code where it has left, the enclave can provide the caller's return address, obtained from `EENTER`.

5.4.2 Asynchronous Exit. When executing in enclave mode, any interrupts such as page faults trigger an Asynchronous Enclave Exit (AEX). All CPU registers are automatically saved in enclave memory. The enclave has reserved some pages as State Save Area (SSA) for this purpose. The CPU register file is then masked to avoid information leakage to the OS. The OS interrupt handler is invoked to process the interrupt event. When finished, the interrupt handler issues the `IRET` instruction, which resumes operation at a pre-defined non-enclave trampoline code. The trampoline invokes `ERESUME` for re-entering enclave mode and restoring the original enclave register file from the SSA. Figure 18 shows a typical execution flow, where enclave code is interrupted by an AEX.

Note that typically an AEX needs an additional pair of `EENTER` and `EEXIT` to fix the cause for the AEX inside the enclave.

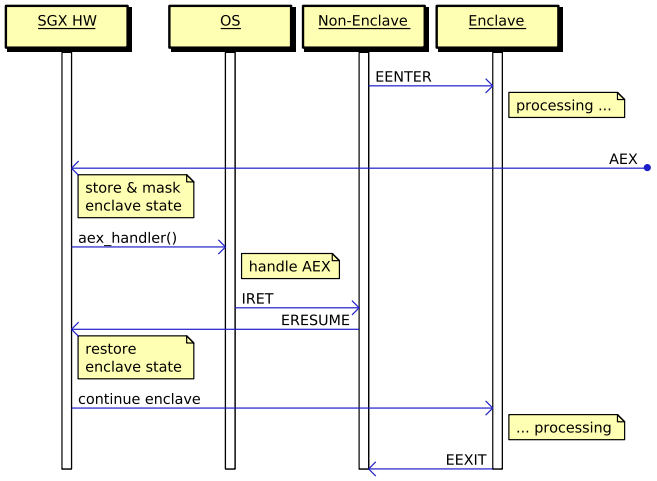

Figure 18: Enclave is entered and left via `EENTER` and `EEXIT`. An asynchronous enclave exit (AEX) event interrupts enclave execution. The CPU stores and masks the enclave state. It invokes the OS interrupt handler. If finished, the handler returns to non-enclave trampoline code. By issuing `ERESUME`, the enclave gets restored and continues its operation.

5.5 Dynamic Memory Management

Dynamic memory management was added to SGX in revision two, following recommendations of Baumann *et al.* [10]. This allows for dynamically growing enclave memory as well as for swapping out rarely used enclave pages.

Dynamic page mapping is done cooperatively between the untrusted OS and the enclave. The OS initiates mapping of a new EPC page by invoking `EAUG` instruction. In order to make use of the page, the enclave has to accept it using the instruction `EACCEPT`. Likewise, changes to the EPC page type (`EMODT`) or permissions (`EMODPR`) have to be acknowledged by the enclave via `EACCEPT`.

The OS can decide to evict an EPC page from the enclave and write it to (unencrypted) main memory using `EWB`, as a preparation for a swap out. `EWB` encrypts the EPC page while it is moved to unencrypted RAM. The OS can reload the EPC page again into the enclave by invoking `ELDB` or `ELDU`.

`EREMOVE` removes an EPC page from a running enclave. Removal or eviction of EPC pages needs no cooperation by the enclave because it does not directly break enclave's security. However, as we will see later on, this opens up a dangerous paging side-channel.

5.6 SGX Identities and Key Derivation

SGX knows two identities: the enclave identity, also denoted as EID, and the sealing identity. The EID represents the security state of the enclave after startup, including all measured enclave pages and some attributes. The EID is directly reflected by the measurement register `MRENCLAVE`. The sealing identity describes the software vendor which signed the enclave code. It is represented by the measurement register `MRSIGNER`. This register holds a hash of the vendor's public signing key.

SGX enables enclaves to derive keys for sealing, attestation and provisioning. Key derivation is done using the `EGETKEY` instruction. By making key derivation dependent on `MRENCLAVE` or `MRSIGNER`, derived keys can be bound to a specific enclave or a specific software vendor, respectively. Apart from `MRENCLAVE` and `MRSIGNER`, key derivation can include other values as well, as seen in Figure 19.

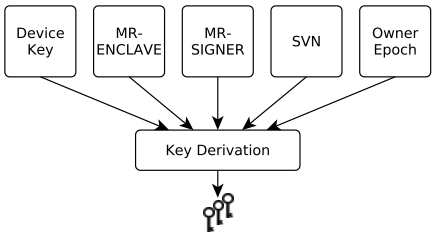

Figure 19: SGX key derivation.

All keys are device dependent, since key derivation includes a device-specific key. Furthermore, an owner epoch is merged into key derivation, allowing the device owner to revoke all derived keys by simply changing the owner epoch. By adding a Security Version Number (SVN) to key derivation, one can build an enclave version hierarchy, which is useful for sealing data.

5.7 Sealing

SGX allows enclaves to query seal keys. Seal keys can be used to encrypt sensitive data for persistent offline storage outside the enclave. By binding the seal key to `MRSIGNER`, all enclaves of the same software vendor can derive the same seal key to access encrypted data. When binding the seal key to `MRENCLAVE`, only the exact same enclave can access its sealed data. Alternatively, the seal key can also be derived from a Security Version Number (SVN) instead of the enclave identity. Thus, the seal key gets dependent on the current SVN. If a software

vendor releases an update to enclave code, the SVN is increased. This yields a different seal key. However, SGX permits decreasing the SVN when issuing `EGETKEY`. Thus, the new enclave can access data sealed by an old enclave but not vice versa. This allows migrating sealed data to the new enclave version.

5.8 Enclave Attestation

In order to assess the security of a running enclave, SGX provides methods for attestation [2]. Attestation comes in two ways: local and remote attestation. Local attestation allows an enclave to verify another enclave running on the same physical CPU. Remote attestation can be used by a remote party to check if the attested enclave is indeed running on a genuine Intel CPU and was not tampered with. Remote attestation thus allows initial provisioning of keys and secrets to an enclave.

5.8.1 Local Attestation. Enclave E wants to proof its authenticity to a target enclave T. Authenticity of an enclave depends on its value of `MRENCLAVE`, which reflects the enclave state after startup. If `MRENCLAVE` has a correct value, the enclave has been indeed loaded correctly. To do local attestation, enclave E generates a signed report over its `MRENCLAVE`. By verifying the report, the target enclave T can assess authenticity of E.

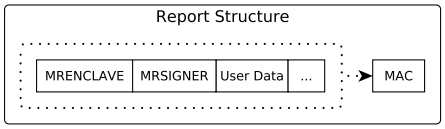

Figure 20: Report structure generated for attestation.

The report is a data structure, which is cryptographically signed. As seen in Figure 20, the report contains `MRENCLAVE`, `MRSIGNER`, some user-defined data and some additional fields. In order to create a report, an enclave issues the `EREPORT` instruction. SGX then derives the report key of the target enclave by calculating a SHA-256 over the target enclave's identity. Next, SGX signs the report by calculating a Message Authentication Code (MAC) over the report structure. It therefore uses AES-CMAC [71].

Algorithm 2 gives the steps involved in report generation. For the sake of simplicity, we denote EID of enclave X with X_{ID}, which equals `MRENCLAVE` of X. Note that we omit some implementation details which are not necessary for understanding attestation. For a detailed description, see the SGX programming reference [40].

Algorithm 2 $GenerateReport_{E_{ID}}(T_{ID}, userdata)$

Enclave E generates a report for attesting itself to target enclave T.
$userdata \in \{0,1\}^{256}$ is an arbitrary data blob which gets signed in the final report.
$E \rightarrow SGX : EREPORT_{E_{ID}}(T_{ID}, userdata)$
$SGX : reportkey_T \leftarrow SHA_{256}(T_{ID}||...)$
$SGX : mac \leftarrow AES\_CMAC_{reportkey_T}(E_{ID}||userdata||...)$
$SGX \rightarrow E : report = (E_{ID}||userdata||...||mac)$

The target enclave can verify the report by requesting its report key from SGX hardware with `EGETKEY` and manually recalculating the MAC. Algorithm 3 explains report verification.

Algorithm 3 $VerifyReport_{T_{ID}}(report)$

Target enclave T verifies a report generated by enclave E.
$T \rightarrow SGX : EGETKEY_{T_{ID}}(REPORT\_KEY)$
$SGX \rightarrow T : reportkey_T \leftarrow SHA_{256}(T_{ID}||...)$
$T : mac \leftarrow AES\_CMAC_{reportkey_T}(report.E_{ID}||report.userdata||...)$
if $mac = report.mac$ **then**
  T : accept $report$
else
  T : reject $report$

A MAC is a symmetric signature since the same key is used for signature creation and verification. Symmetric signatures do not provide non-repudiation. Since both, the signature creator (enclave E) and the verifier (enclave T) use the same key, one cannot distinguish, who actually created a signature. Thus, enclave T could forge reports targeted at itself. This is no problem in practice since enclave T has no benefit in doing so. However, as we will see later on, this symmetry of the report key can be exploited to do an efficient key exchange.

5.8.2 Remote Attestation. A remote verifier can use remote attestation to assess the security state of an enclave. Remote attestation involves multiple steps: First, the enclave of interest attests itself to a so-called quoting enclave. Then, the quoting enclave signs the report of the enclave being attested. Therefore, the quoting enclave owns a device key. The device key is an asymmetric, private signing key, which is bound to the CPU. Finally, the remote verifier can use the public verification key to verify the signed quote. Remote attestation is sketched in Figure 21.

Intel uses a variant of direct anonymous attestation to sign and verify quotes, called Enhanced Privacy ID (EPID) [15]. The verifier can only verify membership in a group, without being able to distinguish individual group members.

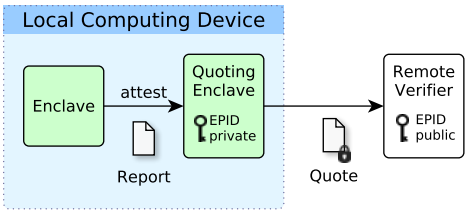

Figure 21: Remote attestation process.

During manufacturing, Intel installs unique keys on each CPU, identifying them as genuine. These keys are in turn used to derive attestation keys. The attestation key proves membership in the group of genuine CPUs while maintaining anonymity.

Official information about the quoting enclave and the provisioning of the device keys is relatively scarce [2,27]. By combining existing sources with some educated guesses, Costan *et al.* draw a more complete picture of remote attestation in [28].

5.9 Further Discussion

So far we have discussed the main features of SGX, which can be combined to build a secure execution environment. To clarify things, we shortly discuss security implications, if SGX is simulated. Afterwards, a powerful attack on SGX's dynamic paging support is outlined. In the end, we address Intel's business model behind SGX.

5.9.1 Simulation of SGX. Enclave startup is always done by the (untrusted) OS kernel, which issues the proper `ENCLS` instructions. During enclave startup, the whole enclave code is accessible to the OS in an unencrypted form. Hence, a malicious OS can easily choose to run enclave code in an SGX emulator. It can simply bypass any hindering SGX security checks, being able to inspect the enclave state over the whole simulation. This cannot be prevented by SGX at all. For this reason, IBM SecureBlue++ requires that an application binary is encrypted for the target CPU. Only the CPU knows the proper decryption key [13] and can actually execute the binary.

For SGX, simulation is no problem when adhering to the following rule: enclave code must never contain hard-coded secrets. The only way to provision secrets is via remote attestation. Remote attestation allows a remote verifier to check if the enclave is being simulated or not. Only if attestation succeeds, the remote verifier provisions secrets to the enclave. If attestation fails, it withholds secret data.

Remote attestation is based on genuine attestation keys, which are derived from the quoting enclave's device key. Since this key is protected by the CPU, the attacker has no access to it. Thus, the attacker fails to simulate remote attestation. For the same reason, the simulated enclave cannot decrypt sealed data which has been encrypted on a genuine SGX CPU, because it cannot derive the correct seal key.

Hence, an attacker can indeed simulate enclaves as long as no secret data is accessed. As soon as the enclave does remote attestation or sealing, the simulated execution deviates from the genuine one. The attacker does not obtain sensitive information.

Having enclave code in an unencrypted form opens up other use cases. It allows for example virus scanners to inspect enclave code before executing it.

5.9.2 Paging Side Channel. Xu *et al.* describe a powerful side channel which an untrusted OS can exploit to attack an SEE [78]. This also affects SGX. By allowing dynamic page mapping of enclave pages, SGX has an inherent weakness, leaking information from within the enclave. When an enclave accesses unmapped memory, a page fault is triggered. The OS is notified, where the page fault has occurred. Normally, the OS uses this information to map the missing page. However, it can also misuse it to profile memory access patterns of enclave code. In order to limit information leakage, SGX masks the lower 12 bits of the page fault address. Thus, the OS can only observe the page which caused the fault but not the exact virtual address.

Yet, this leads to a powerful attack. Since the untrusted OS is in charge of page mapping, it can control this side channel. By evicting an EPC page during enclave runtime with `EWB`, the OS can force a page fault as soon as the enclave tries to access the unmapped page. Then, the OS reloads the page again with `ELDU` to allow enclave code to resume. Although this only leaks information on a page granularity, Xu *et al.* exploit this side channel to do recovery of secret documents [78].

This side channel could be in principle mitigated by disabling `EWB` and `ELDU` for enclave pages. Alternatively, the enclave could be required to acknowledge `EWB`. Both is not possible in SGX. Hence, developers have to rely on mitigation mechanisms on application level. This essentially means that code has to be aligned in such a way that page faults do not leak sensitive information [26].

5.9.3 Intel SGX Business Model. SGX has an under-documented "feature", called launch enclave. During enclave initialization with `EINIT`, SGX not only checks `MRENCLAVE` and some other enclave attributes, as previously described. `EINIT` also involves a so-called `EINITTOKEN` which has to be issued by the launch enclave. Without a correct `EINITTOKEN`, the enclave is not initialized. Thus, the launch enclave is the only door to ever run custom-code enclaves on

the CPU. After digging deeper into some SGX patents, Costan *et al.* conclude that the sole purpose of the launch enclave is to enforce a licensing scheme for business users [28]. This apprehension has been confirmed with the release of the SGX evaluation SDK in January 2016 [44]. The SDK only supports launching enclaves in debug mode, that is, without full enclave protection. Production enclaves need to be licensed by Intel in order to get white-listed for the launch enclave [46].

However, the pricing schemes for licensing an enclave are not revealed. One can speculate that SGX is just the logical continuation of Intel's tradition in providing DRM for the well-paying film industry. Also PAVP, Intel Insider and WiDi are primarily targeted at building strong DRM into consumer devices. DRM is a lucrative business. The total loss among all US industries due to movie piracy has been estimated to $20.5 billion in 2006 [70]. Hence, it would not surprise if Intel reinforces this business model with SGX.

However, the enclave licensing scheme turns SGX rather useless for the open source community since Intel has an effective way of locking out any low-budget projects. Moreover, this undermines the freedom to modify open source enclaves at own will. As long as Intel does not publish further details on its pricing, one has to assume that Intel is not seriously interested in providing SGX for the broad community at all.

Chapter 6

seL4 Microkernel

The security kernel seL4 is the most trusted software in our secure I/O architecture. To better understand its role for secure port I/O, this section describes seL4 in more detail. First, we give some background information on seL4 and outline its design objectives and features. We then explain seL4's capability model in detail, followed by a brief discussion about the syscall API. Finally, we describe RefOS, an operating system running on top of seL4, that is used in our proof-of-concept.

L4 is a family of microkernels aiming at high-speed and security applications while maintaining a low memory footprint. Its success is highlighted by the wide deployment of OKL4, an L4 implementation by Open Kernel Labs. In 2012, OKL4 was used in almost 1.5 billion mobile devices [51]. To furthermore push development of high assurance microkernels, the NICTA group maintains a secure L4 kernel, called seL4 [60]. It is specifically designed to allow formal verification of correctness. In 2009, the seL4 implementation for ARMv6 could be formally verified, claiming to have "*the first formal proof of functional correctness of a complete, general-purpose operating-system kernel*" [48]. Klein *et al.* proved that the C implementation adheres to its specification, given that hardware, assembly code and the compiler are correct. Verification was only possible due to the small code base. seL4 consists of only "*8,700 lines of C code and 600 lines of assembler*", according to [48]. One can infer that also the x86 implementation, used in this thesis, is mostly correct as well since it shares all generic code with ARMv6.

A main design principle of microkernels such as seL4 is having the kernel as small and generic as possible with a small and defined syscall API. The majority of common operating system tasks such as resource management is exported to user processes. Thus, process and memory management, device drivers and even scheduling algorithms run in restricted user mode only. Due to seL4's strong process isolation mechanisms, both the kernel and any independent user process can be isolated from malicious or buggy drivers, which could otherwise take complete control over a system in a monolithic kernel design such as Linux.

Features. seL4 supports threads. Typically, each thread is assigned a separate virtual address space. By doing so, a single seL4 thread corresponds to an application process in Linux, for example. Hence, we use the term *process* to describe such an seL4 thread. Multi-threading is possible by assigning multiple seL4 threads the same virtual address space. Processes can interact via message passing. Therefore, seL4 supports asynchronous and synchronous endpoints. In

order to enforce certain security policies, the kernel implements access control mechanisms based on capabilities.

6.1 Capability Model

By owning a capability, also referred to as cap, a process is able to perform certain actions on defined objects. This includes thread management, page mapping, interrupt handling, I/O access and Inter-Process Communication (IPC), among others. Hence, device drivers, memory managers and likewise are implemented by assigning the correct capability.

seL4 caps can be dynamically migrated among user processes during runtime, at the discretion of the user process itself. This is referred to as discretionary access control. With such a generic approach one can encode various different security policies directly into applications.

Capabilities are protected and managed by the kernel, allowing user processes to invoke its caps only through defined syscalls. In seL4, the set of caps, assigned to a certain process, is stored in a capability space, called CSpace. Initially, when booting seL4, the first invoked user task has access to all root capabilities from which other caps can be derived. This root task is responsible for invoking other processes and migrating capabilities in accordance with a given security policy.

Capability Types. seL4 knows capabilities for all kinds of core resource handling tasks. Table 4 gives an overview of different types and their usage. There exist further capabilities dealing with large pages and virtualization, for example, which are not mentioned here.

Capability Migration. seL4 supports different methods for migrating capabilities among processes. This basically includes the operations *copy*, *mint*, *delete* and *revoke* and some variations of it. When copying a capability, the original cap remains unchanged and the child inherits its parent's capability, possibly restricted in read/write access rights. Minting allows copying and restricting the access range of the child. For example, the capability representing the whole RAM can be minted into two disjunct parts. The delete operation deletes a capability from the process' CSpace. Be aware that when deleting the last reference to a capability, it gets irreversibly lost. In contrast to the delete operation, a revoke deletes all children which were derived from a cap. To achieve this, seL4 maintains a capability derivation tree.

Note that certain types of capabilities have restrictions in allowed operations. Also, not all capabilities can be minted. For more details see the manual [61].

Capability Derivation Tree. seL4 keeps a record of all capabilities and its derivatives in a so-called Capability Derivation Tree (CDT). This CDT is used for revocation of capabilities. Each capability is a node in this global tree. When a cap is copied, minted or retyped, the new cap typically becomes a child node

Table 4: seL4 knows capability types for each kernel object. The types are grouped into capability and thread management, memory management as well as I/O and interrupt handling.

Cap Type	*Description*
CNode	Holds other caps in a list
CSpace	The thread's root CNode, holding all its caps
Domain	Allows creation of new threads (TCB objects)
TCB	The thread control block of a thread. Allows to run, suspend, resume a thread, manipulate its registers, change priority, etc.
Endpoint	Allows IPC between threads
Untyped	Represents unused physical memory or devices
PD	Page Directory allows to manipulate thread's virtual address mapping. Also called VSpace.
PT	Page Table
Page	A memory page frame or an IO frame, when derived from an untyped device region. Can be mapped into PT or IO PT.
IO Space	Represents a PCI hardware device in the IOMMU
IO PT	IO Page Table which can be mapped into IO Space
IO Port	Allows direct access to certain x86 I/O ports
IRQ Control	Allows creation of an IRQ Handler
IRQ Handler	Management and consumption of specific interrupts

of the original cap in the tree. When a process revokes a cap, the seL4 kernel traverses the CDT and deletes all derived children caps.

Typed Memory. To allow a coarse type-checking, all physical memory in seL4 is type sensitive. Physical memory is initially represented as a large, contiguous untyped cap. Untyped means that memory is not in use. This large untyped cap can be iteratively refined by retyping it to smaller untyped regions.

Furthermore, retyping allows to create specifically typed objects in an untyped memory region. These typed objects are again represented by capabilities. Note that by revoking an untyped memory cap, all typed objects, which were derived from it, are destroyed.

Figure 22 shows an example of a CDT, where a Thread Control Block (TCB), a page directory, a page table and some pages are derived from an initially large untyped memory block. Note that the CDT does not represent the virtual memory layout of a process. Rather, it tracks the hierarchy of caps in the system. In order to set up a virtual memory space, one has to map the page table and page caps appropriately into a page directory, that is assigned to a specific TCB.

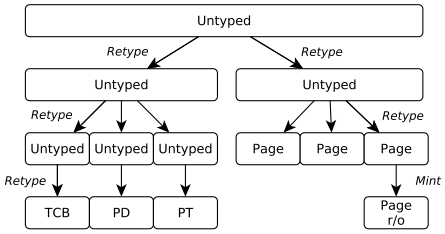

Figure 22: Example CDT with derived caps for the Thread Control Block (TCB), Page Directory (PD), Page Table (PT) and some pages. One page is in addition minted to read-only (r/o).

Typed objects can only be used in accordance to their type. Most objects cannot be accessed directly. Instead, a syscall has to be invoked. Thus, the kernel can restrict allowed operations on the object. The only object type, which an application can directly access, is a properly mapped page.

6.2 Syscalls

seL4 has a reduced syscall API. It only supports roughly three operations: sending messages, receiving messages and yielding. Table 5 lists the available syscalls[5].

As seen, most syscalls deal with message passing. Therefore, seL4 defines a lightweight message format. Short messages are directly passed via CPU registers, which makes message passing extremely fast. Longer messages are backed by a pre-defined message buffer.

When sending data to an endpoint cap, the message is delivered to the process, owning that endpoint. To wait for data to receive, a process issues the syscall `seL4_Wait` on its endpoint. Another process that has access to the same endpoint can then send data by issuing the syscall `seL4_Send` on the endpoint. That way, IPC is possible.

When sending data to a capability other than an endpoint, the message is delivered directly to the kernel. By coding additional tags and arguments into the CPU registers or the message buffer, the kernel can distinguish between different operations it shall carry out on that cap. For example, to map a new page, the caller sends a message to the PD cap alongside with a page and a tag indicating page mapping. If the sender expects some data to be returned, it can issue `seL4_Call` instead of `seL4_Send`. Thus, seL4 effectively enables a whole lot of

[5] Note that the table shows API version 1.2 [61], which is used in the proof-of-concept. In a more recent specification, `Wait` is renamed to `Recv` [62].

Table 5: seL4 syscalls.

Syscall	*Description*
seL4_Send()	Send a message and block until delivered
seL4_NBSend()	Send a message non-blocking
seL4_Wait()	Wait for a message from an endpoint
seL4_Call()	Send a message and block until received a reply. This is a combination of seL4_Send() and seL4_Wait(), using a special reply endpoint for the response.
seL4_Reply()	Reply to a seL4_Call(). This sends a response to the reply endpoint, received via seL4_Call().
seL4_ReplyWait()	Reply to a seL4_Call() and immediately wait for the next call.
seL4_Yield()	Give up remaining CPU time

different kernel calls via `seL4_Call`. We refer to those calls also as syscalls, since they interact with the kernel.

6.3 RefOS

RefOS is a small reference operating system, targeted at the seL4 kernel [67]. It runs entirely in the userspace. RefOS follows a multi-server design, splitting core OS features into several distinct servers. If one particular server for whatever reason denies its service, other independent servers and applications are not affected at all. Therefore, a multi-server design lowers the risk of total system failure by partitioning the attack surface among all servers. Thus, RefOS greatly complements seL4 in building a dependable system.

To understand the multi-server design, we first explain existing RefOS servers. Next, available RefOS libraries are given. Finally, we show how IPC is possible with RefOS.

Servers. A server is a separate process, implementing specific functionality. It is accessible through message passing. RefOS defines certain interfaces, which might be implemented by a server. These are:

- Name interface
- Server interface
- Data interface
- Proc interface

The name interface defines how servers can register itself for service discovery. This is needed such that applications can find other servers on the system. The

server interface has to be implemented by every server. It defines, how applications can interact with the server. The data interface maps the server's functionality to a so-called dataspace. The dataspace represents concrete data the server provides or consumes. For example, a dataspace can represent a region of memory, file content, a network interface, an attestation service, *etc.* Depending on the sort of data, the server might implement different dataspace handlers such as open, close, read, write, lseek, getc, putc, among others. The data interface is typically also implemented by every server. The proc interface defines functionality of the process server, which is the root task of RefOS.

Based on the clear interface definition, servers can be easily integrated into the multi-server design. RefOS already provides some basic servers, which implement core OS features. These are:

- Process server
- File server
- Timer server
- Console server

The `process server` is the root task which is initially spawned by seL4 on system startup. It invokes other servers and implements server discovery, memory management and process management. As such, the process server holds capabilities to all other processes' TCB, CSpace and page directory as well as to all system resources. Thus, the process server is the most privileged process running on top of seL4.

The `file server` implements a small file system, providing access to all executables. It is initialized from a static archive. When the process server loads new applications or resolves page faults, it queries the file server for retrieving the actual page content from the executable.

The `timer server` provides nanosecond-accurate timings. It therefore queries a hardware timer.

The `console server` provides I/O to the COM1 port. It is accessible via `stdin`, `stdout` and `stderr` file descriptors.

Libraries. RefOS comes with a basic set of libraries, accessible to applications and servers. These libraries help in implementing and accessing RefOS interfaces and provide high-level abstraction of some server functionalities. Table 6 gives an overview of the most relevant libraries.

IPC with RefOS Servers. In RefOS, the process server propagates its own endpoint cap to each newly launched process. Hence, any other process can contact the process server via this endpoint. Furthermore, the process server hosts a discovery service, where every other server registers under its name. If an application process wants to use a specific server, it queries the process server to resolve the server's name and establish a session to it.

Table 6: RefOS libraries.

Library	*Description*
libsel4*xxx*	Several seL4 libraries for syscall invocation, bootstrapping, thread management, memory allocation and capability management
librefos	Definition of RefOS interfaces and implementation of helper functions
libmuslc	Partial libc implementation for RefOS
libplatsupport	Access to platform specifics like I/O ports

To simplify this procedure, RefOS' standard library, the `libmuslc`, hides the functionality of server discovery in the POSIX filesystem. This allows to contact other servers by directly accessing a pseudo file, which identifies the service. A server can provide multiple services. The naming scheme for pseudo files is `"[server]/[service]"`.

For example, the console server is registered under the name `"dev_console"`. It provides access to the COM1 port under the service name `"serial"`. Listing 1 gives an example, how an application can send data to COM1.

Listing 1: Accessing COM1 via MUSLC.

```
fd = open("dev_console/serial", O_WRONLY);
if(fd < 0)
  error();
write(fd, "Hello world!", 13);
close(fd);
```

The console server implements the open and the write handler for the serial dataspace. The write handler eventually outputs data to COM1.

Chapter 7

Proof-Of-Concept Implementation

Our proof-of-concept implementation basically uses RefOS on top of seL4 to implement secure port I/O. This section first discusses the competing trust models of seL4 and SGX. This motivates our software architecture, which is given next. Then, we show how inter-process communication between the application and the I/O driver can be secured. Next, secure port I/O is discussed in great detail, explaining how we use our implemented `HasExclusive` syscall to build a transparent I/O binding. We also address shortcomings due to missing syscall support for SGX enclaves. Afterwards, we give our modifications to seL4 and RefOS, which are necessary to integrate support for SGX. Finally, we briefly explain OpenSGX, the SGX emulator which we adapt to work with the seL4 kernel.

7.1 Competing Trust Models

In order to correctly combine SGX and seL4, we have to understand the underlying trust models. After looking at seL4 and SGX, we motivate a sandwich trust model for our secure I/O architecture, as seen in Figure 23.

seL4 Trust Model. seL4 follows a classical, hierarchical trust model. Trust is extended from the bottom up. The seL4 kernel is the most trusted software. It invokes the root task which initially owns all capabilities. The root task furthermore distributes its capabilities among less privileged processes. Processes, which are higher up in the software stack, are more prone to errors since they have to rely on all software running below it. These processes are typically less trusted and therefore have less privileges (capabilities). Sensitive information of the user enters the kernel via I/O. It might be passed up the stack as far as trust is sufficient.

The root task, the most privileged user process, has to be ultimately trusted. Other tasks have to rely on the root task to do process launching and page mapping correctly, for example. Even if the root task would give away all of its capabilities to other processes, it might have already manipulated those processes beforehand. Hence, the hierarchical trust is a consequence from the fact that seL4 does not support verified launch of any process.

SGX Trust Model. SGX inverts the classical bottom-up trust model. The SGX enclave is a secure execution environment for user applications, with verified launch, secure page mapping and some other protection mechanisms. Thus, the enclave can protect itself from untrusted software and even from an insecure

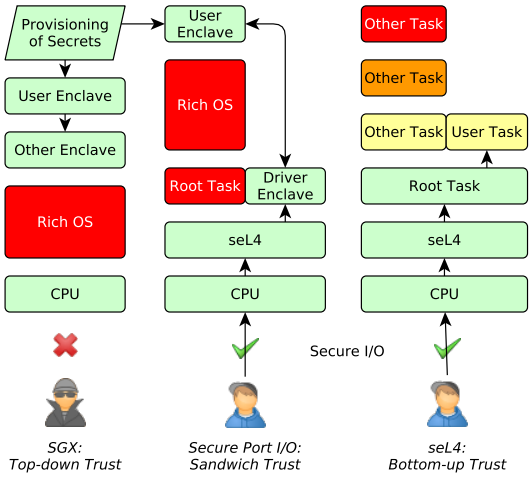

Figure 23: Competing trust models of SGX, seL4 and secure port I/O. Green indicates trusted components. Gradual blending to red shows increasing distrust. Sensitive information flow is depicted with arrows.

OS. Secrets are provisioned to the user enclave only from outside via remote attestation. These secrets might be propagated to other enclaves as well. However, they shall never leak to insecure software. Hence, in SGX trust is built top-down. Indeed, Intel SGX entirely distrusts the kernel. It does not only deny invocation of `ENCLU` instructions from kernel code but also prevents syscalls from enclave code. Furthermore, SGX distrusts the end user owning the computing device. Therefore, it does not provide secure port I/O features in the CPU.

Sandwich Trust. Both trust models do not fit secure port I/O in our mobile device use case. We combine them to get a sandwich trust, as seen in Figure 23. On the top, we trust the user enclave, which gets provisioned some secrets. On the bottom, we trust the driver, the seL4 kernel, the SGX-enabled CPU and the end user. Thus, sensitive information can transition between the end user at the bottom and user enclave at the top.

7.2 Architecture

For our architecture we implement a user application, which wants to securely communicate to the COM2 port, where a user has connected a legacy I/O device. Our architecture reuses existing RefOS servers and libraries. We add a RefOS server, called I/O driver, which has access to COM2. We furthermore add some libraries to support enclave management. The software stack is outlined in Figure 24.

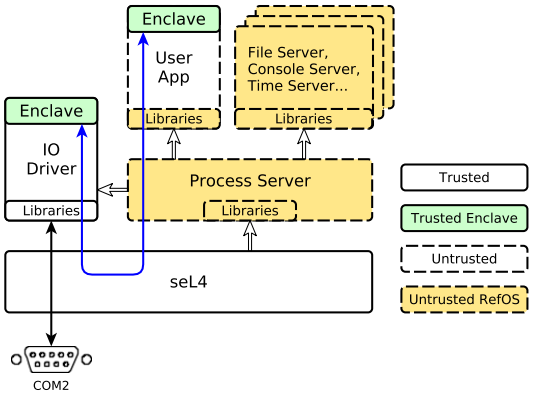

Figure 24: RefOS software stack: A user application communicates with the I/O driver (blue line) to securely access COM2.

The whole RefOS stack is considered as untrusted. The trusted software covers the seL4 kernel and the SGX enclaves within the user app and the I/O driver. We show that trust can be extended to non-enclave driver code as well.

Secure IPC. To get access to the COM2 port, the application has to talk to the I/O driver via some form of Inter-Process Communication (IPC). Since IPC involves untrusted RefOS code such as the process server, we implement a way of doing IPC securely.

Secure I/O Driver. The I/O driver implements secure port I/O. It has direct access to COM2, a platform-specific I/O port, which is intended for communicating with the user. The I/O driver is registered under the server name `"dev_secure"`. It provides direct write access to COM2 under the service name `"serial"`. Since this service is unprotected, it cannot be used for secure IPC but only for debugging purposes.

Table 7: Services provided by the I/O driver. Note that all services implement the open and close dataspace handlers.

Service	*Handlers*	*Description*
`serial`	putc, write	Direct output to COM2 (debug only)
`attest`	write	Secure, attested output to COM2
`eid`	read	Retrieve the driver's EID

To achieve secure IPC, the driver also implements two services, called `"attest"` and `"eid"`, which will be explained later on. Table 7 shows the driver's services and the RefOS dataspace handlers, which they implement.

SGX Libraries. To make use of SGX within RefOS, additional support is needed. Based on existing code from the OpenSGX emulator, we construct two SGX libraries for non-enclave and enclave code. The `libsgxnonenclave` provides support for building and maintaining an enclave from within non-enclave code. It integrates enclave startup procedures as well as some SGX wrappers for libc functions. The `libsgxenclave` implements SGX functions needed inside an enclave. Furthermore, it provides the counterpart of the SGX libc wrappers to allow access to certain libc functions from within the enclave. Note that `libsgxenclave` is direct part of the TCB since it runs within the enclave.

7.3 Secure Inter-Process Communication

In the following, we show how IPC is actually implemented in RefOS. Therefore we give a detailed flow through all untrusted software layers. To protect IPC over this untrusted software stack, we explain how SGX enclaves can be used to build a secure communication channel. Finally, we give our proof-of-concept implementation, in which we do secure IPC via an authentic channel.

7.3.1 IPC in RefOS. IPC between the application and the I/O driver is done via the RefOS data interface. The I/O driver implements a dataspace, which the application accesses. To find the driver, the application does service discovery via the process server. The process server then assists in building up a session between the application and the secure I/O driver. Once the session is established, the application can send data to or read data from the driver's dataspace. For IPC, the application has to leave the enclave and run through the whole non-enclave stack of support libraries.

Figure 25 shows an example flow how the application would send data to the driver for outputting it to COM2. Therefore, the application accesses the driver under the pseudo file `"dev_secure/serial"`. The `libsgxenclave` wraps the libc functions `open` and `write` and routes data to the application's non-enclave code. The `libsgxnonenclave` then invokes the corresponding `libmuslc` function. MUSLC resolves the driver's pseudo filename using server discovery, provided by the process server. It then passes data to IPC message handling of `librefos`. Note that a large chunk of data might be scattered upon multiple IPC messages. The `librefos` wraps data in an IPC buffer and `seL4_Send()` eventually traps to the seL4 kernel to transmit data to the I/O driver. seL4 wakes up the driver, which is listening for new input via `seL4_Wait()`. Data is unwrapped from the IPC buffer and passed to the dataspace `write_handler()`. This handler re-assembles the original data and stores it in a non-enclave buffer,

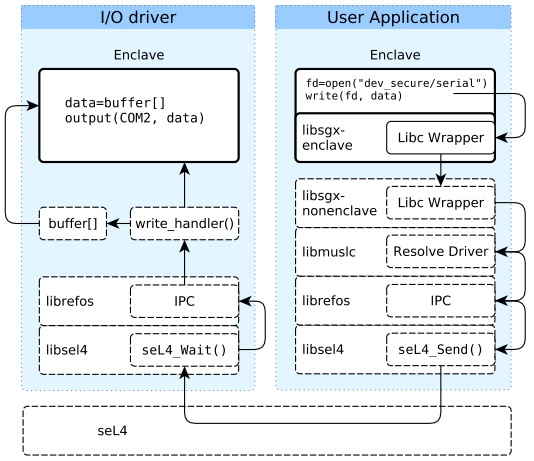

Figure 25: IPC has to go through a bunch of non-enclave libraries.

before invoking the driver's enclave. Finally, the enclave can access this buffer and output data to COM2.

One can see that IPC involves untrusted code at many layers, which does not allow to do IPC securely. For example, the untrusted process server is responsible for server discovery, establishing a session between application and driver. By manipulating this session, the process server could redirect all communication to itself and sniff or alter the traffic. Likewise, when loading the application, the process server or the file server could arbitrarily manipulate non-enclave library code to hijack IPC. To protect against such Man-In-The-Middle (MITM) attacks, we build a secure communication channel between the driver's and the application's enclave. This is achieved by leveraging SGX to do secure inter-enclave communication.

7.3.2 Secure Inter-Enclave Communication

is directly assisted by SGX's local attestation mechanism. With local attestation, one can set up an authentic communication channel. Depending on the use case, enclaves might be fine with integrity protection. However, the majority of security-critical applications also demands confidentiality of the communication. This requires exchanging key material to set up an encrypted communication channel. This section shows how both can be achieved by SGX local attestation.

Integrity Protection. Local attestation provides an integrity-protected communication channel out of the box. Enclave A, wishing to send an arbitrary document to enclave B in an authentic way, issues `EREPORT`. The report structure is able to hold 256 bit of user-defined data. Enclave A includes a SHA-256

hash over the document as user-defined data in the report structure. It sends the signed report to enclave B, alongside with the document. B then simply verifies the hash and the report. The whole procedure is listed in Algorithm 4.

Algorithm 4 Integrity-protected communication

Enclave A sends a document to enclave B in an authentic way.
$A : report \leftarrow EREPORT_{A_{ID}}(B_{ID}, SHA_{256}(document))$
$A \rightarrow B : (document||report)$
if $report.userdata \neq SHA_{256}(document)$ **then**
  B : reject *document*
else if $VerifyReport_{B_{ID}}(report)$ fails **then**
  B : reject *document*
else
  B : accept *document*

Confidentiality essentially requires exchanging a symmetric encryption key. If the symmetric key is established, enclaves should implement an authenticated encryption scheme[6] in software to maintain confidentiality and integrity of the channel.

There are different ways of doing key exchange between enclaves. We first describe Diffie-Hellman key exchange, to which Anati *et al.* refer in [2]. Also, the SDK user guide gives an example of how to implement Diffie-Hellman [44]. However, existing work does not investigate further on this topic. Hence, we also give a non-interactive, symmetric key transport scheme, which nicely exploits a single SGX local attestation procedure.

Diffie-Hellman key exchange [30] is straight forward, as shown in Algorithm 5. Since both parties influence the key material, it is also referred to as key agreement. Diffie-Hellman involves exponentiation of large numbers[7]. To avoid this, SGX local attestation can be used to directly implement a non-interactive, symmetric key transport, as shown in Algorithm 6. Since only one party influences the established key, it is called key transport. Our protocol is based on the symmetry of the target's report key. Any enclave can sign reports for a target by issuing `EREPORT`. Although it has no direct access to the target enclave's report key, it can indirectly use it via the `EREPORT` instruction. In turn, the target enclave is able to access its own report key. Thus, we already have a pre-shared secret from which other keys can be derived.

[6] Currently, authenticated encryption is achieved by using existing block ciphers in special modes of operation, like Galois Counter Mode (GCM) [57]. However, research seeks for new fast and dedicated authenticated encryption algorithms [17].

[7] NIST recommends to use 224 bit exponents and a 2048 bits modulus up to the year 2030 [9,12].

Algorithm 5 Diffie-Hellman key agreement

A symmetric key is established between enclave A and B.
Public parameters: $p \in \mathbb{P}, q \in [2, p-1]$
$A : r_A \xleftarrow{R} [2, p-1]$ chosen uniformly at random
$A : q_A \leftarrow q^{r_A} \mod p$
$A : report_A \leftarrow GenerateReport_{A_{ID}}(B_{ID}, SHA_{256}(q_A))$
$A \rightarrow B : (report_A || q_A)$
if $report_A.userdata \neq SHA_{256}(q_A)$ **then**
  $B : abort()$
if $VerifyReport_{B_{ID}}(report_A)$ fails **then**
  $B : abort()$
$B : r_B \xleftarrow{R} [2, p-1]$ chosen uniformly at random
$B : q_B \leftarrow q^{r_B} \mod p$
$B : key \leftarrow q_A^{r_B} \mod p$
$B : report_B \leftarrow GenerateReport_{B_{ID}}(A_{ID}, SHA_{256}(q_B))$
$B \rightarrow A : (report_B || q_B)$
if $report_B.userdata \neq SHA_{256}(q_B)$ **then**
  $A : abort()$
if $VerifyReport_{A_{ID}}(report_B)$ fails **then**
  $A : abort()$
$A : key \leftarrow q_B^{r_A} \mod p$

Algorithm 6 Non-interactive, symmetric key exchange

Enclave A sends a symmetric key to enclave B.
$A : nonce \xleftarrow{R} \{0,1\}^{256}$ chosen uniformly at random
$A : key \leftarrow GenerateReport_{A_{ID}}(B_{ID}, nonce).mac$
$A \rightarrow B : (A_{ID}, nonce)$
$B : reportkey_B \leftarrow EGETKEY_{B_{ID}}(REPORT\_KEY)$
$B : key \leftarrow AES\_CMAC_{reportkey_B}(A_{ID} || nonce || ...)$

GenerateReport eventually does the same AES-CMAC operation as enclave B but with the distinction that enclave A has no direct access to the $reportkey_B$. Since the generated report is never transmitted, the report's MAC can serve as shared, symmetric encryption key.

This non-interactive key exchange has zero overhead, compared to local attestation. It is not only fast but has also a negligible memory footprint. Report generation and key derivation is done by SGX hardware, and the AES-CMAC implementation is typically already part of enclave code for doing local attestation.

7.3.3 Proof-Of-Concept. Our proof-of-concept implementation demonstrates the interaction between application and driver with a simplified protocol.

Instead of implementing the full mutual authentication and channel encryption, we stick to a one-way authentication. Therefore we use local attestation to build an integrity-protected channel, similar to Algorithm 4. This is enough to demonstrate the feasibility of secure port I/O with SGX and seL4.

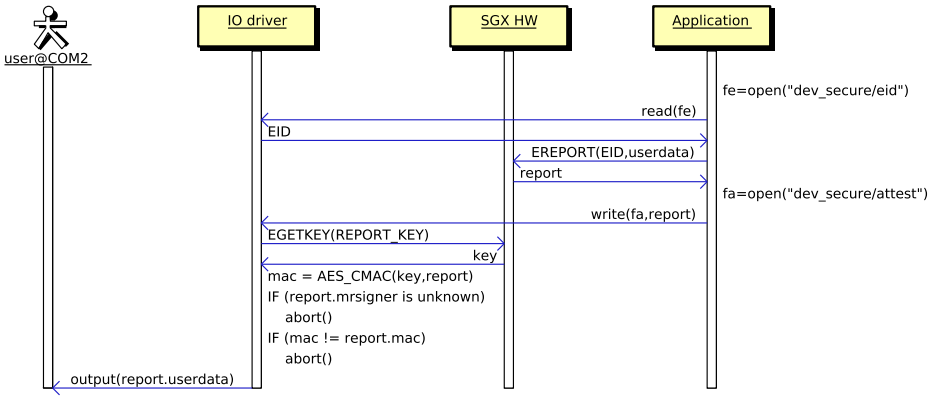

Figure 26: Proof-of-concept: Integrity-protected secure output is achieved with local attestation.

The application wants to output a short hard-coded message to COM2 in an authentic way. The message has a maximum length of 32 bytes and does not need to be kept private. The protocol works as follows: The application attests itself to the I/O driver. Therefore, it first queries the Enclave ID (EID) of the driver. The driver makes its EID (the value of `MRENCLAVE`) available under the service name `"eid"`. Then the application generates a report, targeted at the driver's EID, and includes the message in the report's user data field. Thus, the report does not only authenticate the application itself but also the message. Now, the application sends the report to the driver. Therefore, the driver implements an attestation service under the service name `"attest"`. When receiving a report at this service, the driver first checks if the application is white listed. It therefore verifies the software vendor (the value of `MRSIGNER`) stored in the report. Next, it recalculates the report's MAC with its own report key. If the MAC matches, the driver outputs the user data to COM2, as shown in Figure 26.

Note that hard-coding user data into the application only demonstrates functionality. Since our implementation just requires authenticity of this user data, confidentiality is no issue. In practice, when dealing with confidential data, this data has to be provisioned to the application via remote attestation. Furthermore, a confidential communication channel, as described before, is required.

7.4 Secure Port I/O

Secure port I/O requires a transparent I/O binding between driver and COM2 port. The COM2 port is accessible at address range `0x2F8 ... 0x2FF`. We first describe our new `HasExclusive` syscall which we implemented to achieve transparency of the I/O binding. Then we give the necessary steps to do I/O binding with seL4. Afterwards, we pay special attention to the role of SGX in protecting the driver's code base. Unfortunately, SGX does not support syscalls from enclave code. Hence, we finally show ways to get around those shortcomings.

7.4.1 HasExclusive Syscall. We implement a new seL4 syscall, called `HasExclusive`. This syscall takes as argument a capability. It returns the number of capabilities that overlap the region of the given capability.

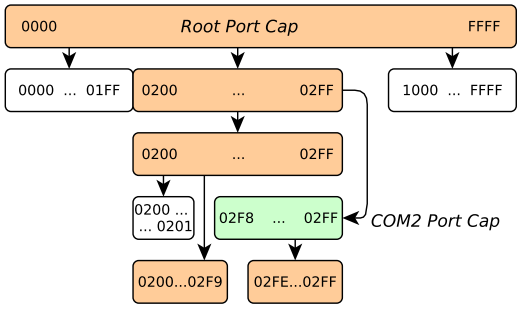

Figure 27: Example CDT showing all I/O port capabilities. All colored caps overlap with the COM2 port cap.

In order to achieve this, the syscall traverses the Capability Derivation Tree (CDT) and checks both, parents and children of the queried cap. It only counts those caps which have the same type as the queried capability and overlap with it. In Figure 27, an example CDT is given. Here, `HasExclusive` has been invoked on the COM2 port cap. The overlapping caps are colored. Note that also the original COM2 port cap is counted. In this case, the syscall returns the number 6. If one would invoke `HasExclusive` on the root port cap, it would return 9 overlapping caps.

If the syscall returns 1, the capability is indeed owned exclusively by the caller. Note that this syscall can also be used to probe if a given capability is empty, in which case 0 is returned.

Implementation Notes. To count the number of overlapping caps, the syscall traverses the CDT in both directions, starting at the queried cap. The CDT is

internally stored as an ordered tree. Hence, we can use this ordering and stop traversal as soon as a neighboring cap does not overlap anymore or has a different type.

We implement this syscall especially for testing I/O port capabilities. The implementation could be also extended to other cap types. However, when considering caps which were retyped from untyped memory, such as memory or I/O pages, special care must be taken. In that case, one cannot only check against neighboring caps in the CDT but also against the untyped memory, from which the tested cap is derived. Otherwise, a process might falsely assume that it has exclusivity over the cap forever. Consider a process having exclusivity over an I/O page. A malicious process, owning the original untyped memory chunk, could revoke this untyped memory in which case the I/O page is destroyed. Then it could retype the I/O page for itself.

7.4.2 Transparent I/O Binding with seL4. When the I/O driver process is loaded, it has to establish a transparent I/O binding to the COM2 port. We briefly recap the requirements of the I/O binding, before explaining, how each one is achieved with seL4.

- I/O binding
  - Exclusive ownership of I/O resource
  - Non-revocability of this ownership
- Transparency
  - Identification of I/O resource
  - Verification of the I/O binding

Exclusive Ownership. The process server initially owns a root I/O port capability covering all I/O ports. By minting this root cap to the COM2 range, we get a capability, directly representing COM2. The process server assigns this COM2 cap to the driver. In order to ensure that the driver has exclusivity over COM2, the process server removes the COM2 address range from its root I/O port cap.

Non-Revocability. When having exclusive ownership over COM2, the driver is the only piece of software (apart from seL4) that has control over COM2. However, seL4 knows some capabilities that can be misused to impersonate a process and act on its behalf. These are the CSpace, the TCB and the PD capability.

The CSpace cap represents all capabilities of a certain process. It allows invocation and manipulation of any capability of that process. By owning the driver's CSpace, one could simply duplicate the driver's I/O port capability for own usage. To remain stealthy, one can delete the duplicate before resuming the driver.

The PD cap represents a process' page directory. It is used to map pages into its virtual address space. When owning the driver's PD cap, an attacker could map malicious code into the driver. Thus, the driver can be forced to duplicate the exclusive I/O port cap and migrate the double to the attacker.

The same can be achieved with the TCB cap. The Thread Control Block allows to modify the register set of a process. Thus, an attacker, who has access to the driver's TCB cap, could manipulate the instruction pointer to do Return-oriented Programming (ROP) [16]. If the driver's code base is large enough, it is likely to have enough gadgets for arbitrary code execution on the driver's behalf.

To prevent such impersonation attacks, the driver not only requires exclusivity over COM2 but also over its own CSpace, TCB and PD cap. To achieve this, the process server first loads the complete driver into RAM. Therefore it creates a new TCB for the driver, assigns it a fresh CSpace and PD, migrates necessary caps to the driver and sets up its virtual memory. Then, the process server uses the TCB cap to launch the driver. If the driver is running, it notifies the process server to do a `LockOut`, as shown in Algorithm 7. `LockOut` means that the process server deletes all its copies of the driver's CSpace, TCB and PD cap.

Algorithm 7 LockOut()

```
Delete all references to the driver's TCB, CSpace and PD cap.
DeleteCap(TCB_driver)
DeleteCap(CSpace_driver)
DeleteCap(PD_driver)
```

Note that, if the process server is locked out, the driver needs to cooperate on unloading. The process server can kill the driver by revoking the untyped memory region from which the driver's TCB has been derived. However, when doing so, the whole driver is destroyed, including its CSpace. By destroying the driver's CSpace, the COM2 cap gets irreversibly lost and COM2 remain inaccessible to the system until the next reboot. Hence, the driver has to freely migrate the COM2 cap back to the process server before getting killed.

However, this is no loss of generality. Also, existing monolithic kernels such as Linux have to rely on drivers to properly free certain resources, when unloading.

Resource Identification. Identification is implicitly provided, since the driver knows the address range of the COM2 port. The driver accesses COM2 by invoking `seL4_Call` with the COM2 port cap and the COM2 port address `0x2F8`, for example. seL4 then checks if the port address matches the cap. On a match, I/O succeeds. On a mismatch, seL4 returns an error.

Alternatively, the driver could identify an I/O port cap without accessing it. This is achieved by minting the COM2 cap to the address range `0x2F8 ... 0x2FF`. If

minting succeeds, the COM2 cap indeed covers the minted address range. The minted copy can be safely deleted afterwards.

Verification of I/O Binding. To verify if the process server indeed removed its reference to the COM2 cap and the driver's TCB, CSpace and PD caps, the driver invokes the `HasExclusive` syscall on those caps. Note that the seL4 kernel itself maintains a copy of each process' CSpace and PD cap. Thus, `HasExclusive` on these caps will always return at least the number 2. The verification of the I/O binding is summarized in Algorithm 8.

Algorithm 8 VerifyBinding()

```
1: Return true if the I/O binding is exclusive and non-revocable.
2: io ← HasExclusiveCap(COM2)
3: tcb ← HasExclusiveCap(TCB)
4: cs ← HasExclusiveCap(CSpace)
5: pd ← HasExclusiveCap(PD)
6: if io ≠ 1 or tcb ≠ 1 or cs ≠ 2 or pd ≠ 2 then
7:     return false
8: else
9:     return true
```

Verification of Driver Code. Once `VerifyBinding` succeeds, the driver has exclusivity over its own CSpace, TCB and PD. Hence, the driver's caps are shielded against all other processes running on top of seL4. The driver can do secure I/O with the COM2 port. Moreover, we have effectively created a secure execution environment for driver's non-enclave code. This is necessary because seL4 binds capabilities to the whole driver process, not just the driver enclave.

Since the driver process is bootstrapped by untrusted code, we have to ensure that `VerifyBinding` is tamper resistant. Otherwise, the process server could simply disable the check in `VerifyBinding` while loading the driver, for example. Hence, we protect `VerifyBinding` by an SGX enclave.

Moreover, the driver enclave needs to verify driver's non-enclave code before ever interacting with it. Doing so is comparable to verified launch. Verification could be done by calculating a chained hash over all non-enclave code and data, similar to SGX measurements done with `EADD`. This is possible since the enclave can access non-enclave memory. The expected hash could be stored in enclave code, which is protected by SGX. Thus, trust is effectively extended to the whole driver.

To sum up, a proper driver startup involves establishment of an I/O binding and verification of this binding from within the driver's enclave. Furthermore, secure execution can be extended to the whole driver process by verifying driver's

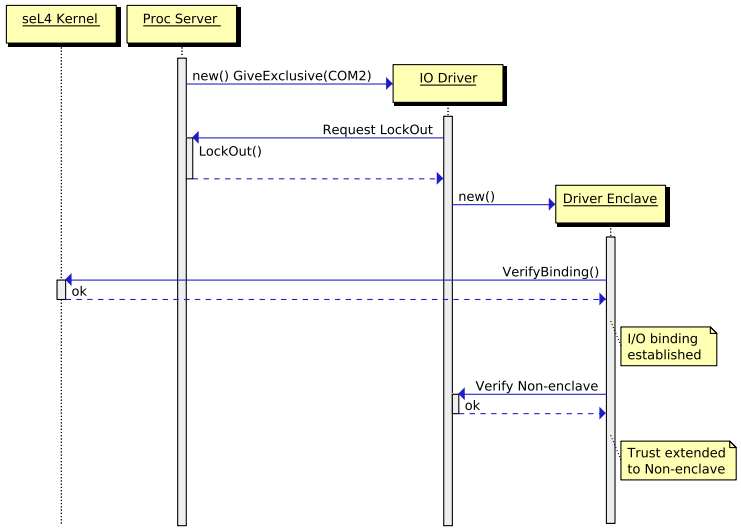

Figure 28: Driver startup involves establishing an I/O binding and verifying it. Furthermore, the driver enclave can extend trust to non-enclave code.

non-enclave region from within the enclave. The whole process is outlined in Figure 28.

Towards Atomicity of VerifyBinding. `VerifyBinding` is non-atomic, even when executed within an enclave. Consider that the process server has loaded the driver correctly but withholds a copy of the CSpace cap. If it could manage to interrupt the enclave after line 3 of Algorithm 8, the TCB check will succeed. The process server could then use its CSpace cap to steal the driver's TCB cap. Before resuming the driver, it deletes its copy of the CSpace cap. Thus, all subsequent HasExclusive checks also succeed. The driver verifies the binding correctly although the process server is in possession of the driver's TCB cap.

However, the process server cannot directly interrupt enclave code at any specific position. SGX disables hardware breakpoints for production enclaves. Hence, the process server would need to rely on page faults or probabilistic timer interrupts or similar. To avoid page faults during verification, the driver should page-align `VerifyBinding`. To get rid of the probabilistic sources of interrupts, the driver could call `VerifyBinding` multiple times.

However, the preferred way of mitigation is to make `VerifyBinding` atomic by shifting it into the seL4 kernel. Therefore, we introduce a new syscall, named `HasLockOut`, which consolidates the `HasExclusive` checks for the driver's TCB, CSpace and PD cap, as shown in Algorithm 9.

Algorithm 9 HasLockOut(TCB, CSpace, PD)

```
Return true if calling process has exclusivity over its TCB, CSpace and PD cap.
if TCB, CSpace or PD do not belong to calling process then
    abort()
tcb ← HasExclusiveCap(TCB)
cs ← HasExclusiveCap(CSpace)
pd ← HasExclusiveCap(PD)
if tcb ≠ 1 or cs ≠ 2 or pd ≠ 2 then
    return false
else
    return true
```

7.4.3 Enclave Syscalls. In order to successfully combine SGX with secure port I/O, one requires to execute the `HasLockOut` syscall from driver's enclave code. Unfortunately, this is not possible in SGX. When issuing `SYSCALL`, `SYSENTER`, `INT` or similar instructions within an enclave, the CPU throws an invalid opcode exception. Syscalls can only be invoked by a detour to non-enclave code. In general, SGX does not allow enclaves to directly exchange information with the kernel. Each interaction has to be mediated by non-enclave user code. This shows a fundamental incompatibility of SGX's trust model with our sandwich trust model. Hence, in our proof-of-concept, the driver implementation does not protect `VerifyBinding` by an enclave. Instead, it is executed in non-enclave driver code, knowing that this does not protect against a malicious process server. In the following we show how one could exploit a covert channel to build a direct communication path between enclave and kernel. This would allow to verify the binding within the enclave again.

Exploiting SGX Page Handling. Although SGX provides no defined path for enclave-to-kernel communication, one can exploit paging side channels to do so. By accessing an unmapped EPC page, a page fault occurs. Thus, an enclave can send information about which exact page faulted to the kernel. To get a channel from the kernel back to the enclave, one can also exploit dynamic page management. The kernel can issue `EMODPE` to extend page permissions of an already mapped EPC page. Then, the enclave issues `EACCEPT` to accept these permission changes. If the page protection flags of `EMODPE` and `EACCEPT` do not match, `EACCEPT` returns an error code. With each call to `EACCEPT`, the enclave can derive one bit of information about the page protection flags. Hence, the kernel and the enclave can agree on certain semantics of the page protection flags to exchange data.

Though this procedure might seems complicated, it only needs to be done once to signal the enclave a single bit of information, namely whether `HasLockOut` succeeded or not. This bit indicates if the process server correctly did `LockOut` and the driver has exclusivity over its own TCB, CSpace and PD. As soon as this is the case, the I/O driver can extend trust to driver's non-enclave code,

as previously described. All subsequent syscalls can be safely delegated to the verified non-enclave region.

To implement this, one could add a syscall wrapper, called `HasLockOutWrapper`, as given in Algorithm 10. This syscall wrapper issues `HasLockOut`, as described before. Since `HasLockOut` verifies if the passed TCB, CSpace and PD caps belong to the calling process, the wrapper can be safely issued from driver's non-enclave code. Furthermore, `HasLockOutWrapper` takes the address of a read-only enclave page, which serves as test page. It codes the result of `HasLockOut` into the write permission of the test page by issuing `EMODPE`.

Algorithm 10 HasLockOutWrapper(TCB, CSpace, PD, test_page)

```
Signal an enclave if its process has exclusivity over its TCB, CSpace and PD cap.
This is done by mapping an EPC test_page either with or without write permission.

if HasLockOut(TCB, CSpace, PD) = true then
  permissions.write = true
else
  permissions.write = false
permissions.read = true
permissions.execute = false
EMODPE(epc_page, permissions, test_page)
```

Algorithm 11 EnclaveVerifyLockOut(test_page)

```
Verify if the write permission of the EPC test_page is set or not. Return true if the
calling enclave has indeed exclusivity over its TCB, CSpace and PD cap.
permissions.read = true
permissions.write = true
permissions.execute = false
result = EACCEPT(permissions, test_page)
if result = SGX_PAGE_ATTRIBUTES_MISMATCH then
  return false
else
  return true
```

Now the driver enclave can verify if the write permission is true. This is done by issuing `EACCEPT` with the expected permissions. If there is a mismatch, `EACCEPT` fails and the enclave knows that some other process still has access to it. The verification is shown in Algorithm 11.

7.5 SGX Integration

The seL4 kernel does not support Intel SGX out of the box. This section describes necessary changes we implemented to make SGX known to seL4. First, the seL4 kernel has to be aware of the Enclave Page Cache (EPC). Therefore, we change bootstrapping of seL4 in order to propagate the EPC up to the process server. Furthermore, we modify the process server to allow allocation of EPC pages. Next, we implement a generic way to bootstrap enclaves within RefOS. Finally, we add a new capability to seL4, which allows issuing of privileged SGX instructions (`ENCLS`) from user processes.

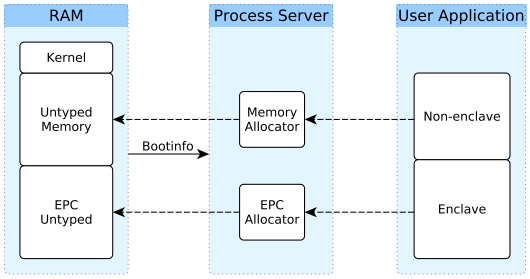

Figure 29: EPC memory is handled by a special memory allocator in the process server. The dashed arrows show how memory is acquired.

Bootstrapping the EPC. The EPC is a contiguous region of memory, which enjoys special access protection. The EPC region is specified as Processor Reserved Memory (PRM) during system boot. The BIOS therefore initializes certain region registers of the PRM properly. To simplify matters, the OpenSGX emulator we use allows specification of the EPC region via a special CPU instruction. Hence, we initialize the EPC region from RefOS code rather than from the BIOS.

To make the EPC region accessible to software, we have to dig into the seL4 boot process. On system startup, seL4 partitions memory into a kernel-reserved region and untyped memory, usable by RefOS processes. To make seL4 aware of the EPC, we split this untyped region into an EPC untyped region and normal untyped memory. The EPC region is propagated to the root task (process server) via an architecture-specific `bootinfo` structure.

The process server already implements a memory allocator, which manages all untyped memory. If an application claims more memory, it can acquire it from this allocator. The allocator then retypes new pages from untyped memory. We reuse the exiting allocator implementation to also provide an allocator for EPC

memory. This allows an application to acquire EPC pages. Figure 29 gives an overview of how EPC memory is propagated up to user processes.

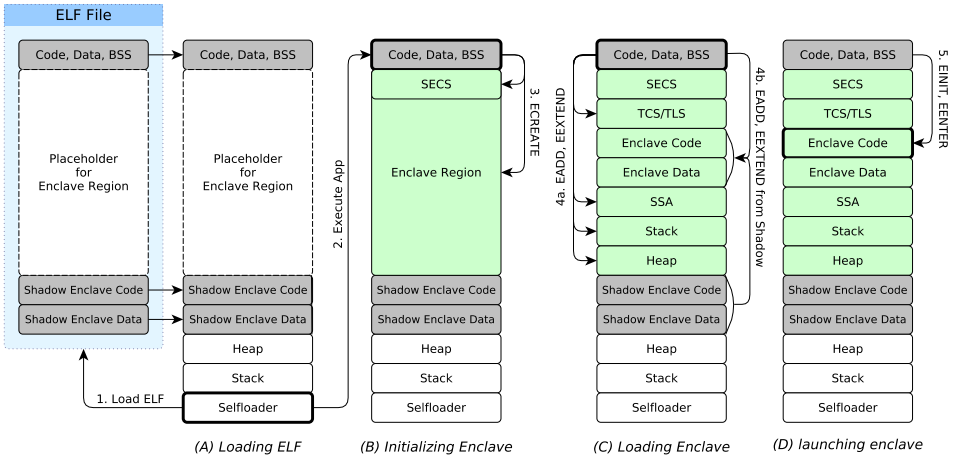

Figure 30: Bootstrapping an enclave involves several steps: After loading the ELF file via a selfloader, the startup code initializes and loads the enclave region before giving control to the enclave. Execution flow is marked with bold frames.

Bootstrapping an Enclave. A RefOS application is stored in an ELF file [56]. For enclave applications, the ELF binary contains both, non-enclave code/data and enclave code/data. The ELF file is statically linked. Hence, enclave code and data is linked for the target address within the enclave region, at which the enclave will be executed. However, to ease the process of enclave startup, the whole enclave's virtual region is spared out in the ELF binary. We shift enclave code/data to a higher address range in the ELF file. This shifted region is never executed but serves as a shadow region for initializing the actual enclave.

Enclave loading involves several steps, as seen in Figure 30. It basically follows the enclave startup procedure, described in Section 5.3. In RefOS, new applications are loaded by an in-application selfloader.

Step 1: The selfloader loads the ELF content (non-enclave and enclave shadow) into its own virtual address space.

Step 2: It gives control to the application, which invokes enclave startup routines from `libsgxnonenclave`.

Step 3: Startup code first reserves an EPC page for the SGX Enclave Control Structure (SECS). The process server assists in allocating EPC pages and mapping

them into the application's enclave region. Then, `ECREATE` is invoked, which reserves the enclave region and initializes the `SECS`.

Step 4: Again, startup code reserves EPC pages for the Thread Control Structure (TCS), Thread Local Storage (TLS), enclave code and data, State Save Area (SSA) and the enclave stack and heap. By executing `EADD`, it fills the enclave region with content. The TCS page is initialized to hold the enclave entry point and to keep track of the SSA. Enclave code and data pages are initialized from the enclave shadow region. Other EPC pages (TLS, SSA, stack, heap) are initialized with zeros. `EEXTEND` measures all initialized EPC pages. See the SGX manual for details about TCS, SSA and SECS [40].

Step 5: The enclave is verified and finalized by `EINIT`. It can be executed with `EENTER`.

Note that `EINIT` involves a `SIGSTRUCT` for verifying correct loading of the enclave. For our proof-of-concept we generate this `SIGSTRUCT` dynamically during enclave loading. For real-world usage, the enclave software vendor would create the `SIGSTRUCT` during compile time and link it to the ELF file.

To reduce the memory footprint, one could unmap the enclave shadow region as well as the selfloader when enclave bootstrapping is complete.

SGX Capability. All processes running on top of seL4 execute in protection ring-3. However, SGX requires ring-0 for special, privileged instructions, called `ENCLS`. In order to execute `ENCLS` instructions from RefOS processes, running in ring-3, we add an SGX capability to seL4. Any process, owning the SGX cap, can instruct the seL4 kernel to execute `ENCLS` instructions in ring-0.

7.6 OpenSGX QEMU

We ran our implementation on OpenSGX, a QEMU fork. OpenSGX provides a rudimentary implementation of Intel SGX and is available at [74]. QEMU is a machine emulator, supporting a lot of different CPU architectures like x86 and ARM, among others [66]. QEMU comes in two forms: as a Linux emulator and as a system emulator. The Linux emulator simulates the Linux syscall API to run a single program. In contrast, the system emulator directly simulates the CPU. Hence, it is used to run a complete kernel. For optimizing execution speed, QEMU translates simulated code to host code before executing it. Furthermore, in system emulation mode, QEMU uses a soft-MMU for caching translations from guest virtual to host virtual addresses, similar to a translation lookaside buffer. This reduces the number of time-consuming guest address translations, see Figure 31.

By the time of writing, OpenSGX does not support full system emulation but only Linux syscall emulation, see also [45]. However, in order to run the seL4 kernel, we require full system emulation. Hence, we reworked most of

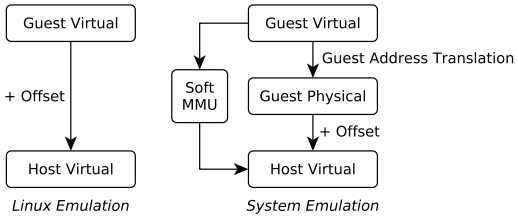

Figure 31: In Linux emulation mode, QEMU maps guest virtual to host virtual addresses by adding a constant offset. In system emulation, translation involves ordinary guest address translation via page directory and page tables, before mapping it to host memory.

OpenSGX's memory accesses to fit the soft-MMU translation process. Also, original OpenSGX library code does not support x86 32-bit compatibility mode, also referred to as i386. However, seL4 contains 32-bit assembler code, especially for system startup and interrupt handling. Hence, we migrated OpenSGX to i386 in order to emulate seL4. Furthermore, OpenSGX has some incompatibilities with the SGX manual regarding SSA, error handling and other implementation details.

Chapter 8

Further Discussion

We have seen how secure port I/O can be done with SGX in conjunction with the seL4 security kernel. In this section we explain memory-mapped I/O as a common alternative to I/O ports. Then we discuss possible hardware modifications for making the integration of secure port I/O easier. Finally, we explain one modification in detail, showing how an I/O binding can be directly assisted by SGX hardware. This comes without the need for a security kernel.

8.1 Memory-Mapped I/O Binding

Nowadays, most devices are accessed via Memory-Mapped I/O (MMIO) rather than I/O ports. We discuss MMIO from an enclave's perspective and outline how one can create an MMIO binding with seL4.

I/O ports are accessible via a separate I/O bus. In contrast, an MMIO device is mapped into the same address space as RAM. Thus, MMIO devices are subject to the same address translation process as memory pages. Once mapped into a process' virtual address space, the MMIO device can be directly addressed.

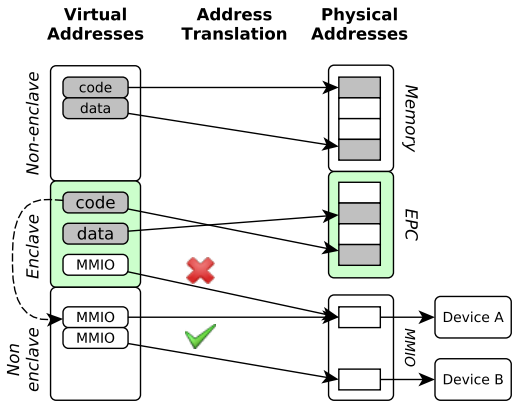

Figure 32: An enclave can access memory mapped I/O devices only via the non-enclave region.

MMIO from Enclaves. When mapping an I/O device directly into the enclave region, address translation will fail due to SGX memory access checks. SGX requires that, if the virtual address is within the enclave region, the translated physical address must be part of the EPC [58]. Since MMIO cannot be part of the EPC, MMIO devices have to be mapped into the non-enclave virtual address range. This is no problem because enclaves have direct read/write access to non-enclave memory, as shown in Figure 32.

MMIO with seL4. In seL4 an MMIO device is represented as a chunk of untyped memory, covering the device's physical address range. The device becomes accessible by retyping the device's untyped memory to pages and mapping these pages into a process' virtual address space. Once MMIO is set up, access to MMIO devices does not involve any interaction with the kernel. I/O is directly carried out by accessing the virtual address of the mapped device. Hence, MMIO has speed advantages over I/O ports because in seL4, access to an I/O port is only possible via syscalls.

For secure MMIO, we again require a transparent MMIO binding between a driver and an MMIO device. It is not enough that the device is properly and exclusively mapped into the driver's virtual address space. In order to verify the binding with the `HasExclusive` syscall, the driver needs to have access to the device's page capabilities. Furthermore, it needs to know at which exact virtual address the device is mapped. Thus, the easiest way to achieve transparency is giving the driver exclusive ownership of the whole device's untyped memory. Then, the driver can directly verify exclusivity of the untyped memory. The driver itself retypes memory and maps it appropriately.

Non-revocability of the MMIO binding is achieved the same way as for I/O ports. However, identification of the MMIO device is not directly possible in seL4. A device is identified by its physical address. In seL4 it is not possible to determine the physical address of an untyped memory or retyped page capability. seL4 keeps the mapping between cap and physical address hidden from user processes. Hence, one would require a new seL4 syscall which can query the physical address of certain memory caps.

8.2 Reducing Complexity.

We see two possible improvements for making SGX more suited to secure port I/O by reducing total complexity. First, allowing syscalls from enclave code or second, providing secure port I/O in hardware. Note that latter improvement makes the first one obsolete. Both options require changes to SGX hardware. We motivate them in the following.

Syscall Support. SGX lacks support for issuing syscalls from an enclave. However, this is necessary for doing secure port I/O in software. A syscall is needed to establish a secure path between I/O driver and seL4 kernel. Without this syscall, one needs to build a workaround by exploiting paging side channels.

This is neither performant nor good practice. Furthermore, since paging side channels exist in both directions between enclaves and kernel, it would be of no harm for SGX to also provide a direct channel. Hence, direct syscall support for SGX enclaves is desirable.

SGX could introduce a new ENCLU user-mode instruction, called `ESYSCALL`. When the CPU traps into the kernel, the enclave register file is stored in the enclave's SSA. Then all registers are masked. `ESYSCALL` could preserve certain registers to allow passing information to the kernel. When returning from the kernel with `ERESUME`, SGX restores the enclave register file from the SSA. Again, certain registers could be spared out. Thus, the kernel could return data to the enclave. Note that `ERESUME` is currently an ENCLU instruction, which cannot be executed from within the kernel (ring-0). `ERESUME` would need to be changed in order to allow invocation from both, ring-0 and ring-3.

Hardware Support for Secure Port I/O. The use of a security kernel like seL4 gives a cheap and flexible method of doing secure port I/O with SGX. However, integration of secure port I/O in an existing software stack might be difficult due to additional complexity. Doing the I/O binding in hardware would help reduce this complexity.

Without additional hardware support, the insecure rich OS has to integrate the seL4 kernel in its resource management. This might be quite intrusive since the rich OS has to be aware of seL4 capabilities. This is, however, required in order to dynamically load the secure I/O driver.

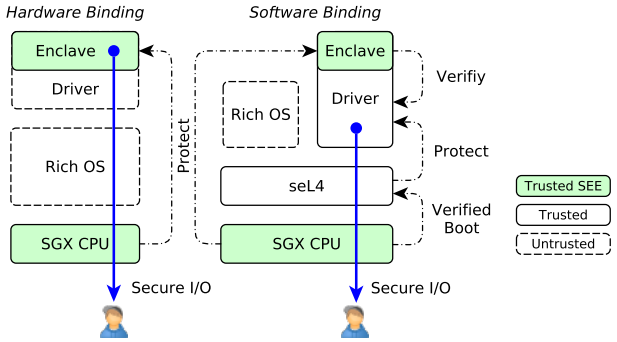

Figure 33: Hardware binding comes with little overhead, allowing secure port I/O directly from within an enclave. Software binding in addition requires verified launch of seL4 and the non-enclave driver.

Also, doing secure bootstrapping of our architecture is quite complex. Verified launch of the seL4 kernel is done using secure boot features of Intel TXT and the TPM. Furthermore, the driver's non-enclave is protected by seL4's strong process isolation. Last, verification of the driver's non-enclave code is performed by the driver enclave, as shown in Figure 33.

If an enclave could directly do secure port I/O without relying on some security kernel, this would reduce the number of verification steps necessary to set up secure port I/O. In special, driver's non-enclave code would neither require verified launch nor secure execution. Likewise, seL4 and the trusted boot would be avoided.

8.3 I/O Binding in SGX Hardware

In this section we show how a secure I/O binding could be directly enforced by an SGX-enabled CPU. We distinguish between hardware-assisted secure port I/O and hardware-assisted secure MMIO. Finally, we discuss how the I/O binding can be initialized and destroyed during driver (un)loading.

Our proposed modifications only affect the SGX CPU. In contrast, ARM TrustZone propagates the secure world status to other peripherals as well via a separate wire. Hence, a peripheral can decide on its own whether to grant access or not. Note that this is impractical for SGX since different enclaves need to be distinguished. In SGX, the enclave process is identified by its SECS page. The physical address of the SECS page has 8 bytes. Using this SECS identifier, one would require at least propagation of a 64 bit identifier to other peripherals.

8.3.1 Hardware-assisted Secure Port I/O. Similar to the EPCM, SGX reserves an I/O Port Map (IOPM) in processor-reserved memory. Each port address maps to a slot in the IOPM. When an I/O port is accessed, the CPU looks up the corresponding slot in the IOPM. If empty, access is granted. This allows unprotected port I/O for legacy applications. If non-empty, the current process must be in enclave mode and its SECS identifier must match those in the IOPM slot. The access check is depicted in Figure 34.

I/O ports range from `0x0000 ... 0xFFFF`. One IOPM slot needs to hold at least an 8-byte SECS identifier. If each slot addresses 8 I/O bytes, the size of the IOPM is 64KB. This is reasonable since RAM is cheap. Even if most ports are unused, one does not loose much memory. Moreover, a granularity of 8 bytes is sufficient for most purposes. The serial COM ports, for example, are addressed by 8 bytes each.

8.3.2 Hardware-assisted Secure MMIO. To integrate secure MMIO in SGX hardware, we outline two options: Either one could extend the EPCM to handle MMIO pages as well, or one could add an I/O protection unit.

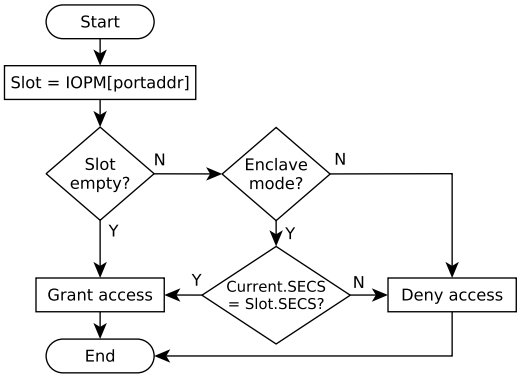

Figure 34: Secure port I/O in SGX hardware.

Extending the EPCM. The EPCM could not only be used to keep track of enclave pages but also of MMIO pages. To achieve this, a contiguous MMIO region is defined as protected. Each page in this protected MMIO region is assigned a slot in the EPCM. During address translation, the CPU checks if a protected MMIO page is accessed. In that case, SGX looks up the corresponding slot in the EPCM and does permission checks, similar to ordinary EPC pages checks.

An EPCM slot mainly contains the virtual address, under which the page is mapped, and the SECS identifier. Furthermore, the slot covers some protection and status flags. Thus, a slot has at least a size of 17 bytes, as shown in Figure 35, although the exact layout is left as an unspecified implementation detail in SGX.

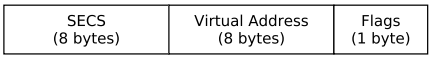

Figure 35: The EPCM slot contains the SECS identifier, the virtual page address and some flags.

For deciding if a page is within the EPC or not, SGX implements a very fast and cheap region check in hardware, as detailed in [28]. This check requires that the size of the EPC region needs to be a power of two. Furthermore, its base address needs to be naturally aligned. Consider L as the region's lower bound and U as the inclusive, upper bound. An address A is within the region if $A \wedge \neg(L \oplus U) = L$. This check only requires a bitwise XOR, AND, negation as well as comparison, which are both cheap and fast operations in hardware.

We reuse this region check for the MMIO region. Hence, the MMIO region check comes with minimal changes to hardware. Also, the permission checks for normal EPC pages might be directly reused for checking permissions of MMIO pages. The biggest hardware cost is the increased memory demand of the additional MMIO slots. The memory demand strongly depends on the size of the MMIO region.

To get an understanding of a typical MMIO layout, we inspect a modern Dell Latitude. Its MMIO region covers the address range `0xD0000000 ... 0xF7FFFFFF`, as seen in Figure 36. To enumerate each MMIO page separately, one would require about 164.000 additional slots in the EPCM. With a slot size of at least 17 bytes, the EPCM increases by at least 2.7MB in this example.

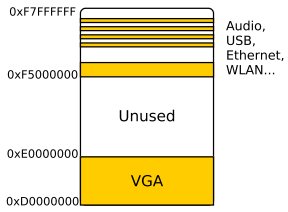

Figure 36: Typical MMIO layout with many holes of unused addresses. Note that sizes are not true to scale.

Two key observations can be made: First, different devices might have significantly different sizes of their address region. This cannot be properly represented on page granularity. For example, the VGA graphics controller claims 256MB of the memory addresses, which are 2^{16} pages. In contrast, Intel's Management Engine only claims 32 bytes of memory, which is less than one-hundredth of a single page. Second, not all addresses of the MMIO region are actually used. The inspected notebook has unused address holes of over 330MB. To be more memory efficient, one should distinguish devices rather than pages. This can be achieved by an I/O protection unit.

I/O Protection Unit. The I/O Protection Unit (IOPU) solves the issue of a sparsely-used and inhomogeneous MMIO region. It keeps track of each protected device in a special cache. The IOPU caches all MMIO bindings, that is, it maps protected physical device regions to enclaves. The IOPU cache is fully associative. Hence, each IOPU slot can represent one arbitrary device region. To lookup a specific address, the IOPU performs address region checks on all slots in parallel. This can be done by the same fast and cheap region checking as before, given that a device region is a power of two.

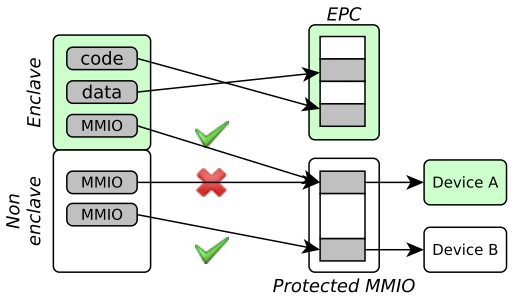

Figure 37: Device A is bound to the enclave. Hence, it can only be accessed from the enclave region. Device B can be mapped to non-enclave code as well since it is not claimed by any enclave.

SGX and the IOPU basically have to implement the following rules: If an enclave page resolves to the EPC, ordinary SGX checks apply. If it points to the protected MMIO region, the IOPU has to find a corresponding slot in its cache. Moreover, the slot has to match the virtual address and the SECS identifier of the calling enclave. If it resolves to unprotected memory, access fails. For non-enclave code there is one difference, compared to standard SGX: If a non-enclave page maps to a protected MMIO device, the device region must not be claimed by any enclave, that is, the IOPU must not find a non-empty slot for the device region. Figure 37 illustrates this restriction. In contrast, the non-enclave can map unclaimed MMIO devices. This allows compatibility with existing non-enclave code.

The big advantage of an IOPU is that each device region is addressed by only one IOPU slot, regardless of its size. Typical notebooks have around 20 distinct device regions. Moreover, the IOPU only caches those device regions which actually need protection. Hence, 64 IOPU slots should be fairly enough.

8.3.3 Driver (Un)loading. To set up an I/O binding, one has to initialize the corresponding slot, either in the IOPM for secure port I/O or in the EPCM or the IOPU for secure MMIO. This slot initialization can be safely delegated to the untrusted OS by adding a new ENCLS instruction, called `EADDIO`, for example. A device driver typically handles a single device during its whole lifetime. Hence, it is legitimate to do this binding statically when loading the driver. During enclave setup, `EADDIO` establishes the binding by initializing the corresponding slot. This only works out if the slot is empty, that is, no other enclave has currently access. Furthermore, `EADDIO` includes the physical address range of the I/O port or the MMIO device in the measurement. Thus, the I/O resource gets identified. If

enclave initialization (`EINIT`) succeeds, the driver can be sure that it has been assigned the correct device or port and that it has exclusivity over it.

In the case of an MMIO binding, additional care must be taken. To detect if MMIO pages are correctly mapped, the expected virtual address needs to be measured by `EADDIO`. Since potentially many MMIO pages are represented by the same IOPU slot, there is certain ambiguity. MMIO pages might be mapped into the enclave in a scrambled order. Hence, we require the IOPU to enforce that the contiguous physical address region is also mapped to a contiguous virtual address region.

To guarantee non-revocability of the I/O resource, deallocation of the corresponding slot requires cooperation by the owning enclave. Note that the slot cannot be automatically deallocated by SGX if the enclave is destroyed. Otherwise the untrusted OS could simply kill the driver enclave while doing secure I/O, grab the resource and impersonate the driver enclave. To avoid this, slot deallocation could be bound to a special ENCLU instruction, called `EREMOVEIO`, for example. When the driver unloads, it issues this instruction and the slot is freed. If the untrusted OS kills the enclave beforehand, the slot and the corresponding resource remain inaccessible until the next reboot.

Chapter 9

Conclusion

In this thesis we investigated secure port I/O with Intel SGX. We argued that a secure execution technology alone is rather worthless for an end user, who wants to protect secret data on a mobile computing device. Even if data is protected by an SEE, the user simply cannot access this data in a secure way without secure I/O. Therefore, for mobile devices the combination of secure execution with secure port I/O is crucial in enabling user-centric security applications like secure password entry, electronic payment, secure chat and similar.

Intel Software Guard Extensions is a comprehensive modern secure execution technology for x86 CPUs. Unfortunately, it lacks support for secure port I/O. Our main contribution is to enable secure port I/O with Intel SGX. We developed a secure I/O architecture in which a user application communicates with a secure I/O driver to securely interact with a user. Our architecture permits to run an insecure rich OS stack on the mobile device. We leveraged SGX enclaves to protect the user application and the driver from the insecure rich OS. The user application uses SGX local attestation to securely communicate with the I/O driver. We added secure port I/O features to SGX using the seL4 security kernel, which runs below the rich OS. We used seL4 to bind an I/O resource to the secure I/O driver. This binding ensures the driver exclusivity and full control over the I/O resource. Furthermore, we modified seL4 to allow the driver to transparently verify if the I/O binding is properly established. Hence, our architecture effectively enables to do secure port I/O from within SGX-protected user applications.

In our architecture, the security kernel seL4 is part of the TCB. This is acceptable since seL4 is small and verifiable. From a security point of view one can compare our architecture with ARM TrustZone in that sense that TrustZone also employs a security kernel for setting up a secure world stack. In order to reduce complexity, we discussed possible modifications to SGX hardware. Therefore, we outlined methods for integrating secure port I/O directly in SGX. This basically shifts management of the secure I/O binding to hardware.

Intel already has experience in hardware-assisted secure I/O features like PAVP, Intel Insider and PTD. However, these are on the one hand not generic enough to support arbitrary legacy I/O devices and on the other hand not available to the broad public. Hence, we see it as a necessary step for Intel to add generic and transparent secure I/O features to its x86 CPUs, which are not subject to licensing schemes of any kind. This would allow Intel to catch up with ARM TrustZone, which already supports generic secure I/O in hardware for several years now.

References

1. Internet Census 2012. Carna botnet: Port scanning /0 using insecure embedded devices, 2012. `http://internetcensus2012.bitbucket.org/paper.html` (2016/04/04).
2. I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. *Innovative Technology for CPU Based Attestation and Sealing*. Intel Corporation, Aug 2013.
3. iOS Security, iOS 9.0 or later. White paper, Apple Inc., Sep 2015. `https://www.apple.com/business/docs/iOS_Security_Guide.pdf` (2016/04/04).
4. ARM. TrustZone. `http://www.arm.com/products/processors/technologies/trustzone/index.php` (2016/04/04).
5. ARM. *PrimeCell Infrastructure AMBA 3 AXI TrustZone Memory Adapter (BP141)*, Dec 2004. Reference no. DTO 0017A.
6. ARM. *PrimeCell Infrastructure AMBA 3 TrustZone Protection Controller (BP147)*, Nov 2004. Reference no. DTO 0015A.
7. ARM. *ARM CoreLink TM TZC-400 TrustZone Address Space Controller*, Sep 2015. Reference no. 100325_0001_02_en.
8. N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, S. Käsper, E. Cohney, S. Engels, C. Paar, and Y. Shavitt. DROWN: Breaking TLS using SSLv2, May 2016. `https://drownattack.com/#paper` (2016/04/04).
9. E. Barker and A. Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. Nist special publication 800-131a, rev. 1, National Institute of Standards and Technology, Nov 2015.
10. A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 267–283, Berkeley, CA, USA, 2014. USENIX Association.
11. BlackBerry Enterprise Service 10. Security Technical Overview. White paper, BlackBerry Ltd., Sep 2014. Reference no. SWD-20140908123239883.
12. BlueKrypt. Cryptographic key length recommendation. `https://www.keylength.com` (2016/04/04).
13. R. Boivie and P. Williams. SecureBlue++: CPU support for secure executables. Research report, IBM Research Division, Apr 2013. Reference no. RC25369.
14. S. L. Braunstein and S. Pirandola. Side-channel-free quantum key distribution. *Phys. Rev. Lett.*, 108:130502, Mar 2012.
15. E. Brickell and J. Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. In *Social Computing (SocialCom), 2010 IEEE Second International Conference on*, pages 768–775, 2010.
16. E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 27–38. ACM, 2008.
17. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. `https://competitions.cr.yp.to/caesar.html` (2016/04/04).
18. J. Camhi. BI Intelligence projects 34 billion devices will be connected by 2020. *Business Insider*, Nov 2015. `http://www.businessinsider.com/bi-intelligence-34-billion-connected-devices-2020-2015-11` (2016/04/04).

19. P. Carbin. Intel Identity Protection Technology with PKI. White paper, Intel Corporation, May 2012.
20. S. Checkoway and H. Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. *SIGARCH Comput. Archit. News*, 41(1):253–264, March 2013.
21. X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 2–13. ACM, 2008.
22. S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: A hardware-software approach to full system security. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 108–119, New York, NY, USA, 2011. ACM.
23. S. G. Choi, J. Katz, R. Kumaresan, and C. Cid. Multi-client non-interactive verifiable computation. In *Theory of Cryptography*, pages 499–518. Springer, 2013.
24. Codenomicon. The heartbleed bug, Apr 2014. `http://heartbleed.com/` (2016/04/04).
25. B. Cooper. Method and apparatus for trusted keyboard scanning, Dec 2006. US Patent 7,145,481.
26. B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy*, pages 45–60, May 2009.
27. Intel Corporation. Intel Software Guard Extensions (Intel SGX). Tutorial slides presented at ISCA 2015, Portland, OR, USA, 2015. Reference no. 332680-002. `https://software.intel.com/sites/default/files/332680-002.pdf` (2016/04/04).
28. V. Costan and S. Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, Feb 2016. `http://eprint.iacr.org/2016/086.pdf` (2016/04/04).
29. S. Crosby, I. Goldberg, R. Johnson, D. Song, and D. Wagner. A cryptanalysis of the high-bandwidth digital content protection system. In *Security and Privacy in Digital Rights Management*, pages 192–200. Springer Berlin Heidelberg, 2002.
30. W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov 1976.
31. J. Drake. Stagefright: Scary code in the heart of android researching android multimedia framework security. Presentation at Black Hat, USA, Aug 2015. `https://www.blackhat.com/us-15/briefings.html#stagefright-scary-code-in-the-heart-of-android` (2016/04/04).
32. GlobalPlatform. *TEE System Architecture*, Dec 2011. Reference no. GPD_SPE_009.
33. Leveraging GlobalPlatform to improve security and privacy in the internet-of-things. White paper, GlobalPlatform, May 2014.
34. M. Green. Reposted: A cryptanalysis of HDCP v2.1. A Few Thoughts on Cryptographic Engineering, Aug 2012. `http://blog.cryptographyengineering.com/2012/08/reposted-cryptanalysis-of-hdcp-v2.html` (2016/04/04).
35. S. Halevi. Practical (F)HE. Presentation at Summer School on Mathematical and Practical Aspects of Fully Homomorphic Encryption and Multi-Linear Maps, Paris, Oct 2015. `https://heat-project.eu/summerschool2015.html` (2016/04/04).
36. HDfury. HDfury launches yet another path breaking device – the HDCP Doctor, HDfury Integral, Nov 2015. `https://www.hdfury.com/11159/` (2016/04/04).

37. D. Hein, J. Winter, and A. Fitzek. Secure block device – secure, flexible, and efficient data storage for ARM TrustZone systems. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 222–229, Aug 2015.
38. M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. 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*, HASP '13, New York, NY, USA, 2013. ACM.
39. Gartner Inc. Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015, Nov 2015. `https://www.gartner.com/newsroom/id/3165317` (2016/04/04).
40. Intel Corporation. *Intel Software Guard Extensions Programming Reference*, Oct 2014. Reference no. 329298-002US.
41. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Sep 2015. Reference no. 325462-056US.
42. Intel Corporation. *Intel Pro Wireless Display Implementation Guide*, Nov 2015. Version 2.4. `https://downloadcenter.intel.com/download/25547/Intel-Pro-WiDi-Documentation` (2016/04/04).
43. Intel Corporation. *Intel Trusted Execution Technology (Intel TXT), Software Development Guide*, Jul 2015. Reference no. 315168-012.
44. Intel Corporation. *Intel Software Guard Extensions Evaluation SDK for Windows OS. User's Guide*, Jan 2016. Revision 1.1.1. `https://software.intel.com/sgx-sdk/documentation` (2016/04/04).
45. P. Jain, S. Desai, S. Kim, MW. Shih, JH Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, Feb 2016.
46. S. Johnson, D. Zimmerman, and B. Derek. Intel SGX: Debug, Production, Pre-release what's the difference?, Jan 2016. `https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference` (2016/04/04).
47. A. Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX:5–38, Jan 1883.
48. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220. ACM, 2009.
49. S. Knight. Intel to enable SGX technology on future skylake CPUs. *Techspot*, Oct 2015. `http://www.techspot.com/news/62324-intel-enable-sgx-technology-future-skylake-cpus.html` (2016/04/04).
50. N. Knupffer. Intel Insider – what is it? (is it DRM? and yes it delivers top quality movies to your pc). *Technology@Intel*, Jan 2011. `https://blogs.intel.com/technology/2011/01/intel_insider_-_what_is_it_no/` (2016/04/04).
51. Open Kernel Labs. OK Labs software surpasses milestone of 1.5 billion mobile device shipments, Jan 2012. `http://web.archive.org/web/20120211210405/http://www.ok-labs.com/releases/release/ok-labs-software-surpasses-milestone-of-1.5-billion-mobile-device-shipments` (2016/04/04).
52. S. Larsen, B. Lee, JH. Yoon, and JY. Yun. Direct device-to-device transfer protocol: A new look at the benefits of a decentralized I/O model. In *2015 IEEE*

International Conference on Networking, Architecture and Storage (NAS), pages 275–284. IEEE, 2015.
53. Digital Content Protection LLC. HDCP Specifications. `http://www.digital-cp.com/hdcp-specifications` (2016/04/04).
54. Juniper Research Ltd. Global smartphone shipments exceeded 1.2 billion in 2014, Jan 2015. `http://www.juniperresearch.com/press/press-releases/global-smartphone-shipments-exceeded-1-2-billion-i` (2016/04/04).
55. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
56. Linux Programmer's Manual. elf - format of executable and linking format (ELF) files. `http://man7.org/linux/man-pages/man5/elf.5.html` (2016/04/04).
57. D. McGrew and J. Viega. The Galois/Counter Mode of operation (GCM). *Submission to NIST Modes of Operation Process*, Jan 2004.
58. F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. 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*, HASP '13, page 10:1, New York, NY, USA, 2013. ACM.
59. K. T. Nguyen, M. Laurent, and N. Oualha. Survey on secure communication protocols for the internet of things. *Ad Hoc Networks – Internet of Things security and privacy: design methods and optimization*, 32:17–31, Sep 2015.
60. NICTA. seL4 security microkernel. `http://sel4.systems` (2016/04/04).
61. NICTA. *seL4 Reference Manual, API version 1.2*, Mar 2013.
62. NICTA. *seL4 Reference Manual, Version 3.0.0*, Mar 2016. `https://wiki.sel4.systems/Documentation` (2016/04/04).
63. OMTP Limited. *Advanced Trusted Environment: OMTP TR1*, May 2009. Version 1.1.
64. E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 13–24, New York, NY, USA, 2013. ACM.
65. PCI Security Standards Council. Approved pin transaction security devices. `https://www.pcisecuritystandards.org/assessors_and_solutions/pin_transaction_devices` (2016/04/04).
66. Qemu open source processor emulator. `http://wiki.qemu.org` (2016/04/04).
67. RefOS source code. `https://github.com/seL4/refos-manifest` (2016/04/04).
68. X. Ruan. *Platform Embedded Security Technology Revealed. Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. ApressOpen, 2014.
69. R. Shaw. System and method for communication of keyboard and touchpad inputs as HID packets embedded on a SMBus, May 2005. US Patent App. 10/723,896.
70. S. E. Siwek. The true cost of motion picture piracy to the U.S. economy. Policy Report 186, Institute for Policy Innovation, Sep 2006.
71. J.H. Song, R. Poovendran, J. Lee, and T. Iwata. The AES-CMAC Algorithm, Jun 2006. RFC 4493.
72. G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.
73. TCG. *Trusted Platform Module Library. Part 1: Architecture. Family 2.0*, Oct 2014. Revision 01.16.

74. SSlab Georgia Tech. OpenSGX source code. `https://github.com/sslab-gatech/opensgx` (2016/04/04).
75. Trusted Computing Group. `http://www.trustedcomputinggroup.org` (2016/04/04).
76. A. Vasudevan, J. McCune, J. Newsome, A. Perrig, and L. Van Doorn. CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 48–49, New York, NY, USA, 2012. ACM.
77. S. H. Weingart. Physical security devices for computer subsystems: A survey of attacks and defenses. In *Cryptographic Hardware and Embedded Systems*, pages 302–317. Springer Berlin Heidelberg, Aug 2000.
78. Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, Oakland, May 2015. IEEE.
79. J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 71–80. ACM, 2008.