Hardware Platform Security for Mobile Devices

Lachlan J. Gunn N. Asokan Jan-Erik Ekberg
Hans Liljestrand Vijayanand Nayani Thomas Nyman

April 8, 2022

Abstract

Today, personal mobile devices like smartphones and tablets are ubiquitous. People use mobile devices for fun, for work, and for organizing and managing their lives, including their finances. This became possible because over the last two decades, mobile phones evolved from closed platforms intended for voice calls and messaging to open platforms whose functionality can be extended in myriad ways by third party developers. Such wide-ranging scope of use also means widely different security and privacy requirements for those uses. The mobile device ecosystem involved multiple different stakeholders such as mobile network operators, regulators, enterprise information technology administrators, and of course ordinary users. So, as mobile platforms became gradually open, *platform security* mechanisms were incorporated into their architectures so that the security and privacy requirements of all stakeholders could be met. Platform security mechanisms help to isolate applications from one another, protect persistent data and other on-device resources (like access to location or peripherals), and help strengthen software against a variety of attack vectors. All major mobile platforms incorporate comprehensive software and hardware platform security architectures, including mechanisms like *trusted execution environments (TEEs)*.

Over the past decade, mobile devices have been undergoing convergences in multiple dimensions. The distinction between "mobile" and "fixed" devices has blurred. Similar security mechanisms and concepts are being used across different platforms, leading to similar security architectures. Hardware enablers used to support platform security have gradually matured. At the same time, there have also been novel types of attacks, ranging from software attacks like return- and data-oriented programming to hardware attacks like side channels that exploit micro-architectural phenomena. It is no longer tenable to assume that the current hardware security mechanisms underpinning mobile platform security are inviolable.

The time is therefore right to take a new look at mobile platform security, which brings us to this book. We focus on *hardware platform security*. The book is divided into four parts: we begin by looking at the *why* and *how* of mobile platform security, followed by a discussion on *vulnerabilities and attacks*; we conclude by *looking forward* discussing emerging research that explores ways of dealing with hardware compromise, and building blocks for the next generation of hardware platform security.

Our intent is to provide a broad overview of the current state of practice and a glimpse of possible research directions that can be of use to practitioners, decision makers, and researchers.

Part I

Mobile Platform Security: Why?

Chapter 1

Introduction

Today, mobile devices such as smartphones and tablets are very widely deployed. All modern mobile device platforms incorporate sophisticated software and hardware *platform security* mechanisms. To understand how this came to be, we need to start in the late 1990s.

1.1 What motivated mobile platform security?

The mobile phone revolution was well under way by the mid 1990s. Initially, mobile phones were simple embedded devices with fixed functionality: voice calls and text messages. Early on, the mobile phone industry recognized the power of billions of people having general-purpose computing devices in their hands. Personal digital assistants were already available and demonstrated the range of uses for portable, personal general-purpose computing devices. Therefore, already by the mid 1990s, the industry was working towards *opening up* mobile phone platforms so that users gain the ability to *extend* their functionality, by installing third-party software modules. This development directly led to today's smartphones and tablets with their "app" ecosystems.

The industry saw the potential for new types of applications like mobile payments, public transport ticketing, and digital media consumption. But it also realized that for these applications to succeed, open mobile devices needed additional mechanisms to safeguard the security and privacy requirements of these novel, and potentially high-value, applications.

Furthermore, the mobile phone ecosystem already had well-established stakeholders. They were sensitive to the security and privacy concern that could arise in the transition from closed fixed-function devices to open platforms. Existing commodity general-purpose computing platforms at the time, like those for personal computers, did not incorporate the platform security mechanisms necessary to address these concerns. Consequently, they wanted to mediate this transition so that their own interests were safeguarded. This, too, drove the development of new platform security mechanisms. Mobile platform se-

curity architectures emerged because of the need to address these stakeholder concerns as mobile devices opened up [178].

1.2 Stakeholders

An important class of stakeholders are mobile network operators (MNOs) (also known as "carriers") who are motivated by business interests. An example of a business interest of MNOs is the need to strongly authenticate their subscribers. This need led to the introduction of subscriber identity modules (SIMs) (discussed further in Section 2.2) right from the beginning. Another example of a business interest of MNOs is the need for robust technical mechanisms to support the *subsidy-lock business model* where a MNO gives a mobile phone to a subscriber for free or below cost, in return for a commitment to maintain the subscription for a specified period of time. The requirement to technically enforce subsidy locks translated into each device having an unforgeable unique identifier and the ability to run subsidy-lock enforcement software in a manner that cannot be bypassed.

Another, equally important, class of stakeholders are *regulators* who safeguard the public good. An example of a regulatory need is to ensure that radio-frequency transmission parameters, which are typically calibrated for each device at the time of manufacture, cannot be tampered with. This need can be met with secure (integrity-protected) storage for storing these parameters.

A third class of stakeholders are end-users. They were used to mobile phones that were reliable and trustworthy. They expected the same degree of reliability and trustworthiness to be maintained, even as mobile phone platforms were opening up.

There are other stakeholders in the ecosystem, like enterprise administrators, and of course the mobile phone manufacturers —also known as original equipment manufacturers (OEMs)—and operating system (OS) vendors themselves. To see what kinds of mechanisms are needed to protect the interest of different stakeholders, it is necessary to understand the threat models from the perspectives of these stakeholders.

1.3 Threat models

A threat model involves characterizing the adversary in terms of its capabilities, and the assets that need to be protected from these adversaries. For example a *software adversary* is assumed to be capable of influencing one or more software modules on the victim device. The adversary's control may be limited to a single application (software in user space) or can extend to privileged software like the OS itself. In contrast, a *hardware adversary* can directly interact with, and possibly manipulate, the hardware components on the victim device.

Rather than presenting an exhaustive treatment of all possible threat models, we will illustrate the concept with three informal examples.

First, consider the threat of a user's address book being exfiltrated from the device by a malicious third-party application that the user happened to install. We are concerned with a software adversary (the third-party developer) and the asset that needs protection is the address book. Standard hardware support (for memory management and process isolation) combined with a good OS security architecture (providing access-controlled persistent storage for each application) would be sufficient to provide the required protection. In Chapter 3 we will discuss OS security architectures.

Next, consider the same setting as above, but with a different asset: credentials for accessing financial transactions like online banking. While we are still concerned with a software adversary, the value of the asset is significantly higher, and its compromise can result in substantial losses. Consequently, relying only on OS security is not reasonable because an OS is a complex software component with a large threat surface for the attacker to exploit. Additional hardware support for protecting high-value assets is justifiable. Hardware-assisted trusted execution environments (TEEs) allow small pieces of trusted software on a general-purpose computing device to be *isolated* from the rest of the software on the same device, including the OS and other applications. Today TEEs are ubiquitous. Nearly every smartphone or tablet is likely to have a processor with TEE capabilities. Many personal computers are also equipped with TEEs. The ubiquity of TEEs is not a recent phenomenon [93]: hardware-assisted TEEs started to appear in mobile phones from the early 2000s. For a technology that is so widely deployed, for so long, the origins and trajectory of TEE technologies are poorly understood. Our primary focus in this book is to explore hardware platform security for mobile devices, with a particular emphasis on TEEs.

Finally, consider the case of technical mechanisms for subsidy-lock enforcement. The adversary in this case is the user of the device who has physical access to the device. The asset that the adversary wants to compromise is the binding between the mobile device hardware and the MNO (so that a successful attack will result in breaking the binding, allowing the adversary to use the device with a different MNO subscription). OS security alone is not sufficient. Since we now deal with a potential hardware adversary, we must use hardware-security mechanisms that can withstand physical attack.

1.4 Chains of trust

In a given scenario, the party relying on the protection mechanism trusts the software and hardware components used to realize the mechanism. A *chain of trust* refers to the process of building up this trust, starting from one or more *roots of trust*. In the first example above, while OS security is sufficient, the relying party, the user, needs to trust that the correct OS is running on the device. *Platform integrity* (Chapter 4) makes it possible to build up this trust. Higher level platform security mechanisms like OS security rely on underlying building blocks like platform integrity (Chapter 4), hardware-assisted isolation

(Chapter 5), and cryptographic primitives realized in hardware (Chapter 6).

An important feature of hardware platform security mechanisms is allowing remote relying parties to build up trust in a device. In the second example above, a bank may need to convince itself that the user is accessing her bank account from a secure device before allowing access. This feature is called *remote attestation*, which is widely supported by modern TEEs. In Chapter 5 we will discuss the chains of trust involved in remote attestation in modern TEEs.

We begin with an overview of the history of mobile hardware platform security mechanisms (Chapter 2), and provide an overview of OS security (Chapter 3) to understand how an OS can make use of these mechanisms. We will explore the nuts and bolts of how platform security is implemented in today's devices, focusing on hardware platform security (Part II), and discuss attacks against hardware platform security mechanisms (Part III). We will conclude with a brief foray into future outlook for hardware platform security (Part IV).

Notes on the scope of this book: The focus of this book is on hardware platform security in mobile devices. We do cover OS security in Chapter 3, but from the perspective of motivating hardware platform security. Mobile device platforms also incorporate sophisticated *software platform security* mechanisms. We refer readers interested in this topic to books dedicated to the topic such as [35]. We also do not cover specific high-level attacks such as *jail-breaking* (removing manufacturer-imposed restrictions on what software can be installed on a mobile device) or *rooting* (obtaining the privileges of the maximally privileged "root" user on Unix-based mobile OSs). However, the basic attacks we describe in Part III can, and often are, used as stepping stones for these high-level attacks.

Chapter 2

Historical Overview

The requirements we saw in Chapter 1 led to mobile device and platform vendors developing and deploying software and hardware platform security architectures. Nokia Radio Application Processors are believed to be the first trusted execution environments (TEEs) deployed at a large scale [178]. These were followed shortly by Texas Instruments' M-Shield™ [248] and subsequently by ARM's TrustZone™[6] which represents the overwhelming share of deployed mobile TEEs today.

In the non-mobile setting, hardware security modules (HSMs) used in the financial sector (starting with IBM's CryptoCard[1]) are an early example of a TEE. Trusted Computing Group's Trusted Platform Modules (TPMs) [34] are widely deployed in personal computers, where they are used for boot integrity and disk encryption, but they have not found common use in the mobile space. Recently, Intel's Software Guard Extensions (SGX) [182] has become the most widely studied TEE architecture, thanks to the easy availability of both the software and hardware[2]. SGX is primarily deployed in cloud settings to enable confidential computing use cases [5, 183]. Desktop use cases for SGX include Blu-ray digital rights management (DRM) [253]

2.1 Hardware security modules

Early examples of the inclusion of a dedicated security co-processor were motivated by the need to perform sensitive cryptographic operations isolated from other computations in systems handling financial transactions. Transaction processing for Europay, Mastercard and Visa (EMV) payment cards use HSMs as the primary security device for key management [85]. An HSM is a discrete computing device usually encapsulated in tamper-evident coating. HSMs in backend systems typically include specialized cryptographic hardware accelerators to enable high throughput because they need to process transactions in

[1] `https://www.ibm.com/security/cryptocards/`

[2] `https://software.intel.com/en-us/sgx`

real-time. An HSM can be realized as either as a stand-alone peripheral device or as an extension board connected directly to the internal bus of the host computer. The operational keys are generated in the cryptographic co-processor within the HSM and are then saved either in a keystore file or in application memory, encrypted under the master key of that co-processor. Any HSM with an identical master key can use those keys.

The first commercially available civilian HSMs were deployed already in the 1970s, originally for IBM mainframes. The IBM 3845 and 3846 data encryption devices [140] allowed exported encryption keys to be encrypted using the recently standardized DES algorithm. These early HSMs included secure key entry devices (cards and PIN pads) for master key loading, random number generation capabilities for seeding, and persistent storage for key materials. They were instrumental in securing early electronic banking, such as automatic teller machines (ATMs).

2.1.1 HSMs in radio communication

HSMs are also extensively deployed for modern military software-defined radio (SDR) communication. SDR refers to wireless communications where the transmitter and receiver mixing, filtering, amplification, modulation/demodulation etc. occur in software instead of in conventional radio electronics. With SDR, software-based transmission algorithms can be downloaded and adapted over the lifecycle of the hardware. While analog military radio equipment include dedicated cryptographic chips for (proprietary) ciphers that are required for communication with compatible equipment, SDR equipment have to support a large number of cryptographic schemes, including legacy protocols and algorithms. Consequently military SDR equipment, such as the U.S. Joint Tactical Radio System (JTRS), employ embeddable HSMs specifically designed for communication security. The Advanced INFOSEC Machine (AIM) [106] is one such programmable, embeddable cryptographic unit developed by Motorola in the late 1990s. It consists of a hardware platform with three independent cryptographic processors, one for key management and two programmable processors for traffic encryption/decryption. The key management cryptographic engine (KMCE) is based on a 32-bit reduced instruction set computer (RISC) processor and includes a math co-processor designed for public key algorithm processing. The KMCE runs a read-only memory (ROM)-based Secure Operating System (SOS). The SOS provides a multi-security level, multitasking environment for the cryptographic applications which allowed the functionality of the AIM to be extended by software. The chip contains the necessary building blocks to implement encryption algorithms such as DES, and the classified SAVILLE and BATON cryptographic algorithms used by U.S. and NATO. Its successor, AIM II [107] is specifically designed for JTRS. Around the same time, a similar crypto-chip, called the General Crypto Device (GCD) [161], was developed in Europe by Dutch electronics giant Philips.

The use of HSMs such as AIM and GCD are early examples of the use of TEEs in telecommunications. The sensitivity of military communication jus-

tified the inclusion of dedicated components for security into end devices. However, for civilian telecommunication devices, the widespread use of TEE-technology only occurred when two conditions were met: 1) economic incentives emerged to justify requiring strong, hardware-based security, and 2) low-cost technological solutions that met those requirement were developed.

2.2 SIMs, mobile handsets, and smart cards

During the early 1990s, civilian wireless communication systems also began to employ hardware-assisted security. Mobile network operators (MNOs) required a reliable way of preventing illicit use of a subscriber identity for making phone calls from mobile phones. For this purposes, the subscriber identity module (SIM) card [132] was developed by Munich smart-card maker Giesecke & Devrient, who sold the first 300 SIM cards to the Finnish MNO Radiolinja in 1991. The use of SIM cards became mandatory in the Global System for Mobile Communications (GSM) standard. Each SIM card contains an international mobile subscriber identity (IMSI) that uniquely identifies the user of the mobile network and a unique symmetric cryptographic key (K_i) assigned to it by the MNO during SIM card personalization. The SIM ensures the integrity of the IMSI and K_i, and the confidentiality of K_i. K_i allows the MNO to authenticate the SIM card when the mobile phone connects to the network. When the mobile phone connects, it obtains the IMSI from the SIM card, and requests network access by transmitting the IMSI to the MNO. The MNO looks up the corresponding K_i of the IMSI from its subscriber database, and generates a random nonce as a challenge which is transmitted to the mobile phone. The mobile phone passes the challenge to the SIM card, which signs it, and returns the signed response, which is transmitted back to the MNO by the mobile phone. The MNO compares the signed response to the response calculated using the MNO's copy of the K_i. If they match, the authentication is successful.

Modern SIM cards are based on tamper-resistant universal integrated circuit card (UICC) technology [238] similar to smart cards. UICC cards can host multiple software applications, typically developed using Java Card software technology [98]. The applications include a SIM application for GSM, and universal subscriber identity module (USIM) for UMTS (3G), Long-Term Evolution (4G), and 5G network authentication. MNOs can also provision additional value-add applications to UICC cards that they issue, such as mobile banking and phone-based money transfer. UICC application can interface with mobile phone users or initiate actions via a card application toolkit (CAT) part of the mobile phone operating system (OS): SIM Application Toolkit (STK) for GSM systems, and USIM Application Toolkit (USAT) for later generation networks. UICCs can support an optional bearer independent protocol (BIP), which allows MNOs to deliver over-the-air (OTA) updates to UICC applications either via cell broadcasts, or short message service packets.

All UICC applications are subject to authorization by the issuer security domain (ISD), namely the MNO who issued the UICC. Consequently UICCs are

effectively closed application ecosystems; it is not possible for third-party developers to leverage UICC security without co-operating with MNOs in their region. This puts add-on services operated by large MNOs into an advantageous position compared to third-party alternatives, as is the case with M-Pesa [179], a money transfer application operated by Safaricom and Vodacom, the largest mobile MNO in Kenya and Tanzania. In developing countries, such as Kenya, low-cost feature phones are still prevalent, and UICC applications is the only ubiquitous application platform available to the majority of mobile phone users. Proprietary SIM overlay technology (a.k.a. "slim SIM" or "skin SIM") [190] can enable third-party applications to operate independently of the underlying UICC.

The SIM overlay is a computer chip embedded into thin plastic sheet that can be placed on top of a standard UICC card within a mobile phone. They were originally developed to support low-cost mobile roaming for Chinese customers traveling outside their home province. The overlay SIM acts as an independent security device, and allows additional functionality to be added to any mobile phone by attaching the overlay SIM to an MNO-issued UICC. However, an overlay SIM also has the potential to facilitate a man-in-the-middle attack by observing sensitive data such as personal identification numbers (PINs) being transmitted to the underlying UICC, or initiate, intercept and/or block mobile communications or CAT instructions [130]. By obtaining unauthorized access to the UICC SIM applications they could also change MNO configuration settings.

Embedded SIMs (eSIMs) [132] are secure elements physically integrated into a mobile phone. eSIM chips can be directly soldered onto the device or even embedded into the system on chip (SoC) itself. This physical integration necessitates MNO SIM or USIM profiles to be remotely provisioned. Additionally, unlike removable SIM cards, a single eSIMs may need to store multiple MNO profiles simultaneously.

2.3 Processor secure environments

Towards the late 1990s, mobile phones were transitioning from closed systems to open application platforms, for which third-party applications could be developed using the Java programming language. While not yet true smartphones, the feature phones of the time were gradually starting to resemble small, general-purpose computers. This brought with it new business opportunities, but also new challenges for device security; regulators and MNOs needed to ensure the protection of certain pieces of information after the mobile phone had left the manufacturing line. In particular, regulators required that the device identity, the international mobile equipment identifier (IMEI), remain unchanged in order to act as a theft deterrent. IMEIs of stolen mobile phones are blacklisted by network operators, thereby reducing the economic value of stolen mobile phones and deterring theft [131].

Similarly, radio frequency parameters, which could affect the quality of ser-

vice of other mobile phones in the area, or the safety of the user, should also remain unchanged. MNOs, who were the primary customers of large original equipment manufacturers (OEMs) such as Nokia, were concerned with ensuring that their subscribers receiving subsidized mobile devices do not break their contract terms. Consequently, they required a strong *subsidy lock* mechanism (colloquially known as SIM lock), which would tie the mobile phone to a particular MNO for the duration of the contract. Another emerging use case was DRM for digital content sold by the MNOs; initially ringtones, later games and music.

Nokia was the first to pursue a hardware-enforced processor secure environment. At the time, the security of Nokia's Digital Core Technology (DCT) generation phones was mainly based on obfuscated software solutions and protected by secrecy within the organization; even within the company, only few security professionals knew the exact design and requirements of the DCT security architecture [178]. The leading market share of Nokia made it an attractive target for hackers who, (typically for a small fee) would "*unlock*" or "*unbrand*" subsidy-locked phones by either reverse engineering the valid unlock codes, or reflashing the phone with a different firmware version.

The fourth generation of DCT mobile phones included hardware components in the form of one-time-programmable memory to aid in the secure storage of sensitive device parameters. However, in the case of SIM locks, the economic motives to break device security were higher than the capabilities of the protection mechanism deployed at the time. Consequently, the revenue losses of important MNO customers resulting from SIM unlocking, increased the pressure to design a better security architecture for the upcoming generation of Nokia phones.

Within Nokia the idea of a coherent, hardware-enforced platform security originated within a team of engineers working with mobile payments and security [178]. Initial designs revolved around introducing a discrete security co-processor to ensure the physical isolation of the security-critical operations. However, the additional hardware chip in the bill of materials was deemed too expensive in the extremely cost-conscious organization, whose competitive advantage largely stemmed from its ability to keep manufacturing and components costs in control. Instead, Nokia engineers opted to implement a logically isolated secure processing mode within the main central processing unit (CPU). This solution was not only more cost effective in terms of component costs during manufacturing, but also functioned as common hardware platform for solutions to different use cases. This processor secure environment [94] would form the cornerstone of Nokia's Baseband 5 (BB5) generation mobile phone security architecture.

Initial hardware designs were based on the Nokia's own radio application processors (RAPs), but from very early on Nokia collaborated with the U.S. semiconductor and Integrated Circuit (IC) manufacturer Texas Instruments (TI) with whom they had a close partnership at the time. The first BB5 mobile phone, the Nokia 6630 (codename "*Charlie*") was based on TI's Open Multimedia Applications Platform (OMAP) processors based on the ARM architec-

ture. TI would brand the processor secure environment technology initially developed jointly with Nokia as M-Shield [248]. It was however in Nokia's interest to ensure that it could invite bids from multiple hardware manufacturers for processors implementing a security architecture meeting Nokia's requirements. This became possible around 2003, when ARM proposed to develop system-wide hardware isolation architecture for secure execution for the ARMv6-A application processor architecture which included security extensions to the ARM SoC covering the processor, memory controllers and peripherals. ARM's design would become known as TrustZone [6]. Integrating TrustZone in ARM processor architecture would ensure that any semiconductor manufacturer that implemented the TrustZone security extensions could supply Nokia with processor chips that met their requirements.

2.4 Trusted execution environments

In Chapter 1, we introduced the notion of TEEs – intuitively, a TEE is a computing environment on a device that a relying party trusts to a greater extent than the rest of the software running on the same device. Consider a device running a general-purpose operating system and applications, which, following standard practice, we will refer to as rich execution environment (REE) [119]. For the purposes of this book we deem the device to have a TEE capable of running trusted code, if it has the following *capabilities*, possibly based on hardware support:

1. **Isolation**: the ability to run trusted code strongly isolated from the REE so that the REE cannot influence or learn the computations carried out by the trusted code,

2. **Secure Storage**: the ability for the trusted code to store persistent data guaranteeing its integrity and confidentiality with respect to an adversarial REE, even across reboots, and

3. **(Remote) Attestation**: the ability to convince a (possibly remote) party of the presence of the above attributes, and the characteristics of the trusted software protected by them.

This is an intentionally broad definition. It encompasses both physically distinct components—such as HSMs and TPMs—as well as processor secure environments where the isolation is logical and is enabled by extensions to the processor hardware[3].

TEEs have largely evolved based on business needs, a number of commercial TEEs (Table A.1) have emerged over the years. For mobile TEEs there is a framework of applicable standards, and a core set of these has reached critical mass in industry adoption. Standardization has followed in two contexts:

[3]Sometimes the term TEE is used as a synonym for the particular instance that we call "processor secure environments" in 5. The broad definition we adapt in this book is consistent with the terminology used by GlobalPlatform [119].

1) whenever and wherever common interfaces and application programming interfaces (APIs) are needed for interoperability, and 2) where common agreement for the formulation of the required security level for today's TEEs has been required.

The main standardization organization for mobile TEEs is the GlobalPlatform (GP) consortium[4]. GP provides a system architecture document [119] that describes the main components of the standards set related to TEEs, and how these individual standards contribute to the overall TEE system. Ostensibly the GP TEE architecture is not tied to any particular underlying hardware mechanism for ensuring isolation, but is, in practice, heavily influenced by the ARM TrustZone security architecture. Consequently GP standards are primarily adopted by TrustZone-based TEEs. Enclave architectures (Section 5.2), such as Intel SGX, do not yet have well-defined interoperability specification. But there are on-going efforts like the Linux Foundation's Confidential Computing Consortium which includes projects like the Open Enclave SDK[5] to provide a common development environment across different enclave architectures.

The GP TEE Client API [113] is the common operating system interface (endpoint) to all TEE services. The specification primarily includes APIs for installing trusted applications (TAs) within the TEE, and for allowing REE applications – also known as client applications (CAs) – to communicate with their respective TAs, defining the data interaction model and the session management for this purpose. A separate Debug API, when available, enables a TA developer to receive logs from his TA, and also some post-mortem data in the case of critical crashes.

The GP TEE Internal API [114] is the specification against which TAs are written. For the time being, it provides C-language binding. The internal API defines the transactional model of TAs in the form of a set of standardized callback functions that are called when the TA is loaded, when it is connected to initially, and when it receives an incoming command. The data formats are TA-specific, but communication follows a paradigm of shared memory, allocated by the caller and accessible by the TA, when an incoming message is received. Another aspect of the internal API is the standardized programming framework, a "`libc`-like" interface that provides the TA developer with memory management, secure storage, time, peripheral access and cryptographic primitives. Due to the emphasis on security, the coverage of the cryptographic functionality in the internal API is extensive, and features most contemporary algorithms for public and private key cryptography, symmetric ciphers as well as digest and signature functions. Optional extensions (standards) to the GP internal API includes interfaces to smart cards and embedded secure elements (from within the TEE) [117], APIs by which trusted user interfaces can be setup and controlled [120, 121], and a socket API for network endpoints [118].

For remote administration of TEEs, two separate specifications exist. Both are based on the notion that security domains are established on the device in

[4] `https://globalplatform.org/`
[5] `https://openenclave.io/`

a hierarchical fashion, after which the lifecycle of a security domain can be remotely managed, and secrets (data) and TA codes can be remotely provisioned to it. The two variants are the TEE Management Framework (TMF) [115], and the Open Trust Protocol (OTrP) [208, 116]. The latter is specified both in the context of the GP consortium [116] and in the context of IETF [208]. Even though both protocols accomplish the same thing, TMF is better suited to off-line (or store-and-forward) provisioning, whereas OTrP is explicitly an on-line protocol.

Another provisioning standard, used for virtually all smart cards with application update functionality (including UICC cards) is GP's *Card Specification* standards [110]. These define the card commands by which software can be provisioned to the smart cards, and how security domains, i.e., keys identifying a certain card context, are managed. The secure communication between the provisioning entity and the card, as used by the Card Specification standard, is defined in the GP Secure Channel Protocols [112].

Part II

Mobile Platform Security: How?

Chapter 3

Operating System Security

Operating system (OS) security constitutes the core of software protection in computing platforms. In terms of security, the OS on a device guarantees code and data integrity in the presence of potential attacks, and provides isolation for OS services and applications as well as for system and user data residing on the device. In this context, we use integrity to mean that the data on or the operation of a device shall remain in accordance with the expectations of the device platform provider and the user. As a consequence, unauthorized or accidental modifications of data or deviation from the intended operational flow do not happen. By isolating data and/or processing into isolated domains the OS can provide higher levels of integrity protection, in that potential integrity violations can be contained within a single isolated domain. More prominently, isolating processing together with its associated data is a foundation for information security. For example, such isolation is used to keep secret keys inaccessible to all code except for the cryptographic operations that operate on them. Additional hardware-supported mechanisms like trusted execution environments (TEEs) augment the technical means for implementing confidentiality, access control, authentication, privacy and communication security. Such features can protect the system against attacks, but also be used to implement vertical security services (such as payment or user identification services) needed in computing devices. To achieve device integrity and isolation, trustworthy device boot-up is also crucial, as most isolated environments are set up as part of it. Integrity and isolation are very much intertwined mechanisms in a typical OS.

We can identify two types of protection on a device: user (data) protection and system protection. System protection features aim to protect the computing system against accidents or malice. A good example of this is the separation of privilege, where higher privileged components, such as the OS kernel, are protected (e.g. via memory protection settings) from modification or data leakage induced by software applications. The governing principle is that the system design aims to reach a state of "least privilege", where each active component in the system only is given the processing capability and data access

that are strictly needed for its own operation. If we can reach this idealized state then the system becomes safe (protected) by design. Data protection features, such as user file encryption, primarily aims to protect user data - while in use, in transit or while in storage. In practice, data protection features rely on system protection to guarantee their secure operation. But also a reverse relationship exists: A working system relies on the existence of data such as configuration or log files, and the data protection of these entities becomes an inherent part of system protection.

Many of the security mechanisms related to OS protection were invented, described and implemented in the 1960s. We will introduce them in the form of a short historical cavalcade, covering integrity, isolation, different forms of access control, and authentication. In the second part of the chapter we will explore hardware-assisted run-time protection for the OS kernel. Some other elements of OS hardware-assistance such as cryptographic acceleration for storage protection or key management for device authentication or secure boot are deferred to later chapters, especially Chapter 4.

3.1 General concepts

3.1.1 Integrity

The term "bug", dates back to the development of the Mark-II computer at Harvard in 1947 [194], where a moth trapped in the computer resulted in incorrect computations. This was an instance of accidental integrity violation, where the intent of the computing logic was not met. The concept of integrity was well understood right from the beginning of computing history, starting from the introduction of the stored-program computer [259] in which the memory technologies used often incorporated integrity checksums in hardware simply because the bit error rate at the time was high enough to cause real problems without adequate mitigation. Memory integrity can be ensured either by hardware mechanisms or using software techniques like cryptographic hash functions. In the 1970s, hardware mechanisms to validate the integrity of running programs [99] were introduced in computer systems. But only in the 1990s did cryptographic hashes start to be deployed [17] to protect against *intentional* integrity violations. Today, system integrity validation and enforcement for executing code is often divided into static (off-line) and run-time integrity. The former guarantees integrity at system or program start-up, based on the code stored on persistent storage. The latter provides integrity during execution, based, for example, on measuring snapshots of memory or configuring memory properties (like making executable code memory non-writable). In practice, integrity mechanisms need to be rooted in hardware as we will discuss in Chapter 4.

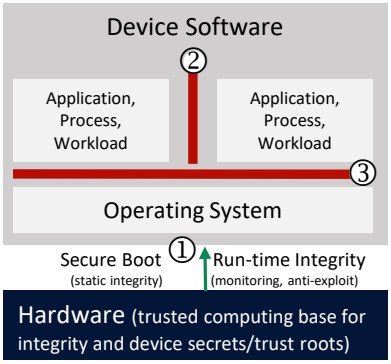


Figure 3.1: (1) Boot time integrity of the OS is achieved statically by securely booting the platform. Run-time integrity can be used to monitor platform state as it executes. By leveraging hardware features like memory virtualization, the securely booted platform can enforce (2) isolation among workload instances (like applications and processes), as well as (3) isolation between the workloads and the system. The latter is typically structured using hierarchical privilege rings where the workloads are isolated from the system whereas the system does have access to workload memory.

3.1.2 Isolation

The need to isolate applications arose from the timesharing computers of the early 1960s. At that time, applications or processes were executed on computers originating from different users and programmers with no mutual trust—what was needed was a way to isolate the code and data from different stakeholders so that software bugs in one workload would not disturb other workloads, or that intentional data theft between computer users would not happen. Already in 1965, Jack Dennis [88] proposed a memory isolation solution that was ahead of its time. The paper proposes a "protection sphere" for programs (which includes a piece of software, its data and its run-time state), and achieves this isolation property using hardware-assisted memory reference protection, using what we today call a "Memory Protection Unit" (MPU). Furthermore, the paper identifies which attributes should apply to which types of memory (e.g., code memory should allow execute privilege, not write)—an insight that was not realized in consumer devices until about 2015—50 years later. For completeness, it was always known that isolation can always be achieved by full device interpretation in software (emulation), but the performance of such solutions then (as well as now) have always been inferior to hardware- assisted isolation.

The concept of privilege levels (also known as rings) was introduced three years later [129]. The logical difference between memory isolation and hardware-assisted privilege levels is that operations (code) residing in a higher privileged ring, has full (or significant) access to memory and resources in the lower privileged rings, but not vice versa. So in essence the presence of privilege rings models isolation in one direction. This mechanism allows for an obvious solution for information flow between rings (higher privileged one copies data from/to the lower one). This is the main protection mechanism used in operating systems today with respect to to applications running on the platform, as visible e.g. in Figure 3.1. The integrity protection derives from a *trust root*, i.e. data or components using which trust in a system can be bootstrapped. For example, a manufacturer signature verification key can serve as a trust root for boot-time integrity. The introduction of hardware support for virtual memory in the 1980s (in the form of memory management units (MMUs)), in conjunction with the notion of privilege levels, brought about the main memory isolation paradigm for operating systems: The OS kernel runs at a higher privilege level, providing isolation between the OS and applications, but allowing for communication between them and the OS kernel. When context-switching between applications takes place, the MMU is reconfigured for each scheduled application, in turn achieving hardware-supported isolation between applications.

3.1.3 Access control

In the presence of proper isolation mechanisms, the notion of *access control* is to stipulate that only authorized entities (such as users) shall have access to

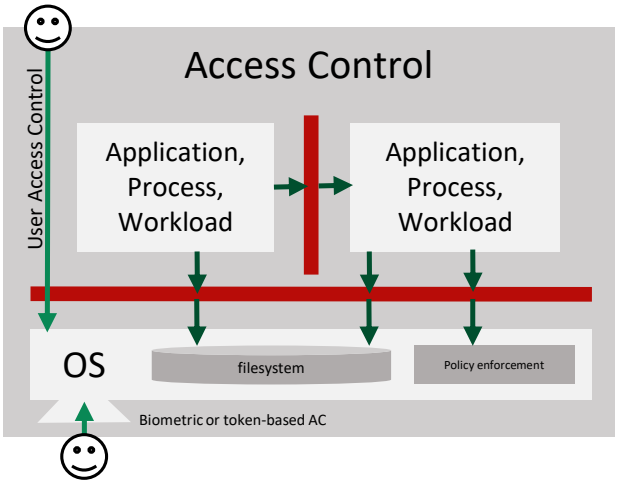


Figure 3.2: System access control takes place both as user access control and access control between system components. Users can authenticate themselves to the system either using system I/O such as the touchscreen (PINs, patterns) or through access-control hardware such as fingerprint readers. System access control can be set up between workloads (to provide controlled access through isolation boundaries) or to selected system resources (e.g. the filesystem). For both user and system access control, the definition and enforcement of access control policy within the system plays an important role. The integrity of the policy itself is also crucial to system security.

computer resources (such as files). For example, by default data from one user should not be accessible to other users. At the same time, full isolation is not desirable, as modular programming systems includes program modules as well as data storage that should be shared between user workloads without sacrificing data isolation properties. The seminal reference for access control [160] puts this in context - whereas the original motivation for isolation in a time-sharing system was "to keep one user's malice or error from harming other user"; today the interpretation is that all of the above reasons for protection are just as strong if the word *user* is replaced by the word *application*.

Access control mechanisms are classified into two types: *discretionary* and *mandatory*. A mechanism is discretionary when subjects such as users, and applications operating on their behalf, can configure access control policies based at their own *discretion*. On the other hand, if a central authority (administrator) decides on a system-wide policy, typically with the intent to protect the system itself against malice or error originating from the subjects, then the access control is mandatory. In mobile phones, mandatory policies often originate from OS providers or original equipment manufacturers (OEMs). In most devices we also distinguish *user access control*, where the subject that is granted access is the user from the case where the subject, in accordance with the insight above, is a software entity, like an application, in a local or remote device. From a technical perspective, the main difference between the two is how the subject is identified or authenticated – the identity of a human being is notoriously difficult to confirm with high accuracy from within a computer system, a program subject can be labeled by some universal identifier such as the digest of the program, the integrity of which can be using device integrity mechanisms in a straight-forward manner. For remote access, we can further rely on remote attestation primitives to transfer the integrity knowledge between physically separate devices or computing environments. The solution for all of these subject identification problems, as well as maintaining the integrity of device access control policy, often uses of hardware security assistance.

The introduction to access control would not be complete without the model Butler Lampson defined in his 1974 paper on protection [160]. He introduces the concept of the *access matrix*. This is a sparse matrix that lists what access attributes (say read / write / execute / modify policy) each subject (a row in the matrix) has for each object (a column). A subject could be a process and an object a data file. See Figure 3.3 for an illustration.

Based on the matrix, the main problem of defining access control for a computer system is a 3-dimensional:

- Decide how and where in the system is it relevant to perform an access control check, i.e., where to place the policy enforcement point? For many mandatory access control schemes, the system call interface (between the application and the OS kernel) is an appropriate place. For more abstract application permissions (say granting access to a camera), libraries or OS daemons maybe more appropriate enforcement points. In microkernel architectures, parsing inter-task messages for identifying system com-

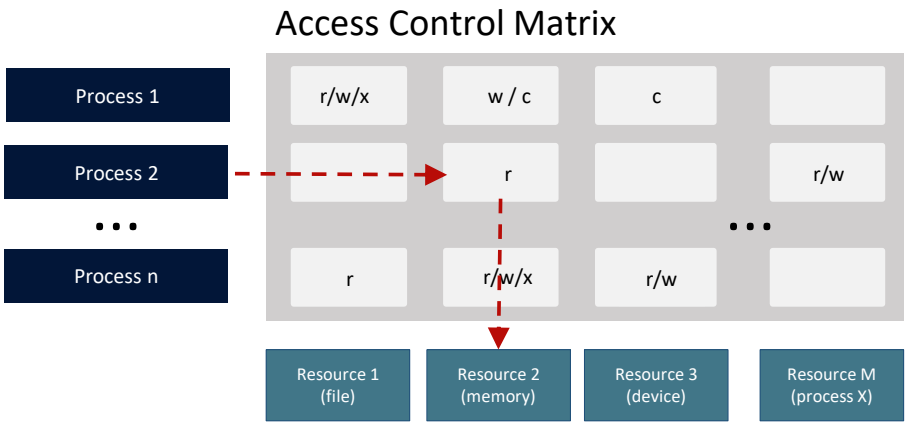


Figure 3.3: The access control matrix defines which access attributes are permitted when a subject (a user, a process, a program), at one row in the matrix intends to access a resource (system hardware, file, a software service) defined as one column. In this example, process 2 has only read access to memory resource 2. The access control matrix is a theoretical construct, and actual access control enforcement mechanisms and policy descriptions are optimized instantiations of the matrix.

mands to be access controlled can be the right approach.

- Centralise access control decisions into one *policy enforcement point*, a *reference monitor*. Having all the policy in one place makes it easy to configure, update and enforce rules and to possibly accelerate decision making using caching of earlier decisions. One well-known example of this is the FLASK architecture [171], which was conceived when the Linux OS was first equipped with the SELinux mandatory access control system.

- Abstract, minimize, and define a policy system and a language by which the access control matrix can be configured and defined in a practical manner. E.g., considering that a mobile phone can have thousands of subjects accessing up to millions of objects, it is not realistic to configure policy for every pair of subject and object separately. Over the years many systems have been deployed to implement this abstraction. *Access Control Lists* define for each object which subjects can access it, with the assumption that if a subject is not listed, access is not granted. The opposite, a *permission*, defines with each subject what objects (or services) it has access to. A *capability* is similar to a permission, but can be passed around (or delegated) between subjects. Different forms of *level-based* architectures define security levels for subjects and objects and decide access right based on the relative levels configured for a subject-object pair. *Domain-Type*-architectures minimize the matrix by clustering subjects and objects together based on how similar their access rights are to one another. Like these examples show, the variation in policy mechanisms is huge, and furthermore, different *policy languages* can then be applied for all of them to describe the policy in some, often human-readable, form.

Access control mechanisms of contemporary mobile devices constitute a hardware-supported set of mechanisms that integrates user access control with the access control needed for granting permissions while isolating system- and 3rd-party workloads on the the device. Now we will look at the commonalities of the mechanisms used in mobile OSs today, with particular emphasis how these are supported by, or guard access to, hardware security mechanisms. An overview of access control principles in end-user devices can be found in [226].

On Android, the highest-level access control mechanism, used primarily by 3rd-party and system applications, are the Android permissions [123]. Permissions are lists of rights associated with an application. The (requested) permissions are listed in the application package metadata, but are granted based on a variety of rules explained below. The permissions are enforced during run-time, and relate to service access beyond the default application sandbox. Examples include network access, storage access and accesses to physical sensors and events such as location, acceleration or microphone / loudspeaker.

The approach to granting permissions has evolved over the lifetime of mobile OSs, starting with Symbian 9 capabilities, and today visible in Android

and the Apple iOS operating system (iOS). Permission granting and the granularity of permissions needs to balance user understanding and ease-of use against achieved level of security and privacy. Earlier systems distinguished between user-granted permissions and signed system permissions. The first set was originally presented to the user at application installation time, and required acceptance in order for the application to be installed. The system permissions set has always been under the control of the OEM or ecosystem provider—these provide access to system features whose acceptance is not under the discretion of the device user, and therefore require an associated cryptographic signature by a trust root in order to be activated for the installed application. A typical example of such a permission might include the right for the installed application or service to function as an (extended) installer for other applications.

User permissions in Android 10 and iOS 13 are divided into install-time permissions and run-time permissions. While install-time permissions can be reviewed and accepted at install-time, run-time permissions are requested (from the user) in situ, when the application is running and attempts the operation requiring permission. This mode of operation intends to help the user contextualize the request, as the dialog is shown specifically when it is needed. When displayed, the user may choose to grant the permission to the application permanently, or every time the permission is needed. The user can later change the set of permissions granted to the application via centralized application management.

The latest version of iOS can also distinguish permission usage based on whether the application is currently used (in the foreground) or all the time. Furthermore, to protect users' privacy, the OS has the ability to audit and potentially raise an alarm to the user if hidden or background applications excessively exercise a permission such as location access. Today both dominant mobile operating systems carry a total of 100–200 permissions that can be set and operated on by the user, the application developer and the OEM. It is likely that the current approaches strike the right balance between the complexity of the permission granting process and the security and privacy guarantees provided by permissions.

The next level of access control in the mobile OSs happens at the level of process interaction with the kernel, as part of kernel system calls. Some of the permission checks discussed above may be enforced at this level, but more to the point, this is the level where the sandboxing of 3rd-party applications takes place – to isolate these from each other, but also to control the privileges of such sandboxed applications with respect to system daemons, driver frontends and other system-critical functions implemented as high-privileged user-space programs. For iOS, the sandboxing protection is implemented on top of the TrustedBSD framework [260]. TrustedBSD is a kernel framework that allows for policy modules to be hooked into the kernel, i.e. kernel system calls are augmented to accept access control decisions from the policy module, which also can account for subject identification such as process labels (applications). In some cases the policy framework can also invoke user daemons to collect

extra information for access control decisions. The iOS policies [58] are written in tinyLISP: some 100+ mandatory access control enforcement points in the iOSsystem calls make up the (container) isolation framework for the application sandbox and potential exceptions for e.g. system daemons.

Android has a similar mandatory access control enforcement mechanism. It is built on a similar hook-based mechanism as with TrustedBSD, called Linux Security Modules (LSM) [191]. The selected policy engine for Android is SE-Android, a Domain-Type policy enforcement mechanism inherited from the SELinux project [241][128]. The Android Linux kernel imports the system policy at boot—it is loaded from the system ramdisk which in turn is covered by the secure boot of the mobile device. The policy is transcribed from a set of M4 macros via a SELinux policy language into binary representation entered to and used in the kernel engine. Policy can also be applied to the filesystem metadata and to application-side callbacks, just like with iOS. In Android, part of the policy originates from the OS provider (Google), and part from the OEM, the latter providing policy for accessing OEM-specific hardware.

3.1.4 User authentication

User authentication refers to the process of identifying followed by verifying the identity of the user of the system. Often, the mobile device is considered to be personal, i.e. the identification step is implicit. Authentication traditionally consists of the users entering passwords, PINs or graphical patterns to prove their identities. The mechanism needs to be able to securely distinguish correct authentication attempts from wrong ones. Typically the access control framework as a whole also incorporates related aspects like elements of inactivity control, forcing re-authentication after a period of inactivity, and managing multi-factor authentication. On the other hand, system protection, such as extending the inter-authentication attempt interval, is also needed in the occurrence of too many failed login attempts. Depending on policy, it may also result in wiping local data or requiring additional (more secure) authentication in case of repeated authentication failure. In recent devices, the access control system also orchestrates file decryption ability, i.e. access to cryptographic key material for proper device operation may be limited when a device is in a locked state.Underlying hardware support for secure storage can facilitate the implementation of all of these different policy variants.

Proof of identity submitted by the user can be broadly categorized as 1) “Something the user knows” such as PINs, 2) “Something the user has” e.g an external NFC authentication token or 3) “Something user can be characterized by”, e.g. a biometric fingerprint. Biometric authentication has become the dominant authentication mechanism in mobile devices, driven by ease-of-use considerations. This has resulted in changes to the hardware security mechanisms appearing in mobile devices.

Fingerprint sensing is one such biometric authentication mechanism. Three fingerprint sensing technologies are popular: optical, capacitive and ultrasound. An optical scanner can be considered to take picture of the fingertip, and the

print pattern is then processed into an identification code. A capacitive scanner measures electrical signals sent from the finger to the scanner, thereby mapping out contact points between lines, ridges, valleys and air gaps in the fingerprint. Today, capacitive scanning is dominant. Ultrasonic scanning sensors, deployed in some high-end mobile phones, augments the capacitive technology by sensing the 3D profile (depth) of the print's ridges and valleys which in in theory should result in higher accuracy. For non-optical sensors, the accuracy of sensors deployed in phones is more than 98% per finger for a false positive rate of 0.01.

Facial recognition systems are the latest addition to user authentication with mobile devices. Facial recognition is achieved by extracting facial features from a potentially augmented camera image. Features can include distances between eyes, lips and nose, as well as a 3D map of the face. The iPhone FaceID unlocking mechanism uses 30,000 infrared dots [16] to create an approximate 3D map of the user's facial features. Facial template matching is conceptually similar to fingerprint matching, and shares similar concerns: An important platform challenge related to all biometric authentication such as fingerprints or facial recognition is how to keep the reference template private (secure) and strictly local to the device. Neither Android nor iOS backs up templates onto cloud servers. Furthermore, template matching happens inside a TEE, equipped with secure storage where the template is stored. An important consideration is how data from the biometric sensor is channeled from the sensor to the matching operation in the TEE, so that no obvious locations for harvesting attacks are present on the system. Finally, life-cycle processes must assure the deletion of old biometric templates should the device be re-sold or scrapped.

In recent mobile devices, context-based user authentication decisions augment the above mechanisms, or potentially the selection of which authentication mechanisms are required for access in different usage scenarios. For example, Android SmartLock implements such an architecture, deployed by OEMs such as Samsung and Sony [184]. Context used for access control decisions can include geographical information, Bluetooth and WiFi access points in the vicinity, temperature, light sensor readings and even weak authentication input such as user gait (measured via accelerators) and touch-screen typing patterns . Machine learning can also be leveraged: "Familiar" patterns regarding the user's context can be recorded and taught to the machine learning model, which is then be used to augment access control decisions, typically allowing for better usability in case context familiarity is high [186][146].

3.1.5 Software authentication

Although mobile devices opened up to allow third-party software in the 1990s, installation of third party applications was initially constrained due to business reasons as well as security reasons (protection against malware). Over time, the secure installation of applications has come to follow a certain pattern:

- **Signed application packages**: The application software is a *package* containing a mix of executable binary code, code for various runtimes such as Java, libraries, graphical elements such as icons as well as program data. The packages are put together, and typically, digitally signed by the application software developers using their signing keys. The mechanisms, and sometimes the infrastructure, needed for application signing are defined and managed by the OS/ecosystem provider. The package also lists the expected *permissions*, e.g., the access to platform services needed by the program when it is eventually installed on a device.

- **Publishing software via application stores:** The developer submits the *package* to a centralized store, often operated by the ecosystem provider. The store is not only a software repository, but also an active part of securing the device platforms. On submission, the software package can be validated for its functionality. The application can be analyzed manually or with machine learning methods to ensure that it is benign, e.g., that it does not over-consume resources or masquerade as some other software. Virus scanning can be performed on it to weed out unintentional vulnerabilities or intentionally malicious behavior. The store may also contest some of the requested permissions, if they violate some business or other constraint, e.g., an application requesting the the following combination of permissions—networking, microphone, and loudspeaker access—will allow it to serve as an endpoint in a voice communication channel; historically, this has not always been allowed in all device ecosystems. When all its checks are successfully completed, the store publishes the application for download to mobile devices.

- **Restricting installation sources:** The store identity is an integral trust root assigned and configured in device software at manufacturing. By default, application download is restricted to the stores decided by the manufacturer, although some devices do allow the user to add new trusted stores, which is a very similar or the same operation needed when configuring an enterprise device-management server in the device, or to *sideload* applications by installing them locally without going via a store. When stores do adequate checking on submitted application packages, and devices exclusively use authorized stores, the resulting security benefit has proven to be substantial. For example, in Europe and North America the problem of fake applications or malware in application stores is virtually non-existent.

- **On-device validation of application packages:** When an application package is downloaded onto a device, over an authenticated connection to the application store, local application installation happens. Again, a public key infrastructure (PKI) is leveraged for package authentication, but the root of this PKI varies among ecosystems. The application may be verified using a certificate chain up to the ecosystem provider, starting with the developer signature, or in some cases the store may countersign the

package. The chain may also consist of a single element, the developer signature, e.g. Android deploys a solution called *same origin policy* where the first installation of an application can be from any source (developer), but application upgrades are then enforced to originate from the same source. Depending on the ecosystem, the local installation is done into some form of sandbox, i.e. by default a third-party application does not get full system access. It should be well-isolated from other application running on the device, but also from the system based on the permissions applied for it by the developer, granted by the store, and potentially further approved by the device user at installation or at application usage time. An important part of this sandboxing is the part of the filesystem available to the application, which is often severely restricted. Also, as part of the installation, some of the metadata related to the application such as application name, trust roots, permissions, inter-process communication intent or the system events the application is supposed to react to are as a rule added to some form of local registry for later application management in the device.

- **Over-the-Air Device Management:** Although more prevalent in enterprise settings, the application store can also incorporate some *over-the-air device management*. The store may remember downloads by individual mobile devices, and orchestrate the updating of the software when new versions are provided by the developer, this is especially relevant for programs that are purchased (i.e. not free of charge) Also if security problems are later found with a program, the device management system can warn the device and its user, or even forcibly remove the offending application.

The details available related to application installation security and the ways e.g. the PKI is set up for specific ecosystems vary across OSs and ecosystems. The principles above have remained constant for mobile OSs since Symbian 9, including the currently prevalent Android and iOS. To a large extent the principles also carry over to home appliances with app support such as SmartTVs. A common property of secure application installation is that it does require that the trust root of the store is configured in the device at manufacture and potentially also the PKI trust roots related to application signing.

3.1.6 Storage protection

For data-at-rest protection in mobile OSs, two requirements intersect: The first is *integrity* – ensuring that modifications to applications as well as system code and data cannot be undetectably modified, even when the device is powered off, The second is *confidentiality*, especially of user data. The architectures for file (system) protection includes Android *DMVerity* [124] that is applied to the system partition since Android 7. Android boots with the Linux kernel and the initial RAMDisk protected by secure boot integrity measurements. The data in the kernel configuration and RAMDisk is therefore in the trusted computing

base (TCB), and can include trust roots for further operations. The DMverity block-based file-system integrity mechanisms leverage this fact. The file system is concatenated with a hash tree, representing the blocks in the read-only system partition. The root of the hash tree is further bound to the kernel/RAMDisk. Using a hash tree for block verification contributes to boot-up efficiency, by allowing the filesystem integrity to be validated on individual block loads, rather than all at once. The same principle is extended to a file-based variant called *FSVerity* [125], a Linux kernel file system plugin, that implements the same structure for a single file. In this case, the hash root is managed separately, e.g. by managing it within a certificate, as is done in Android 10 and later. FSVerity is a block-based mechanism to achieve integrity for a file residing in an otherwise read/write system, and by being block-based (with the hash tree validation) on-demand paging is trivially supported. In Android, FSVerity can be used by system and third-party application packages alike, and allows e.g. for application updates where a new system application overrides the default one in the read-only filesystem (without sacrificing integrity). In iOS, the root filesystem is also mounted read-only.

For user file-system confidentiality, potentially the most potent protection regime is the "Data Protection" mechanism of Apple iOS [10]. This mechanism can even protect individual parts of large files (as supported by Apple's APFS file system) with different keys, but even without this extension, the file protection (implemented using AES-XTS-128 encryption mode), works in a file-based manner as follows: On read or write, the kernel identifies the file, and reports this identity to the isolated Secure Enclave Processor. The SEP manages and derives key material for the file in question, and configures the key material (diversifier and key) directly to the hardware accelerator for encrypting or decrypting the data. In this operational pattern, file keys are never exposed to any part of the OS kernel, and therefore not vulnerable to leakage [42]. Contemporary Android devices also follow a file-based encryption pattern (Android FBE). The way to handle key management varies across OEMs. E.g. Google pixel devices [127] manage the file-specific keys in the Linux kernel key-ring, but leverage a hardware accelerator to encrypt/decrypt. Samsung and Huawei devices also use hardware acceleration for encryption. Specific to Android FBE are the two protection modes related to the status of user authentication in the phone. Data by applications can be protected by Credential Encryption (CE), whereby the file keys are available only on successful device unlock, i.e. access to such files is dependent on the user authentication status (when the device is locked, access to files is not possible). The alternative is device encrypted (DE) data, which is available also without device lock, and allows applications to function early at boot on in the background (when the device is locked). Block-based filesystem encryption (protecting all reads and writes to flash, for a filesystem, with a single key per filesystem) was used in the past in mobile phones, but is not in widespread use any more. Samsung Knox devices [225] include the Dual-Data-at-Rest solution, which provides double encryption for selected files, the main use being that the OS can arrange the second level of protection to be outsourced to a plugged-in (separate) encryption module, pro-

viding e.g. government-certified encryption as an option.

3.2 Run-time Hardware assistance

In legacy mobile devices, additional hardware-assisted protection has been added to protect the OS, and especially its kernel. Apple has deployed *Kernel Integrity Protection* from the time the Apple A11 processor was launched. This is a hardware-assisted processor feature that protects the kernel as a combination of two separate operations. First, the memory controller is augmented with a feature than can write-lock physical memory regions after boot, not allowing unlocking until reboot. Second, the MMU does not allow re-mapping this memory to outside the protected region and conversely, mapping any other writable memory into the virtual memory of the kernel memory region. In combination, these two hardware augmentations allow the device kernel code to be configured as immutable during run-time, effectively protecting against rootkits at this level of operation.

In Android devices, especially those manufactured by Samsung and Huawei, also offer similar features for kernel memory protection, although implemented using standard privilege level isolation rather than with legacy hardware extensions. The ARMvA8 architecture supports two levels of virtual memory page tables, and the ability to control the lower level page translation from inside the hypervisor privilege level (the so-called EL2) allows the protection to be constructed, such as having the Linux kernel code trap to the hypervisor code for kernel memory management. This way, some parts of the virtual memory configuration for the kernel can be outsourced to EL2, and thereby protected by policy. The hypervisor can enforce e.g. kernel code write-protection, double-mapping protection, as well as some data protection for system registers contents, read-only data and security protection for critical process (credential) parameters in task structures. As outlined in Chapter 7, attack mitigations for return-oriented programming (ROP) and jump-oriented programming (JOP) such as call-flow integrity can also be further hardened with these mechanisms. Samsung uses the brand name 'Real-time Kernel Protection' (RKP) for the collection of the features [224], whereas Huawei calls its corresponding set of features 'Huawei Kernel Integrity Protection' (HKIP) [139].

Both Samsung and Huawei complement kernel memory protection (and secure boot) with run-time attestation, that extends beyond measuring kernel data into system services, including user-mode processes. Huawei's Extended Integrity Measurement Architecture is a framework for measuring and reporting on code and data in a way where some of the static measurements are stored and reported from the TEE. Samsung's Periodic Kernel measurement (PKM) measures kernel data like like SELinux kernel polices, and complements its TrustZone-based Integrity Measurement Architecture (TIMA) depicted in Figure 3.4—a framework capable of locally attesting system firmware, and furthermore based on these metrics allow or disallow e.g. user access to system keys available in the TEE [39].

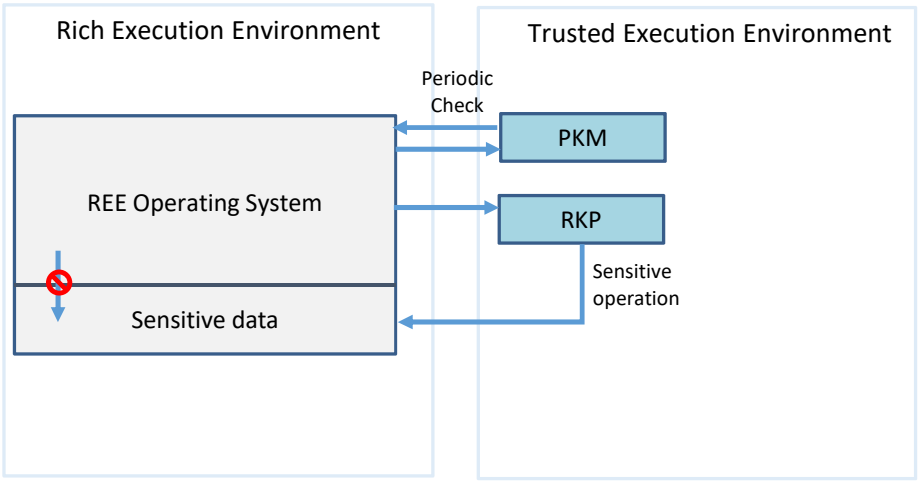


Figure 3.4: TIMA's kernel integrity protection mechanisms. Periodic Kernel Measurement (PKM) periodically inspects the code and critical data of the kernel to ensure that it has not been modified. Real-time Kernel Protection (RKP) intercepts attempts by the kernel to access sensitive data, preventing integrity violations in the first place.

Furthermore Apple A11 and later processors contain a hardware feature by which memory write permissions can be quickly eliminated from application memory pages without requiring kernel-assisted MMU configuration (saving the overhead of a system call and a page-walk). This feature is useful for self-modifying code, e.g. for just-in-time compilers, where executable code is first written, but then ideally executed without memory write permission.

3.3 Conclusions

In this chapter we reviewed OS security with a focus on how isolation, integrity and access control is arranged for in mobile phones today. Figure 3.5 summarizes this in a single figure, but also indicates the presence of such hardware-assisted support that is fundamental enough to merit its own discussion in later chapters. Beyond run-time integrity and file system protection which was considered as part of OS security, Chapter 4 provides a separate discussion on boot integrity, which for the most part is the trust root on which OS security either stands or falls. The hardware-assisted isolation mechanisms and co-processors discussed in Chapter 5 allow for significantly improved control (and protection) of workloads running at different security levels. Especially integrity validation during secure boot relies heavily on hardware-accelerated cryptography, outlined in Chapter 6. Finally, the run-time protection mechanisms introduced in Chapter 7 are an increasingly essential element in the on-

going fight against vulnerabilities at all levels of the software stack (including the OS kernel).

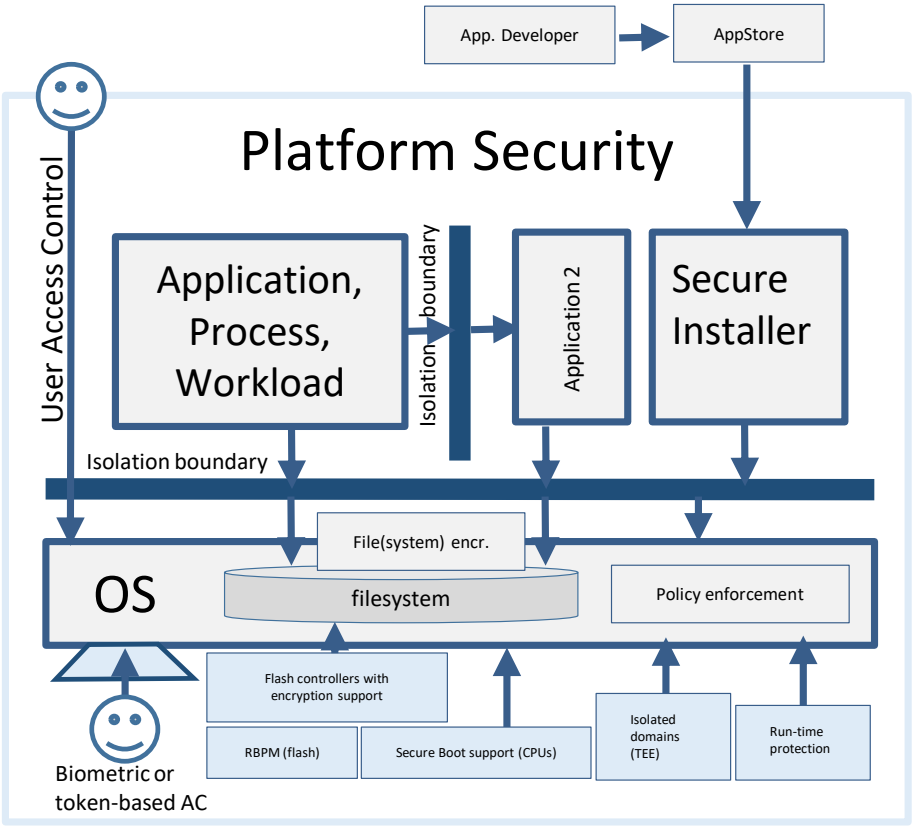


Figure 3.5: Platform Security in context: The OS kernel serves as an orchestrator of security mechanisms in the platform - it upholds memory isolation barriers / boundaries between services and applications and also between user space components and itself. The access control mechanism is in contemporary mobile OSs rooted in kernel access control, even though user space components take part in permission control and user authentication. Filesystem and storage protection is also at large an OS kernel operated activity. However, secure kernel operation relies on hardware support for isolation (TEE), trust roots, accelerators for cryptographic operation and secure boot, for rollback-protection (RPMB) and for user authentication (like finger-print sensors). These components are presented in Chapters 4-6.

Chapter 4

Platform integrity

All of the security features that we describe rely on a correct platform configuration to achieve practically useful results. The platform must boot into the correct operating system (OS) and execute the correct applications. Furthermore, it is not enough to configure the platform correctly. The system must be designed to facilitate remote attestation.

In this chapter, we explore the two phases required for ensuring platform integrity: the supply chain and the boot process.

4.1 Supply chain security

Supply chain security refers to the process of securing the integrity—and sometimes confidentiality—of the software components provisioned to a device, as well as the device hardware itself.

4.1.1 Provisioning

Mobile devices hold various types of immutable data that correspond to device identity, device secrets, keys, and certificates. This data plays an important role in the platform's security needs such as boot integrity, attestation, identity-proving, and software version control. Devices can only be fully-functional when all needed parameters are provisioned.

Provisioning is a challenging process whose design must take into account many factors: the sensitivity of the data, secure data management over the lifespan of the device, and the fact that multiple entities may be responsible for different parts of the data being provisioned (e.g. system on chip (SoC) vendors, original equipment manufacturers (OEMs), and platform providers) [90]. Provisioning happens at various stages throughout the device's life-cycle, such as during SoC design, SoC manufacturing, device manufacturing, and when the device is repaired or refurbished.

Security-critical data must be provisioned in a controlled environment into persistent storage media such as read-only memory (ROM), fuse or flash memories. Throughout the provisioning process, manufacturers must take adequate protective measures to ensure the integrity of the data being provisioned, and to limit access to unprovisioned devices to only trusted entities. The storage type depends primarily on the device's security profile; in mobile devices this data can be provisioned into a read-only flash section or into an on-chip one-time programmable (OTP) memory, but where such storage is unavailable or must be user-replaceable, a dedicated secure element such as a subscriber identity module (SIM) or Trusted Platform Module (TPM) may be used. [95]

The provisioning process for mobile devices starts during the SoC design process, when the contents of ROMs are determined, and continues throughout the manufacturing process. In addition to ROMs whose data is incorporated directly into the SoC design, SoC vendors provision the chip with common configuration data such as the chip revision, model, and manufacturing series, and characterization parameters for hardware blocks such as true random number generators (TRNGs). Commercial SoCs are further provisioned with a security-hardened configuration that disables debug ports and or the ability to override processes like secure boot, and enables code encryption and rollback-protection mechanisms. SoC vendors provide their customers (normally OEMs) with a set of tools and utilities that can be used to further provision the SoC.

In some cases, such as with Apple devices, the same entity is responsible for the SoC, the OS, and the application ecosystem, which allows for a highly integrated provisioning process. In other cases, the SoC vendor's provisioning process is followed by the OEM's provisioning of device-level configuration. For example, an ARM-based device using ARM Trusted Firmware [33] would be provisioned with some or all of the following:

- **Hardware Unique Key (HUK)** (a.k.a. Device Root Key): An immutable key, unique to the device, acting as a root-of-trust (RoT) and the base for other derived device-bound keys used for attestation, device encryption, and secure storage. Access to the HUK is limited to code running in the trusted execution environment (TEE). As the key is specific to the device and accessible only to trusted software, data encrypted with a HUK-derived key is cryptographically bound to the device, in the sense that copying the data to another device will prevent it from being decrypted.

- **RoT public key:** An immutable asymmetric public key, or a hash of a public key, that is used to verify the authenticity of the first code run by the device. The authentication key is normally either an Rivest-Shamir-Adleman (RSA) or elliptic-curve cryptography (ECC) public key and is referred to as the RoT public key. This is used to verify the first-stage bootloader. Depending on the device security model, the OEMs may provision it with several hashes, allowing for separate RoTs where different parts of a platform are implemented by more than one developer.

- **Attestation keys:** Certified key pairs used to demonstrate that a statement or message originated from a particular device. Devices are provisioned with a set of asymmetric key pairs as part of the manufacturing process. These keys are used for 1) platform attestation, to prove the trustworthiness of a device to remote verifying parties, and 2) key attestation, to prove the characteristics of a TEE-generated cryptographic key. One example of key attestation is the ability of Android devices to make security claims about key generation and storage environments. For example, an Android device can attest that a key was generated inside a TEE and cannot be exported from the TEE, or that a key can only be used after biometric authentication. These claims can be trusted, because Google, before certifying a manufacturer's attestation keys using the Google Hardware Attestation Root certificate[1], verifies the device's attestation functionality as part of the compliance checks made on devices that are to be approved for the use of Google Mobile Services (GMS).

- **Device identity information:** Device specific information such as device ID, model, international mobile equipment identifier (IMEI) etc., that is used to provide a unique and immutable device identity.

At least some of these keys are needed for compliance with standards, such as ARM's Trusted Base System Architecture (TBSA) [20]. To be ARM-TBSA-compliant, ARM Cortex-M devices must be programmed with at least a HUK for data confidentiality, and an authentication key for the firmware. Optional platform keys such as attestation and firmware-decryption keys can be derived from these two keys [32]. In addition to the above, OEMs program additional keys and certificates to support third party trusted services such as digital rights management (DRM). The service providers' certificates are stored in non volatile flash memory and protected using either the HUK or one of its derived keys.

4.1.2 Roots of trust

In mobile platforms, RoT requirements are standardised by GlobalPlatform [111]. Originally standardized by the Trusted Computing Group (TCG) in the context of TPM-based systems, the root of trust for measurement (RTM) is an immutable piece of code that runs before anything else on the device, ensuring that the correct code is subsequently loaded and executed.

Since then, a well-established taxonomy [78, 111] has been developed around RoTs to describe the system properties ensured by the RoT on which the security of a TEE is built, and to allow the relative levels of security of RoTs to be evaluated during security certification.

[1] `https://developer.android.com/training/articles/security-key-attestation#root_certificate`

4.1.3 Signing infrastructure

Software updates are digitally signed before they are packaged by the OEM to be installed on a device in the field. Device security hinges on the secrecy of the signing keys used to sign the device images. OEMs must take adequate measures to control access to signing keys corresponding to the device roots of trust, as the RoT is kept in devices' immutable storage and cannot be revoked.

Software updates are accompanied by a chain of X.509 certificates, digital signatures, and a manifest containing package metadata. Rather than including this data in a separate manifest, this metadata can be included in each program binary as an Executable Linkable Format (ELF) segment [89].

Depending on the OEM's software update process and device security model, this metadata can include any or all of the following: 1) cryptographic hashes of independently updateable images, 2) software version numbers, 3) the devices (hardware IDs) for which the images are targeted, and 4) version numbers used to prevent software version rollback. A digital signature may be computed over the contents of just the image metadata rather than the entire image, to facilitate verification in memory-constrained environments. OEMs can also encrypt the manifest with a pre-shared encryption key in order to preserve the confidentiality of their software updates.

The manifest, in whatever form, must contain the following elements for each image contained in the update package:

- **Image metadata:** Image version number, rollback protection information, image size, hash values, and cryptographic parameters used for verification [33].

- **Target device information:** Device, batch or product information for which the software update is targeted. If manifest files are used, this can also contain versioning information of the manifest file.

A manifest might include other parameters in order to aid update selection, so that a single package can be used to place devices into a wide variety of configurations.

The software update mechanism verifies the authenticity of the package by using the supplied certificate chain to ensure that the public key used to verify the package signatures can be linked back to the RoT, which is often a hash of the public key contained in the root of the certificate chain. Having securely identified the public key needed to verify the authenticity of the package, the update mechanism must ensure that a signature covers the entire contents of the package, including any configuration data, version information, software, firmware, and initialization scripts. OEMs often use multiple signing keys, each of which is used to sign data used in a different stage of boot process. It is common practice to use different signing keys to sign normal and security-critical software, with the highest-security and least-convenient processes being reserved for the most critical software. This is particularly useful when software components originate from different vendors, each of which can only

sign the component for which they are responsible. Security-sensitive modules such as secure OSs and critical firmware may be required to be signed by both the software vendor and the OEM itself to ensure the updates are approved by both.

4.1.4 Software updates

The two dominant platforms in the mobile ecosystem, iOS and Android, differ significantly in the way their devices are updated. Apple has a vertically integrated ecosystem, giving them full control of the software update process for the entire device. Android devices, on the other hand, have multiple software providers, each with their own testing infrastructure and quality norms: Google provides the Android platform, OEMs and carriers customize the platform with their own software, and other developers provide applications. Apple's tight control on the update process provides an advantage when responding to zero-day attacks, as they can immediately deploy patched software to all devices. In comparison, the number of developers responsible for Android devices means that the software update process for Android devices is relatively slow and requires significant coordination amongst the developers involved.

This dependence on OEMs to re-customize and deploy updates has led to longer response times to security issues, resulting in a greater number of vulnerable devices. In order to mitigate this, newer versions of Android decouple the base Android system from OEM customizations, reducing the time and effort needed for OEMs to roll out updates[2].

Software updates for mobile devices are rolled out through an over-the-air (OTA) update process involving client and server infrastructure. The software update procedure is initiated by the device and may consist of the following stages:

- **Downloading:** Devices regularly poll the OTA update servers, and when a new update is available, the device downloads a signed update package.
- **Verification:** The integrity of the update package is verified using the mechanisms discussed in Section 4.1.3.
- **Installation:** The new software is installed to the device.
- **Postprocessing:** After successful installation, the update is configured.

In the case of Android devices, OEMs can use either Google's OTA update infrastructure or their own servers to distribute system updates. As well as avoiding the need to run their own update servers, OEMs using Google's OTA update infrastructure avoid the need to provide their own update client[3].

[2] `https://www.androidauthority.com/project-treble-818225/`
[3] `https://source.android.com/devices/tech/ota/ab`

To facilitate the installation of system updates, most mobile devices include two separate partitions. At any time, one is marked as active and one as idle. The active partition holds the currently-running software, and updates are installed to the idle partition. Then, the idle partition is marked as active, and the active partition as idle. This ensures that a working copy is available as a fallback if the software update fails or the device fails to complete the boot process after an update. In the event of boot failure, the device can instead boot from the old known-good partition.

4.2 Boot integrity

It is not enough to download and install only valid updates to a device: the device must ensure the authenticity of the software being executed, so that an attacker cannot execute malicious code by bypassing the update mechanism using a software vulnerability or physical access.

Secure boot and *authenticated boot* [207] mechanisms ensure the integrity of the boot process, detecting attempts to boot an altered system image. Secure boot and authenticated boot are standard features in modern PCs [205] and mobile platforms [35], although their architectural realizations can differ significantly.

4.2.1 Secure boot

In secure boot, each step in the boot process verifies the authenticity of the next software component in the boot chain, before it is launched. The system RoT is used to verify the authenticity of the first software component. Consequently, software images must be signed by the manufacturer before being deployed on the device.

4.2.2 Authenticated boot

In authenticated boot, each step of the boot process is measured, e.g., by computing a cryptographic hash over the platform configuration information and the subsequently-executed software component. Unlike with secure boot, the resulting measurement is not used to decide whether the component can be executed, but only stored securely for later retrieval.

Authenticated boot permits any software to run on the device. This means that any software may attempt to access secret keys or other sensitive resources, but the securely stored measurements can be used for access control, so that access to sensitive resources is only possible after an appropriate boot sequence. Moreover, the measurements can be used for other purposes than access control, such as producing a signed statement of the system's state that can be used to attest the state of the system to a remote verifier.

Authenticated boot provides greater flexibility for multi-function devices, since users can execute arbitrary code, so long as it does not perform sensitive

operations. This flexibility comes at the cost of additional complexity, since access control checks must be added to all sensitive operations.

4.2.3 Rollback Protection

Signing of software prevents unauthorized software from being installed onto a device. But once software has been signed, it cannot be "un-signed", and some other mechanism is needed to prevent legitimate but vulnerable software from being installed and exploited by an attacker. Since mobile devices need to verify the authenticity of software without access to the internet or correct time, solutions developed for web cryptography, such as Online Certificate Status Protocol (OCSP) [227], cannot solve the problem. Instead, mobile devices use rollback protection to ensure that old, vulnerable versions of their software cannot be reinstalled by an attacker[4].

Rollback-protection counters

Rollback protection is implemented by using a designated region of secure storage to keep track of the minimum software version that the device is permitted to boot.

Updating the version information for device bootloaders, firmware and the OS is done as part of secure boot process, often by the bootloader. Rollback counters are incremented when the device boots a version of the software is with a higher version number than the current value of the counter, after the software has already been used successfully. In practice, this means that the first time a new software version is used, it will mark the image as 'successfully used', but not update the counter. The second time the updated image is been booted, the rollback counter will be updated with the version number of the new image. This is needed so that if a new version of the software does not work then it will not update the rollback counter and thereby render the device unusable. OEMs can increase the version number whenever they release a software update addressing a security vulnerability. As each component is loaded, the rollback-protection counter is checked against the version to be executed; if the component is older than the current value of the rollback-protection counter then its validation fails. The result is that once a vulnerability has been patched, an attacker cannot reintroduce it by reinstalling vulnerable software.

Modern day mobile devices execute a multi-stage boot process with many different components. Each of these stages has its own rollback-protection counter, allowing OEMs to independently update each component while still protecting against rollback attacks[5].

[4] `https://source.android.com/security/verifiedboot/verified-boot#rollback-protection`

[5] `https://developer.trustedfirmware.org/w/tf_m/design/trusted_boot/rollback_protection`

Google added rollback protection to Android 8 and made this mandatory for devices from Android 9 onward. Android Verified Boot does the rollback verification checks using the version information in the verified boot metadata[6], and prevents the device from booting if downgraded firmware is detected [7].

Apple takes a slightly different approach, using an online signing server to provide rollback protection for iOS and iPad devices. Rather than containing a simple version number, the anti-rollback memory contains a random value that is sent to Apple when requesting an update. The OTA update server signs the update along with the anti-rollback value and the device's unique identifier. The result is that if a software update package is successfully validated, the device can be certain that it is newer than the most recently installed version of the same component, and that it is guaranteed to be the most recent version available at the time that it was downloaded [15].

Validating software versions against locally stored anti-rollback values comes with certain limitations. In a situation where OEMs have issued multiple security updates in quick succession, a device that has not been updated recently cannot ascertain if an update is the most recent one. In this case, it is possible to run and update the device with software that is newer than the version held in the rollback counters, but nevertheless outdated.

4.3 Secure storage

Devices hold many types of data, including application data, OEM configuration, cryptographic keys, DRM-protected media, certificates, and personal information such as biometrics, messages, and passwords. Different types of data have different levels of sensitivity; they require require protection from different kinds of threats, and have different performance requirements. Fulfilling these requirements in a coherent way requires careful design of the storage subsystem. In this section we discuss some of the tools that can be used to achieve this goal.

4.3.1 Memory Technologies

Secure nonvolatile memories can be classified as multiple-time programmable (MTP), few-time programmable (FTP), or one-time programmable (OTP), depending on their programmability. MTP memories such as flash memory or electrically-erasable programmable read-only memory (EEPROM) can be reprogrammed many times, making them suitable for mutable data such as anti-rollback counters. These memories are built to provide endurances on the or-

[6] `https://android.googlesource.com/platform/external/avb/+/master/README.md#Rollback-Protection`

[7] `https://source.android.com/security/verifiedboot/verified-boot#rollback-protection`

der of a million write cycles[8].

Where reprogrammability is not required, device manufacturers can use OTP memories. Once an OTP memory bit has been written, it cannot be returned to its original state. This makes OTP memory a good choice for data such as RoTs. In addition, OTP memory requires relatively little space on a chip and requires few manufacturing steps, making it relatively cheap. These properties make OTP memory a popular choice for low-cost Internet of Things (IoT) devices that will not be reprogrammed after manufacture[9].

OTP memories can be classified into electronic fuse (eFuse) and antifuse memories. An eFuse is a strip of metal that can be programmed by passing a large current through it, vaporizing the metal and breaking the circuit[9]. The memory can then be read out by attempting to pass a small current through the strip, which will fail if the particular bit has been programmed. This approach has the downside that the broken circuit is visible to microscopy, allowing data to be read out, and it is possible in some cases that the metal will regrow, allowing manipulation of the data[9].

An alternative to eFuse-based memory is *antifuse memory*. An antifuse memory cell is based on a field effect transistor (FET) constructed using normal complementary metal-oxide semiconductor (CMOS) processes[10]. It is programmed by applying a high voltage to the gate terminal of the transistor, causing breakdown of the oxide layer and providing a conducting path. As with eFuse-based memory, antifuse memory, once programmed, cannot be rolled back to non-programmed state. However, unlike eFuse memory, the oxide breakdown is invisible, meaning that confidential data such as secret keys are better suited to storage in antifuse-based memory. Moreover, the oxide layer does not regrow, providing greater integrity than eFuse-based memory.

These advantages, as well as improved yields and lower powere consumption, has resulted in a gradual replacement of eFuse-based memory by antifuse-based memory.

4.3.2 Access control to secure memories

The access control requirements of secure storage depend on its ultimate use. Some areas of memory must remain *confidential*—for example, cryptographic secrets. Both read and write access to these must be restricted to the trusted OS or certain trusted applications (TAs). Conversely, other areas of memory must not be improperly modified, but can be read without any impact on security—for example, untrusted software can safely read the bit that determines whether or not secure boot is enabled, but must not be able to modify it. In ARM terminology, these are referred to as confidential and public fuses, respectively [20, p43].

[8] `https://www.synopsys.com/designware-ip/technical-bulletin/advantages-of-mtv.html`

[9] `https://semiengineering.com/the-benefits-of-antifuse-otp/`

[10] `http://semiengineering.com/the-case-for-antifuse-otp-nvm-for-secure-reliable-socs/`

It may also be desirable to write-protect a region of memory so that it can no longer be written. This is necessary even with OTP memory, as one possible value for each bit is represented as an unprogrammed state, and bits with these values can still be flipped. This can be achieved by writing a lock bit [20, p43], which prevents further writes to the memory in question. In contrast, the TEE provides only those hardware and software components that are necessary to support TAs.

4.3.3 Cryptography on top of secure storage

Where MTP memory cannot be located within an SoC due to its size or manufacturing considerations, cryptographic mechanisms are used to provide secure storage despite the possibility that an attacker might intercept or modify communications between the SoC and the external storage.

One technique is to use an encrypted file system, which encrypts (and potentially authenticates) data before sending it to external storage, allowing decryption only by a key known to the TEE. The data encryption key must still be stored inside the SoC, protected by access control mechanisms that ensure that the storage encryption key is accessible only to the component of the TEE responsible for secure storage. This is commonly achieved by deriving the key from the HUK (Section 4.1.1).

Even if an encrypted file system provides authentication, an attacker may attempt to replay old data from secure storage to the SoC. Roll-back protected memory block (RPMB) is a type of memory that provides an active interface by which an SoC can securely communicate with non-volatile storage, not only providing authentication of data being read and written, but also protecting against replay attacks by making the external storage device take an active part in the protocol, so that the SoC can ensure that the data it receives is the most recent state of a particular external storage device [96]. The mechanism for this protocol involves a shared key, stored in the flash device and in the SoC, which is used to generate and verify message authentication codes (MACs), used to authenticate all read and write operations that access the secured area. Most current flash device types such as eMMC, UFS and NVMe include a fixed-size RPMB partition (configured during device manufacture).

Chapter 5

Hardware-assisted Isolation Mechanisms

In a conventional computing platform, the trusted computing base (TCB)—the components of the system that, if they behave incorrectly, can cause a violation of the system's security guarantees—encompasses the hardware, operating system (OS), middleware services, as well as the code of the application itself. On the other hand, traditional OSs and applications have become so large and complex that the task of adequately securing them has become increasingly difficult. A trusted execution environment (TEE) provides a secure, integrity-protected processing environment where security-critical functionality can be executed, in separation, and isolated from the traditional operating system. To this end, the TEE must provide processing, memory, and storage capabilities which cannot be manipulated by any means from outside the TEE. Generally, a TEE does not host complete user-facing applications. Instead, the security-critical functionality required by an application, such as its cryptographic functionality and key storage, is encapsulated into a separate trusted application (TA) which is run isolated within the TEE. Consequently, a TEE allows the OS, middleware, and the non-security related portions of an application to be excluded from the system's TCB.

Figure 5.1 shows a device as a series of distinct execution environments, each with its own set of features and services. The rich execution environment (REE) ① hosts a traditional, "feature rich" OS ②, services and applications, such as Android, iOS Windows, Linux or OS X. In contrast, the TEE ③ provides only those hardware and software components that are necessary to support TAs ④. Typically a TEE management layer ⑤ provides application programming interfaces (APIs) for TAs within the TEE. TAs typically do not act on their own, but are invoked as needed by their respective client applications (CAs) ⑥ residing in the REE. The REE is subject to access control which prevents it from accessing TEE resources unless it does so via well-defined APIs exposed by the TEE. This access control may be implemented through physically isolating the

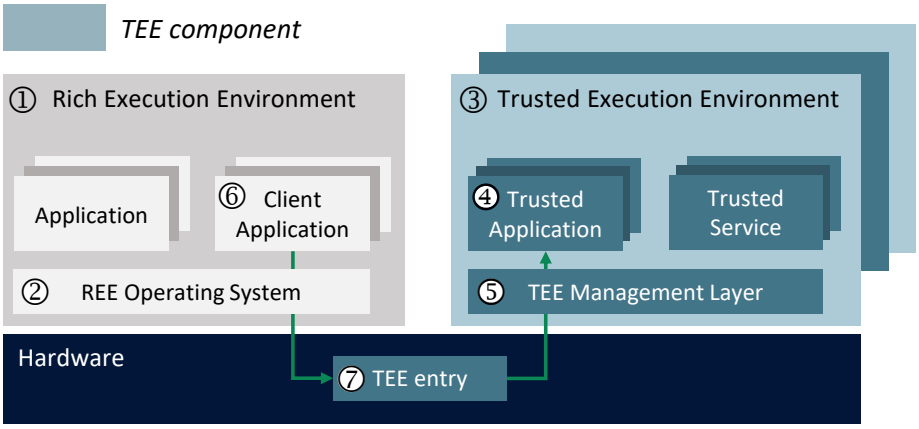
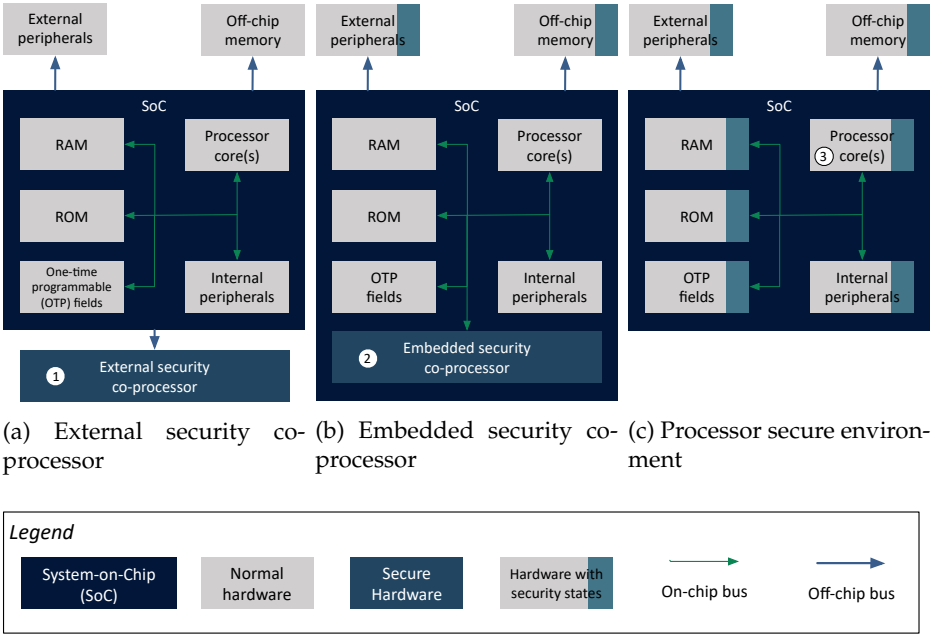


Figure 5.1: Generic system architecture of a TEE-equipped computing device adapted from [119].

TEE (e.g., by applying a discrete co-processor), applying isolation enforced by hardware logic (e.g., a processor secure environment such as ARM TrustZone or Intel Software Guard Extensions (SGX)), cryptographic sealing, or a combination thereof. Transitions between the REE and the TEE are facilitated by hardware mechanisms for TEE entry ⑦.

Figures 5.2a to 5.2c illustrates different options for the architectural realization of TEE hardware. In Figure 5.2a the hardware platform features a discrete, dedicated security co-processor ① outside the physical system on chip (SoC) that contains the central processing unit (CPU)(s). The external security co-processor is fully isolated from the CPUs on the SoC, and contains its own memory and peripherals. This enables minimal coupling between the TEE and the REE, which is beneficial for resisting side-channel attacks which stem from physical characteristics of sharing hardware resources. In some cases, external co-processor designs may feature additional tamper-proofing to hinder physical attacks against the TEE. However, designs featuring external security co-processors are costly, the co-processors are generally less powerful compared to the main on-SoC CPUs, and transferring data between an on-SoC CPU and a security co-processor incurs relatively high overhead. Figure 5.2b shows an alternative security co-processor design, where the discrete co-processor is embedded on the SoC ② together with the on-SoC CPUs. This allows it to more readily share resources, such as memory and peripherals, with the main on-SoC CPUs while still being strongly decoupled and isolated from the REE.

Yet another alternative realization of hardware support for TEEs is the processor secure environment shown in Figure 5.2c. A processor secure environment architecture enables the main on-SoC CPUs ③ to operate in different security states depending on whether it is executing TEE or REE software. In a processor secure environment the on- and off-chip resources, such as memory and peripherals, need to be shared between the processor states corresponding



(a) External security co-processor (b) Embedded security co-processor (c) Processor secure environment

Figure 5.2: Design options for architectural realization of TEE hardware. Adapted from [119].

to the TEE and the REE. Consequently processor secure environment designs typically feature hardware access control logic which decide which areas of memory and / or peripherals may be accessed from which CPU security states. The CPU itself contains hardware logic (e.g., a dedicated instruction or interrupt) to trigger a mode switch from the REE state to the TEE state. Because the TEE and the REE execute on the same processor, a processor secure environment is generally more cost-effective than a dedicated security co-processor, and allows the TEE to benefit from high-performance processing capabilities of the main CPU. On the other hand, because the CPU must be time-shared between the REE and TEE both processing environments cannot be active simultaneously. The resource sharing between the two environments also leads to a tighter coupling between the TEE and the REE where physical properties of the underlying hardware, e.g., timing characteristics and cache utilization, may inadvertently leak information about the TEE operation to the REE. Processor secure environments are also more susceptible to micro-architectural weaknesses, such as transient information leakage through side-channels caused by speculative execution (see Section 9.1.2).

5.1 Split-world architectures

The processor secure environment in Nokia's radio application processor (RAP), Texas Instruments (TI)'s M-Shield and the currently widely deployed ARM TrustZone are all based on the notion of a *split-world* architecture, where hardware-enforced isolation inside the CPU separates execution performed on each core into the "normal" execution environment (REE), and a *single* TEE. In this section, we discuss ARM's TrustZone implementations as representative examples of split-world TEEs.

ARM microprocessors are a family of reduced instruction set computer (RISC)-based computer designs widely used in computing systems which require reduced cost, heat, and power consumption compared to the processor architectures found in personal computers. The ARM family includes three different classes of processors;

Cortex-A series application processors are deployed in mobile devices such as smartphones and tablets, laptop computers, networking equipment and other home and consumer devices.

Cortex-R series real-time processors are deployed in embedded devices with strict real-time, fault tolerance and availability requirements such as wireless baseband processors, mass storage controllers as well as in safety critical automotive, medical and industrial systems.

Cortex-M series of embedded microcontrollerss (MCUs) are geared toward Internet of Things (IoT) systems requiring minimal cost and high energy-efficiency such as sensors, wearables and robotics.

The ARM architecture supports two variants of TrustZone; one for Cortex-A application processors [18], and TrustZone-M for the Cortex-M series of MCUs [19]. Due to security requirements in mobile OSs such as Android [7], TEEs based on TrustZone are today deployed in all Android smartphones based on ARM application processors. TrustZone has also been used as a security foundation in other application domains, including industry, automotive, and aerospace [211].

5.1.1 TrustZone

Figure 5.3 depicts the architecture of TrustZone for Cortex-A application processors. It introduces two protection domains: the REE (referred to in TrustZone-terminology as the *normal world*) and the TEE (the *secure world*). At any given point in time each individual processor core in an ARM SoC operates exclusively in a state corresponding to one of the worlds, alternating between normal and secure execution in a time-sliced fashion. The processor state is determined by the Non-Secure (NS) bit in the individual core's Secure Configuration Register (SCR). Transitions between the non-secure and secure state is mediated by a *secure monitor* ③, which is a security software module that is part of the low-level firmware. The secure monitor is responsible for securely preserving the processor state whenever a world transition occurs.

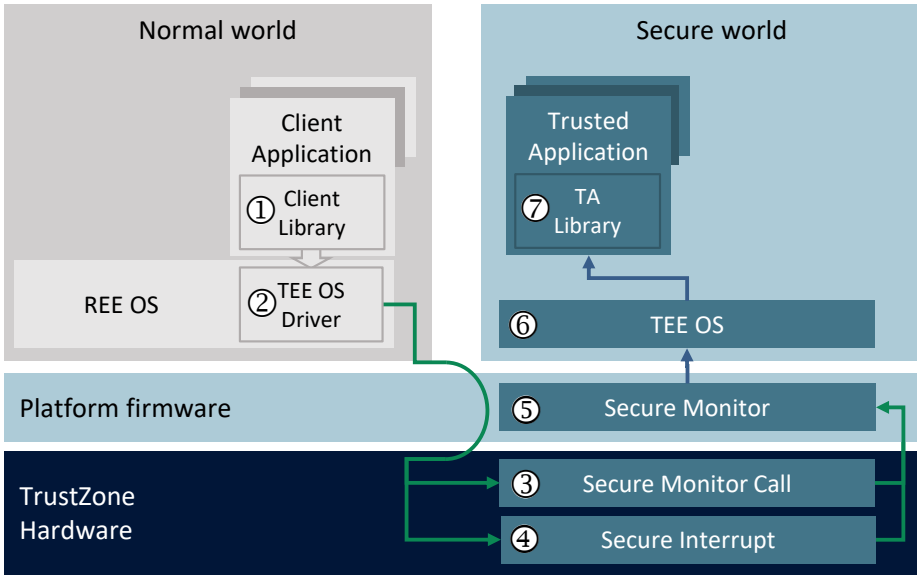


Figure 5.3: TrustZone TEE architecture. ① – ⑦ indicate the sequence of events during a transition from the normal world to the secure world.

The normal and secure worlds in TrustZone are orthogonal to the traditional hierarchical protection domains [229] that enable the separation of privilege between user applications and the OS kernel. ARM processors implement separate privilege levels through exception handling facilities; when encountering an exception, the privilege level can either increase or remain the same, and when returning from handling an exception, it can either decrease or remain the same. For this reason, the privilege levels in the ARM architecture are referred to as exception levels (ELs). The ARMv8-A architecture defines four exception levels, EL0 to EL3, where EL3 is has the highest level of privilege. User applications typically run at EL0, and the OS kernel at EL1. In a virtualization environment, the hypervisor executes in EL2. The secure monitor executes in EL3 (also referred to as *monitor mode* in the ARMv6-A and ARMv7-A revisions of the ARM architecture).

Figure 5.4 illustrates the relationship between exception levels and split-worlds in ARMv8-A. The normal world EL0 and EL1 have corresponding counterparts in the secure world, called *secure EL0* and *secure EL1*, respectively. The secure world lacks EL2, i.e. a hypervisor mode for the secure world in the ARM architecture revisions up to ARMv8.3-A. The ARMv8.4-A architecture revision also adds secure EL2. We discuss virtualization in TrustZone on page 56. Secure EL0 is typically reserved for TAs, whereas the TEE OS kernel runs secure EL1. In contrast to its normal world counterpart, secure EL1 has unconstrained access to the whole physical address space, including memory that belongs to the normal world.

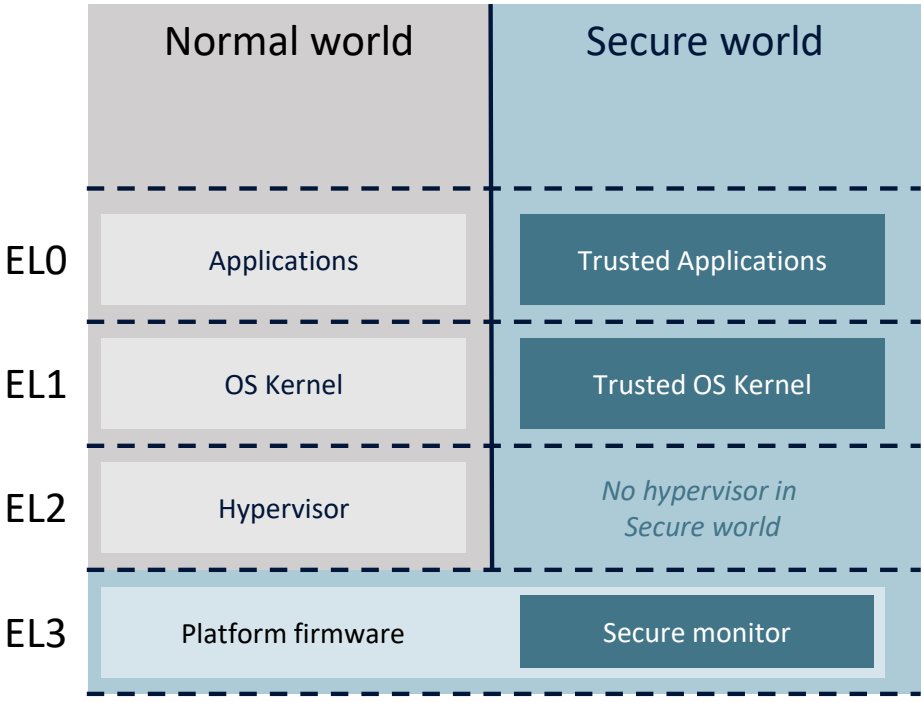


Figure 5.4: ARMv8-A exception levels. Adapted from [25].

Figure 5.3 illustrates the sequence of events in a transition from the normal world to the secure world. First, the client application in the normal world that invokes the *client library* ①, which implements a communication API with the respective TA. The client library issues a system call to a TEE OS driver ② residing in the REE OS. The TEE OS driver implements a marshaling scheme and calling convention understood by the TEE OS and secure monitor. The TEE OS driver can trigger a world transition from the normal world EL1 by executing a Secure Monitor Call (SMC) instruction ③, or via certain hardware exception mechanisms ④. A subset of processor exception lines (namely interrupt request, fast interrupt request, external data abort, and external prefetch abort) can be configured such that the exception is to be handled by software in the secure world. If that is the case, the processor will trigger a world transition when receiving such an exception. Regardless of whether a world transition is triggered by a SMC instruction or via an exception, as a result of the trigger execution will trap into the secure monitor in EL3 ⑤. When in EL3, the processor is always executing in the secure state regardless of the value of the SCR NS bit. The secure monitor will perform a world switch, and a return-from-exception to restart processing in the restored secure world. The TEE OS ⑥ can then check for incoming call parameters per established convention with its REE driver, determine the recipient TA, and invoke the corresponding service

hook in the TA's library API ⑥.

TrustZone in ARMv8-A

In the original TrustZone implementations for ARMv6-A and ARMv7-A, the secure monitor was part of the secure OS, and its exact function specific to a particular secure OS implementation. In ARMv8-A the code running in EL3 is more clearly separated from the TEE OS, allowing key platform management functions to be set apart from the world switch logic and be placed in discrete EL3 firmware. The most important of such services is power management, but EL3 firmware can also include platform specific services and refinements.

As a consequence the software interfaces visible from the firmware to the TEE and the REE OS implementers required interoperability. These are specified by the following three documents from ARM:

SMC Calling Convention (SMCC) [27] specifies how registers are to be used for arguments and return values and how the SMC instruction is to be invoked when requesting a service from the secure world.

Power State Coordination Interface (PSCI) [26] specifies a standard for system and processor power management, including boot, hotplug, idle, and system shutdown and reset.

Software Delegated Exceptions Interface (SDEI) [28] defines a software equivalent to non-maskable interrupts.

In addition to specifications, ARM is backing an open-source effort to provide a reference implementation of the EL3 software stack, the *Trusted Firmware* project[1]. Trusted Firmware implements the SMCC, PSCI and SDEI specifications and provides a reference implementation for boot integrity.

Memory and peripheral isolation

In order to enforce memory isolation between the normal and secure worlds, the memory infrastructure in TrustZone-enabled SoCs have been extended with the addition of the TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA) components. The TZASC is a component on the Advanced eXtensible Interconnect (AXI), which allows specific memory regions in dynamic random access memory (DRAM) to be configured as secure or non-secure. The NS bit that indicates the current execution state of the processor is propagated on the AXI bus with each bus transaction. The TZASC enforces that accesses to secure memory are only allowed if the processor is executing in the secure state. However, memory configured as non-secure is accessible from both the secure, and the non-secure processor states. Programming the TZASC requires secure state access. The TZMA provides similar access control functionality for off-chip read-only memory (ROM) or static RAM (SRAM). To provide memory isolation at the cache-level, the cache line tags

[1] `https://www.trustedfirmware.org/`

of the processor also store the NS bit to indicate whether cache line access is granted in secure or non-secure state.

In a typical ARM system most peripherals are not connected directly to the main AXI bus, but to a simpler, lower power Advanced Peripheral Bus (APB). For backwards compatibility with existing peripheral designs, the APB protocol does not carry the bits related to the TrustZone security state of the bus transactions. Instead, peripheral access control occurs at the AXI-to-APB bridge that translates transactions between the high-speed AXI and the low-power APB. Each AXI-to-APB can mediate accesses for up to 16 peripherals on its local APB based on an input signal fixed by the system designer. The signal is used to determine if the peripheral is configured as secure or non-secure. Similar to the TZASC and the TZMA, the bridge will reject non-secure transactions to secure peripherals. In systems equipped with a TrustZone Protection Controller (TZPC), the security state of peripherals connected to the APB can be dynamically configured at run-time.

The TZASC, TZMA and TZPC are all optional components which may or may not be present on specific SoC designs; older SoC implementations do not incorporate such memory controllers at all; so any form of run-time partitioning of secure and non-secure regions is not possible. Unless the SoC incorporates either dedicated on-SoC memory for the secure world, or a hardwired partition scheme, TrustZone isolation may not be usable to its full extent. However, it is typically less expensive to incorporate a single large memory device and partition it into secure and non-secure regions than it is to provide separate dedicated memories for each world. ARM specifications also leave the number of memory regions and the granularity of configurable regions as implementation defined.

Virtualization in TrustZone

Over the years, as TrustZone deployment has grown especially in consumer devices, a number of TEE OSs (Table A.1) have emerged from different vendors . Despite industry efforts to standardize the APIs between the REE and TEE OSs [113], and the TEE OS and TAs [114], the majority of contemporary TEEs are either based on proprietary APIs, or provide proprietary extensions to standard APIs. Unfortunately those dependencies on proprietary APIs and vendor-specific libraries limit the portability of TAs between TEE OSs from different vendors. The high cost of porting TEE OSs TAs across devices additionally points to that the corresponding CAs also are TEE specific, thus preventing service providers from taking full advantage of standardized TEE functionality As a result, some original equipment manufacturers (OEMs) have opted to host multiple TEE OSs in secure EL1 [52] in order to support a wider range of CAs and their TAs. This, in turn, poses a number of architectural challenges, such as ensuring logical isolation between kernel components belonging to each TEE OS that share secure EL1. Typically this requires the TEE OSs to be modified, i.e., requires collaboration between different TEE OSs vendors.

Another challenge that stems from the original TrustZone architecture is its

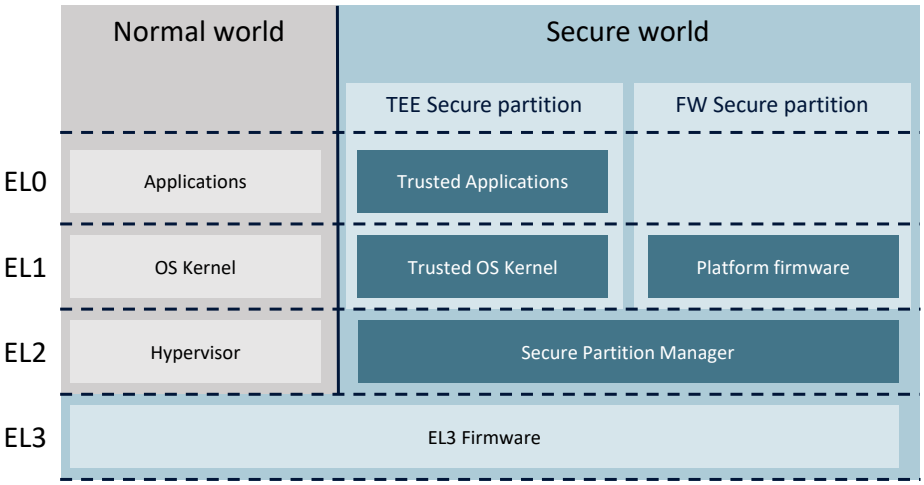


Figure 5.5: ARMv8.4-A exception levels. Adapted from [25].

inability to protect the REE OS from the TEE OS. Software in EL3 and Secure EL1 have *unconstrained* access to the entire physical address space, including memory mapped by the REE OS. This property has been exploited in privilege escalation attacks, where vulnerabilities in the TEE OS are leveraged to compromise the security of the REE OS kernel. Conceptually the original TrustZone architecture suffers from over-privilege of the secure EL1 software, since the split between the normal and secure world is in actuality asymmetric, akin to separation between privilege levels; the hardware only enforces that Non-Secure accesses to memory are isolated from secure memory regions, but not vice versa.

To address this challenge with respect to secure EL1 the ARMv8.4-A revision of the ARM architecture adds virtualization support in the secure world [25]. Concretely this means the addition of a new exception level, *secure EL2*, to the available privilege modes. This, paired with a second stage of address translation controlled by a secure partition manager (SPA) at secure EL2, enables separation among distinct TEE OSs. The SPA is effectively a minimal hypervisor that partitions distinct TEE OSs or platform-specific firmware components into separate *secure partitions* at secure EL1 or secure EL0. Figure 5.5 illustrates the ARMv8.4-A exception levelss. Similar to traditional virtualization, the partitioned "*guest*" TEE OS controls the first stage of address translation from the guest's virtual address (VA) space to an intermediate physical address (IPA) space. The second stage or address translation from IPAs to actual pointer authentications (PAs) is performed by the SPA, thus allowing it to decide which areas of physical memory each secure partition has access to. Secure partitions effectively provide sandboxed environments that enable: 1) distinct TEE OSs at secure EL1 to be isolated from each other, 2) the REE OS to be isolated from TEE OSs, 3) platform firmware to be isolated from TEE OSs.

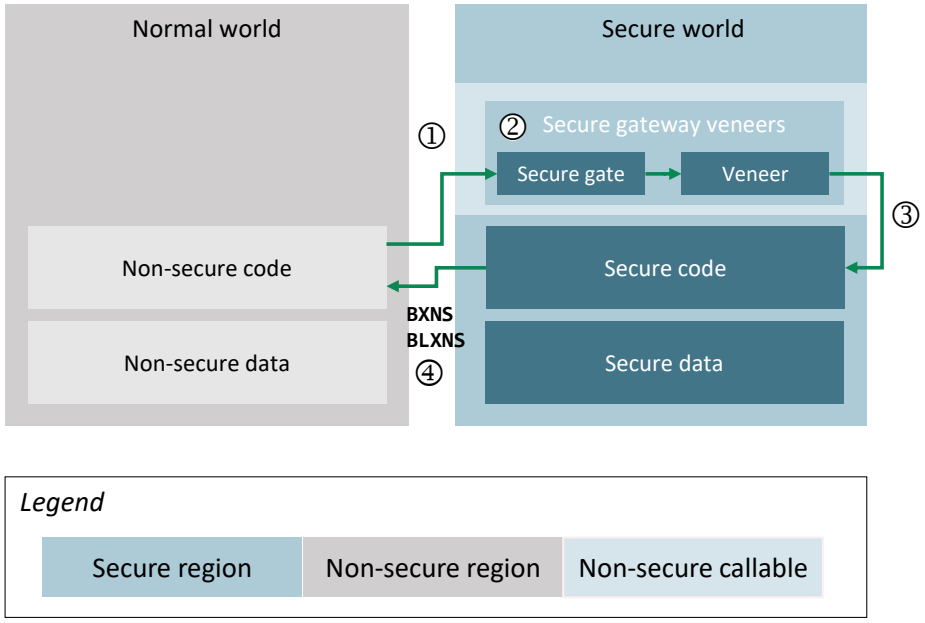


Figure 5.6: TrustZone-M architecture.

5.1.2 TrustZone-M

TrustZone-M [19] is a variant of TrustZone for the ARMv8-M MCU-class Cortex-M processors. In terms of functionality, TrustZone-M replicates the properties TrustZone, but differs significantly in terms of its architectural realization. Like TrustZone, TrustZone-M introduces *secure state* non-privileged and privileged processor modes in addition to the conventional *non-secure state* counterparts.

Cortex-M processors implement two stacks in non-secure state, the *main* stack and the *process* stack. The stack pointer (`sp`) is *banked* between processor modes, i.e., multiple copies of a register exists in distinct register banks. The register bank, and consequently the version of the stack pointer (SP) register in use, is determined by the current processor mode. Register banking allows for rapid context switches when dealing with processor exceptions and privileged operations. Application software on Cortex-M processor executes in *thread mode* where a *stack-pointer select* (`spsel`) register determines whether the application uses the main or process stack. When the processor executes an exception it enters a *handler mode*. In handler mode the processors always use the main stack.

In TrustZone-M `sp` registers used to address the main and process stacks are banked between the non-secure and secure states. Figure 5.7 illustrates the set of SP registers in a ARMv8-M processor equipped with TZ-M. The `MSP_ns` ① and `PSP_ns` ② represent the main and process stack pointer in non-secure state, whereas the `MSP_s` ③ and `PSP_s` ④ represent the corresponding stack pointers

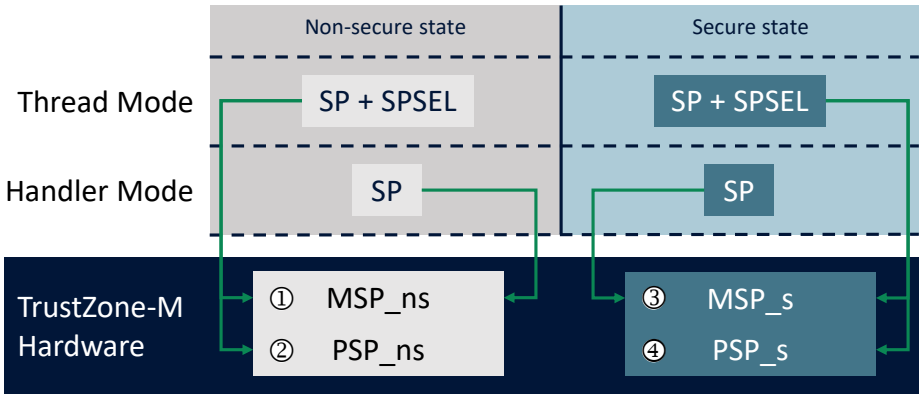


Figure 5.7: Stack pointer management in TrustZone-M.

in the secure state. The remaining general purpose registers are shared (not banked) between the non-secure and secure states.

Also similar to TrustZone, the memory management is extended to allow partitioning the device's physical memory into secure and non-secure regions. In addition, TrustZone-M introduces a new *non-secure callable* (NSC) memory type. Whereas secure memory regions contain the secure program image and data, the NSC memory regions contain *secure gateway veneers*, i.e., branch instructions which point to the actual subroutine code in secure memory. Enforced by hardware, non-secure world code is allowed to call secure world code only via these secure gateway veneers.

Memory regions and their security states are defined either by a Secure Attribution Unit (SAU) or an Implementation Defined Attribution Unit (IDAU). The specifics of the IDAU are left up to the hardware manufacturer, but typically the IDAU enables the definition of up to 255 static memory regions either based on a hardwired memory map, or one configured though one-time programmable (OTP) memory. The optional SAU on the other hand allows the adjustment of the memory layout at run-time by up to 8 additional non-secure or NSC memory regions at run-time. The SAU is configured through a set of memory mapped registers placed in a secure region of memory. In cases where a region mapping conflicts between the IDAU and SAU conflicts, the mapping with the highest security level takes precedence; with secure memory being the highest, followed by NSC memory and non-secure memory. For instance, if an address is defined to be non-secure by the SAU and secure by the IDAU, the address is considered to be secure.

Figure 5.6 illustrates the transition to and from the secure state in TrustZone-M. Whereas transitions to the secure world in TrustZone are initiated through a dedicated hardware exception that traps into the secure monitor at EL3, in TrustZone-M the context switch to secure state occurs automatically as the program flow of non-secure state execution ① targets specific *call gates* in a NSC memory region ②. Each such call gate is identified by a secure gate (SG) instruc-

tion, a virtual opcode that has to precede the memory location in NSC memory to which a inbound branch can jump. The veneer located immediately after the call gate branches into the actual secure subroutine entry point in secure memory ③. The purpose of NSC memory is to prevent non-secure program code to branch into invalid entry points in secure program code (such as into the middle of subroutines, a common exploitation technique for run-time attacks that exploit memory vulnerabilities). Calls from the non-secure state targeting secure memory outside NSC regions, or non-`sg` instructions in the NSC will fail in a *secure fault*, a new type of hardware exception that always traps into the secure state. Note that the TrustZone-M design makes do without an explicit NS bit stored within the processor; the security state is determined by the address of the currently executing instruction. If the address lies within secure memory, the processor is in the secure state, otherwise the processor is in the non-secure state.

Two new instructions are available to secure world software in order to transition into the non-secure state: branch and exchange (to) non-secure (BXNS) and branch with link and exchange (to) non-secure (BLXNS) ④. The BXNS instruction is used by functions executing in the secure state to return to the non-secure state on function return. The BLXNS instructions allows the secure world code to call a function in the normal world code. When transitioning from the secure to the non-secure state, the return address and processor state are pushed onto a secure state main stack in secure memory. The link register of the processor is set to a predefined FNC_RETURN value. To return to the secure state, the non-secure state attempts to branch to FNC_RETURN (as this is the return address found on the stack). Upon detecting FNC_RETURN the processor will instead restore the processor state and the true return address saved on the secure state main stack will be targeted. This elaborate mechanism prevents normal world code from manipulating the destination address returned to in secure state. Since general purpose registers are shared between the secure and the non-secure states, secure world software is additionally responsible for sanitizing any sensitive information held in general purpose registers when transitioning into the non-secure state.

TrustZone-M also extends processor exception handling to facilitate interrupts which are handled by interrupt service routines (ISRs) that can be configured to execute in secure state [22]. When an exception is taken the hardware saves the processor context onto the stack corresponding to the security state which was active before the exception arrived. Exception priorities can be set such that non-secure exceptions are allowed to preempt secure state code. In this case, the processor hardware saves the secure state processor context onto the secure state main stack. The saved context includes a return address, which indicates the address of the next instruction to be executed in the interrupted program. The link register visible to the non-secure exception is set to a predefined EXC_RETURN, which indicates which SP corresponds to the stack frame and what security state the processor was in before the entry occurred.

5.2 Enclave architectures

Split-world architectures rely on the traditional, hierarchical model of protection where software executing at a higher privilege, principally the TEE OS, provides logical separation of TAs from one another. This makes split-world TEEs susceptible to some security pitfalls that are well understood in the context of REE OSs. In general, all software that runs with a higher privilege level than application software is *inherently trusted* and part of the application's TCB, simply because anything with greater privileges than the application will have greater access to the underlying machine than the application has. Consequently, the trustworthiness of a TEE implementation relies on the correctness of both the secure hardware, as well as the ostensibly secure software stack. Furthermore, split-world architectures lack bidirectional isolation: while the TEE is protected from software in the normal world, the opposite is not true. Recall that in TrustZone, secure world software that can operate with physical memory addresses can not only access any secure memory, but has full access to REE memory as well. If an attacker can exploit software vulnerabilities in the TEE OS to breach the separation between TAs, or between the TEE and the REE, they can potentially undermine the trustworthiness of the entire platform. The original rationale for processor secure environments was to exclude the large and complex REE OS from the system's TCB. A smaller TCB generally leaves less room for implementation flaws. As the complexity and size of software inside the TEE increases, so does the likelihood of implementation flaws inside the TEE, which may introduce security weaknesses and vulnerabilities. This, in turn, disincentivizes OEMs and other stakeholders, who rely on the trustworthiness of the TEE from opening the TEE ecosystem for third-party TA developers, simply because more code running in the TEE means a greater risk for software flaws in the system.

Enclave architectures aim to *decouple the notion of privilege from the notion of the trusted computing base*. They do so by providing strong isolation between each (unprivileged) TA using unconditionally trusted hardware, such that the (privileged) OS does not become part of the TCB. This allows each isolated *enclave* to reduce its TCB to just the TA (and a small amount of supporting code) executing within the enclave, and the hardware that provides the underlying execution environment and isolation. The hardware-enforced isolation not only protects each enclave from client applications and the OS, but also from other enclaves on the system, effectively making each enclave its own distinct TEE. This prevents TAs from posing a security risk to one another. In addition, as enclaves do not necessarily possess more privileges than regular userspace processes, the REE kernel also maintains a level of protection against a potentially compromised TA. This reduces the risks involved in allowing software developers to take advantage of TEE functionality in applications and may thus remove some of the disincentives that have prompted OEMs to limit TEE usage to a restricted set of TAs.

To date, enclave architectures have not seen widespread deployment in mobile devices. At the time of writing, the only commercially-deployed enclave-

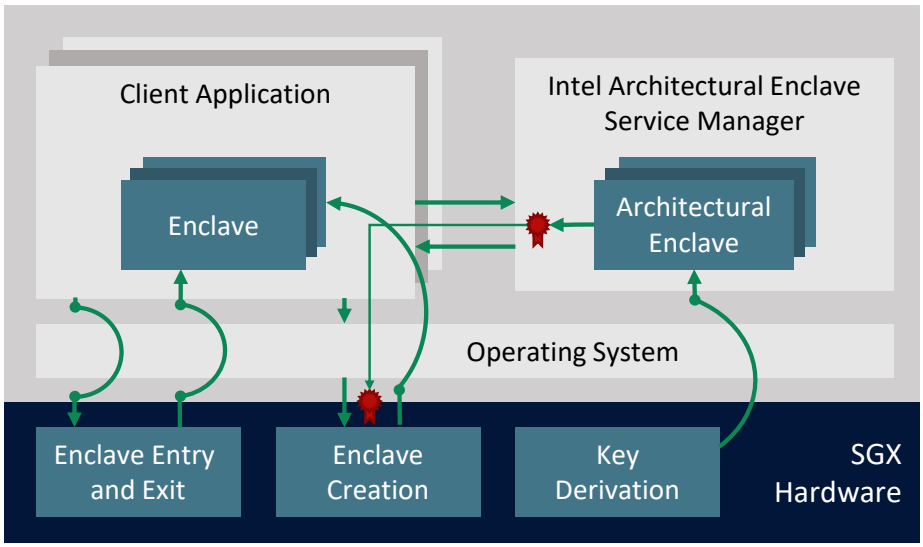


Figure 5.8: Intel SGX architecture. Isolation between TAs is provided by the SGX hardware itself, which mediates all interactions between an enclave and other software components (the application hosting the enclave, other applications, and the OS itself). This protects enclaves from a compromised operating system, in contrast to split-world architectures whose isolation depends on the security of the secure-world OS.

type TEE architecture is Intel's SGX [181, 182] which is principally used to enable trusted remote computation in a public cloud setting. However, enclave-type TEEs have rapidly become the predominant design paradigm in emerging TEE architectures. In this section, we provide and overview of SGX as a representative example of an enclave-type TEE architecture (Section 5.2.1). We later discuss recent academic enclave architectures (Section 5.2.2) and enclaves in the next generation ARM architecture (Section 12.1.10).

5.2.1 Intel SGX

An application that uses SGX can create an enclave, consisting of a region of protected memory containing both code and data, and that can only be read or written by the code stored within the enclave. Code within the enclave can be executed by jumping to a defined entry point of the enclave using an `ecall` instruction.

The initial enclave state is measured by the processor during enclave initialization, yielding an MRENCLAVE value, a hash of the enclave metadata concatenated with the contents of each page of the enclave's memory state. The enclave must also be signed by an authorized developer, whose public-key fingerprint—denoted MRSIGNER—also forms part of the TA identity.

SGX protects against physical and cold-boot attacks [134] by encrypting the contents of enclave memory before it leaves the processor SoC [133]. A random memory-encryption key is generated on every boot, and the encrypted data uses a combination of message authentication codes (MACs) and a Merkle tree to ensure the integrity of data read back from off-chip memory.

Each processor supporting SGX contains embedded secrets accessible only to the SGX hardware. Various SGX instructions will then use these secrets to derive keys that are specific to a particular TA or developer. Encrypting data with these TA-specific keys *seals* the data to a particular processor and TA, such that only the same TA running on the same processor will be able to decrypt the sealed data. SGX provides hardware-backed (non-volatile) monotonic counters that can be used for rollback protection. Together, these are sufficient to provide rollback-protected secure storage.

Rather than implementing the entirety of SGX in hardware, Intel placed some functionality into so-called *architectural enclaves*: enclaves signed by Intel that have access to some additional features of the processor. There are currently two architectural enclaves: the launch enclave and the quoting enclave. The launch enclave is used to validate the public key used to sign an enclave against a whitelist of authorized developers, and to compute a *launch token* for the enclave, which is validated by the `einit` instruction before the enclave is allowed to run. The quoting enclave is used for remote attestation and will be discussed below.

SGX supports two kinds of attestation: local and remote. Local attestation allows a piece of data to be authenticated to an enclave as having come from another enclave on the same machine with a particular identity. Each enclave can obtain a *report key*, derived from a combination of a processor-wide secret and the enclave identity—including whether the enclave is in debug or production mode, the enclave signer, (optionally) the enclave hash. Other enclaves cannot access this report key directly, but one enclave can use the `ereport` instruction to have the hardware build a predefined 'report' structure containing the identity of the current TA and some application-defined data, as well as a MAC over the rest of the report computed with respect to the report key of some other TA (the 'target TA'). An enclave running the target TA can access this key and directly validate the MAC over the report, assuring itself that the report was obtained by an `ereport` instruction in the specified TA.

Since the report key used for local attestation is derived from a hardware-specific key, a report structure cannot be validated on a remote machine. Remote attestation is a two-step process in which an enclave produces a report verifiable by a special *quoting enclave* provided by Intel. The quoting enclave has access to a secret embedded into the processor at manufacturing time that allows it to authenticate itself to Intel. After authenticating itself, it obtains an Enhanced Privacy ID (EPID) [68] anonymous credential that can be used to sign a report, such that the report can be verified as coming from a real quoting enclave, without linking the message to a physical processor. This allows a remote party to trace the provenance of a message to a particular TA.

5.2.2 Emerging Enclave TEEs

A growing body of academic work demonstrates that the isolation boundaries provided by processor secure environments are not always meaningful. Intel SGX, in particular, has received criticism for not defending against *side-channel attacks* (Section 9.1) by design. Side-channel attacks may enable an adversary, who is able to gather statistics from the CPU regarding the execution of an enclave, to deduce private information from within the enclave. Characteristics such as the enclave program's memory access patterns may leak information to the adversary without them directly violating the enclave's isolation. Emerging enclave architectures, such as Sanctum [83], MI6 [63], and Keystone [165] build upon a similar premise as SGX enclaves, isolating the enclave software from the underlying OS using primitives based in hardware (and software), but extend the security properties of the isolation to also cover a threat model that includes side-channel attacks.

The overarching principle behind many of these architectures is an "*immutable commitment of resources*" to an enclave, which ensures that no other process (or even the OS) competes for said resources with the enclave. The resource commitment can enforced by either a *spatial* or *temporal* separation of resources, or a combination of both. Architectures using spatial separation partition caches so that cores executing enclave code receive a dedicated set of cache lines that are spatially separated from cache lines used by untrusted code. With temporal separation, caches are flushed whenever a core transitions from enclave mode to non-enclave mode, preventing the cache state from leaking enclave data to non-enclave code. This prevents side-channel attacks that are based on the cache-timing (Section 9.1).

This challenge is exacerbated by modern processors supporting out-of-order execution, where control-flow mis-speculation can be used by an adversary to tamper with enclave execution (e.g., the *Foreshadow* attack against SGX [70]). This makes it necessary to clear microarchitectural state when transitioning between enclave mode and non-enclave mode. This is particularly challenging given the wide variety of microarchitectural state and resources that may be shared. For example, even though the enclave's cache lines are spatially separated from untrusted code, the cache as a whole can only handle a fixed number of requests at a time. If the shared cache cannot take any more requests, cache misses in the per-core caches will stall the processor, causing timing variations on cache misses which can be detected by untrusted software. By measuring such timing variations, untrusted software can potentially infer characteristics about an enclave's memory access patterns.

In Chapter 12, we will review a number of new TEE architectures that attempt to overcome many of these challenges.

5.3 Security co-processors and multi-TEE architectures

As we saw in Chapter 2, the dominant mobile TEE design, processor secure environments, was chosen because of the desire to keep the bill-of-materials costs low. The drawbacks of current processor secure environments, such as the susceptibility to micro-architectural and other side-channel attacks, have prompted research into sophisticated side-channel resistant enclave architectures, such as Sanctum and MI6 discussed above, but also justify the inclusion of dedicated security co-processors. A separate dedicated co-processor equipped with its own internal memory, caches, and other supporting hardware components such as cryptographic accelerators, are likely to be more resistant to side-channels and microarchitectural attacks originating from untrusted software running on the main processor. In this section we will explore TEE designs that are either based solely on a dedicated security co-processor, or supplement on-board processor secure environments by other types of hardware security modules. We refer to the latter as *multi-TEE system architectures*.

Prominent examples of multi-tee system architectures include Apple's Secure Enclave Processor (SEP) [12] and Google's Titan M chip [188]. The inclusion of a tamper-resistant hardware security module (HSM) *in addition* to a processor secure environment has the benefit of reducing the attack surface, thereby limiting the scope for side channels and invasive physical attacks. It can also greatly simplify some TEE uses cases, such as those requiring secure access to peripherals. These can be accounted for at SoC design time by ensuring that peripherals used as e.g. a *secure path* to a user are hardwired to be accessible only from a discrete HSM.

5.3.1 Intel Security & Management Engines

In 2012 Intel launched it's first smartphone SoC, designed to provide support for the Android operating system on Intel x86 processors. At that time, the existing SoCs developed for the smartphone and tablet market by companies such as TI, Nvidia, Qualcomm and Samsung were all based around ARM-based processors. Intel's SoC platform, codenamed *Medfield* [156], is based around an Intel Atom processor combined with all the necessary input/output (I/O) interfaces to complete a smartphone design. The Medfield SoC incorporates a dedicated "*security engine*", a discrete co-processor that provides secure storage via eMMC NAND flash and OTP fuses, cryptographic acceleration and random number generator (RNG) as well as security timers and counters [142]. The security engine hosts a programmable TEE environment, called "*Chaabi*"[2]that supports the necessary TAs required by Android, i.e. keymaster, digital rights management (DRM) etc.

[2]`https://android.googlesource.com/platform/hardware/bsp/intel/+/4cfb5f6f9d07112035bacdc3959a66feab938de1`

The notion of an independent, isolated security co-processor was not new to Intel. Since 2007, Intel's "mainstream" chipsets, e.g., the *Core* family of microprocessors, have included an embedded computing environment commonly referred to as Management Engine (ME) [222]. ME was first introduced for implementing Intel's Active Management Technology (AMT) in its mainstream chipsets. AMT is an advanced (remote) system management feature originally introduced in Intel's gigabit Ethernet controllers. It was the first application to be moved to ME. Security applications soon followed, the first being integrated Trusted Platform Module (TPM) functionality (marketed under the Intel Platform Trust Technology (PTT) moniker). In contrast to a conventional, discrete TPM chip, ME integrated TPM is realized as TA within the ME. Similar to processor secure environments, the principal motivation for integrating TPM functionality as part of the ME firmware is a reduced bill-of-materials compared to a discrete TPM chip. SGX enclaves also leverage ME by utilizing the ME's cryptographic acceleration, monotonic counters, and secure timer.

The security engine for mobile chipsets and ME for the mainstream chipsets have gone through a number of iterations throughout different generations of Intel architectures. In current Intel terminology (as of 2017) the security engine and ME are referred to as the Converged Security and Manageability Engine (CSME), with different firmware sets for mainstream, server, and mobile chipsets respectively.

On Intel's modern server chipsets (starting with 2017 "*Lewisburg*" chipset) ME is complemented by another co-processor, the Innovation Engine (IE) [177]. Whereas ME is a closed system that solely runs Intel's platform software and applications, IE is intended as an isolated execution environment open to *system designers* (e.g., OEMs) to run customized firmware in. The combination of ME and IE therefore also constitutes a multi-TEE architecture, where different discrete co-processors are utilized by different stakeholders.

5.3.2 Apple Secure Enclave Processor

SEP [12, 173, 174] is a security co-processor in Apple iPhone, iPad, and MacBook devices introduced in 2018 and later. SEP was introduced for iOS devices using the A7 SoC with the release of the iPhone 5S, the first phone manufactured by Apple based on the 64-bit ARMv8-A architecture. In addition to SEP, it featured a TrustZone-enabled ARM application processor cluster. TrustZone in iOS is used for REE kernel integrity protection, while SEP is responsible for key management and cryptographic operations for application data protection, including file-level encryption as well as fingerprint and face data from the *Touch ID* and *Face ID* biometric sensors. SEP prevents the main application processors from gaining direct access to sensitive biometric data and cryptographic keys. Only a small amount of dedicated SRAM is available to the SEP, and similarly to TrustZone, SEP must therefore also utilize external DRAM shared with the main application processor. To do so securely, DRAM allocated to SEP is protected by 1) TZASC protected memory regions (TZ0 for SEP, and TZ1 for TrustZone), and 2) memory encryption, based on an ephemeral mem-

ory protection key generated at boot time using SEP's dedicated RNG. The memory protection key is created by the Secure Enclave Boot ROM, which acts as the root-of-trust (RoT) for SEP code integrity. SEP itself runs a Secure Enclave OS based on a customized version of the L4 microkernel. Later iterations of SEP employ authenticated encryption to ensure the integrity of the enclave memory (A8 SoC and newer) and run-time replay protection based on nonces stored in on-chip SRAM (A11, and S4 SoCs and newer). The replay protection is extended to data stored by SEP on the REE file system in A12 and S4 SoCs, by including an anti-replay counter stored in non-volatile memory in a discrete Integrated Circuit (IC). Communication with the application processor occurs via an interrupt-driven mailbox and shared memory in unprotected dynamic random access memory. On MacBook devices based on the Intel architecture, SEP is part of the T2 Security Chip [11], which apart from SEP also integrates a system management, a controller, an image signal processor, an audio controller, and an SSD controller. The T2 chip also implements a hardware trusted path from the monitor lid hinge to the device's microphone, which ensures that the microphone is disabled whenever the lid is closed.

5.3.3 Google Titan M

Google's Titan M chip [188] is an evolution of the Titan HSM [228] deployed in Google's cloud platform. However, in contrast to Titan, Titan M is geared towards mobile devices. Specifically Titan M is designed to meet the the security needs of Google's Pixel line of Android devices, starting with the Pixel 3 launched in 2018. Similar to Apple's SEP, Titan M is a separate, physically isolated chip from the main ARM processor. Unlike SEP Titan M does not share memory or persistent storage with the main processor. However, Titan M is based on a Cortex-M3 MCU with only 64 Kbytes of RAM. Consequently it is limited to providing low-level primitives for use by software running on the more powerful TrustZone processor secure environment. The Titan M chip stores rollback counters used by Android's verified boot functionality, and key material for the Android Strongbox keystore, a hardened version of the previous processor secure environment-based keystore. In addition, Titan M is hardwired to the Pixel's side buttons, providing a trusted, unforgeable path to button presses by users. This functionality is exposed to third-party apps that rely on user interaction to confirm a transaction, giving a higher level of assurance that the user, and not malware, has confirmed the transaction.

While Apple's SEP and Google's Titan-M are custom hardware solutions designed to meet the needs of specific operating systems, a logical next step for the development of multi-TEE architectures is the addition of such functionality to general purpose SoCs. Qualcomm secure processing unit (SPU) [215] is an *on-SoC* embedded secure element available in Qualcomm hardware starting from from the Snapdragon 855 SoC. Integration on-glssoc means that the SPU is fabricated on the same silicon die as the main processor. This has two benefits compared to a discrete security chip such as the Titan M, while still being isolated from the main processor: 1) reduced bill of materials cost for OEMs,

and 2) a small advantage in terms of power efficiency. The SPU can replace Titan M on Android devices as the security element protecting cryptographic keys placed in Android's Strongbox keystore, but in addition can also function as an integrated SIM (iSIM).

Chapter 6

Cryptographic hardware

As we have seen previously, many platform security building blocks like trusted boot (Section 4.2.1), remote attestation (Section 2.4), and secure storage (Section 4.3) rely on the use of cryptography. Hardware platforms provide cryptographic functionality as a service to both the platform itself and applications, in the form of secure storage for key material, acceleration of cryptographic operations, and hardware random number generation.

Hardware support for cryptography is necessary for a number of reasons. The first is performance. Cryptography implemented in software may require a substantial share of the central processing unit's (CPU) capabilities, and consume significantly more power than an implementation using specialized hardware. For example, an ARM Cortex-A9 core at 800 MHz (often used in embedded devices such as screens and cameras) can encrypt only around 20 Mbytes/s using the AES-256 algorithm if the whole CPU is dedicated to this task[1], even though modern mobile networks are capable of transferring hundreds of megabytes per second. Moreover, many types of data such as video streams require significant additional processing after decryption. The second reason is that software-based implementations are less secure: cryptographic keys reside in normal memory and so are vulnerable to runtime attacks (discussed in Chapter 8), or attacks against the random access memory (RAM) itself (discussed in Chapter 10). As the operating system (OS) provides isolation between processes, any compromise of it can lead to leakage of key material. Also, the software implementations of random number generators (RNGs) and storage mechanisms are often vulnerable to differential power analysis and side channel attacks [44], discussed in Chapter 10. These issues can be avoided by implementing cryptographic primitives atop purpose-built hardware.

Cryptographic libraries normally provide plugin mechanisms so that applications can take advantage of cryptographic accelerators and hardware-protected data stores without hardware-specific changes.

[1] https://www.design-reuse.com/articles/36013/cryptography-hardware-platform-for-socs.html

6.1 Cryptographic modules

Hardware-based cryptographic modules can improve performance and security. As symmetric cryptographic operations such as bulk data encryption and decryption widely outnumber asymmetric operations such as key establishment and digital signing, symmetric cryptography is the most common target of hardware acceleration, whereas both symmetric and asymmetric operations benefit from hardware-backed isolation mechanisms.

Hardware assistance need not necessarily occur at the level of an entire cryptographic primitive, such as encipherment of a block or the signing of a chunk of data. Vendors often choose to accelerate only the most computationally intensive sub-components of an algorithm, such as multi-precision modular exponentiation. This saves valuable space in the processor design and cuts down on design costs. In practice, this means that the overall cryptographic primitive may remain software driven, and some pre-processing such as data origin verification or the initialization of key material takes place in normal software.

The extent to which a cryptographic module's configuration is exposed to the software using it will influence the overall level of security that the module can provide. At one end of the spectrum, traditional hardware security modules (HSMs) expose little of their internal state. For example, in the cryptographic module for Intel vPro Platform Security Chipset, keys for encryption and message authentication are directly loaded from internal secure storage to secure registers, and are not visible to any software or firmware. Operations such as Advanced Encryption Standard (AES), Secure Hash Algorithm 256 (SHA-256), Secure Hash Algorithm 1 (SHA-1) and large-number arithmetic are performed by the Converged Security and Manageability Engine (CSME) hardware block [36]. This prevents compromised software from obtaining raw key material.

There are a number of ways in which system on chip (SoC) designers implement hardware acceleration:

Special instructions for cryptographic operations: In this case, the CPU provides instructions that perform all or part of a cryptographic operation. This is convenient from a backwards-compatibility point of view, as there are no shared resources that need to be managed by the operating system.

For example, ARMV8-A (Cortex A53/A57) and newer include a single-instruction multiple-data (SIMD) and floating-point unit that provides such instructions [21].

Similarly, Intel processors since 2010 include the Advanced Encryption Standard New Instructions (AES-NI) [265] extension to perform AES algorithm rounds and key expansion [21].

Cryptographic modules packaged inside the main processor: Qualcomm's secure processing unit (SPU) hosts a cryptographic subsystem (the cryptographic management unit (CMU)) providing hardware blocks for accelerating cryptographic operations. Supported operations include hashes, block ciphers, and asymmetric cryptographic operations for Rivest-Shamir-Adleman (RSA) and

elliptic-curve cryptography (ECC) [216]as well as a hardware key store [215]. Similarly, ARM provides the CryptoCell-300 and CryptoCell-700 families of cryptographic accelerators as off-the-shelf blocks that can be integrated into processor designs [231]. The CryptoCell-700 family integrates with ARM TrustZone to provide a root-of-trust (RoT), cryptographic accelerator, and a hardware key store. This tight integration is useful for access control, e.g. to provide for encryption keys that can be used only by secure-world software [31]. ARM CryptoCell-312P is targeted more for low-power and small-area designs and comes with security enhancements to protect against side channel attacks [232], for example nRF610 system in package[2].

Cryptographic modules packaged separately into a security co-processor: This approach is used by the Offload Cryptography Subsystem (OCS) module in the CSME subsystem [36, 135], and Google's Titan M. Titan M has built-in hardware accelerators that can be initialized with keys provided by firmware or with device-specific and hardware-bound keys generated by the Titan M's Key Manager module [188]. Similarly Intel's Management Engine (ME) contains accelerators, as well as a hardware key store that can load keys directly from storage to internal registers for AES or HMAC operations [36]. Apple's Secure Enclave Processor (SEP) provides cryptographic acceleration, but is also integrated with the direct memory access (DMA) path between storage and main memory, so that the SEP can perform highly efficient disk encryption [11] without exposing the encryption keys to the main CPU.

In CPU designs where cryptographic accelerators are shared, the corresponding hardware and software stacks must be engineered to avoid concurrency issues and race conditions. These may lead to leakage of keys or cryptographically sensitive data if software designs do not provide proper separation under all circumstances [24].

The Federal Information Processing Standard 140-2 (FIPS 140-2) standard provides guidelines for development of hardware cryptographic modules. Modules can be certified at one of several qualitative levels ranging from one (the least secure) to four (the most secure) [101]. The fourth level requires co-processors with tamper-resistance and the ability to self-erase if attacks are noticed. For example, Qualcomms's Snapdragon Cryptographic module and ARM's Cryptocell 712 and 713 are certifiable to FIPS 140-2 [233] [3].

6.2 Random number generators

In addition to cryptographic operations themselves, random number generation is essential for cryptography. However, secure and reliable sources of randomness are difficult to obtain in digital systems, which are highly deterministic. It is therefore common for a processor to include a hardware RNG;

[2] https://community.arm.com/developer/ip-products/processors/trustzone-for-armv8-m/b/blog/posts/nordic-announce-first-cortex-m33-based-chip-with-trustzone

[3] `https://www.qualcomm.com/news/onq/2014/11/07/cryptographic-module-snapdragon-805-fips-140-2-certified`

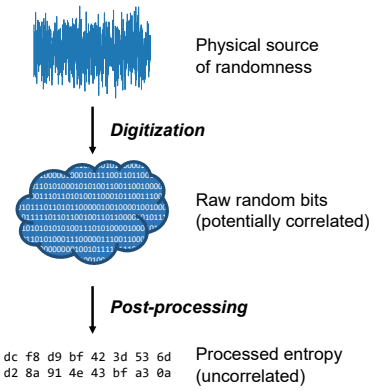


Figure 6.1: A physical noise source of noise is digitized and post-processed to provide a source of high-entropy random values.

the use of a hardware implementation allows its designers access to stochastic physical phenomena such as electronic noise.

Even if the system has the ability to monitor a noisy analog source like thermal noise [255], atmospheric noise [136, 86], hard disk seek timing [255], mouse movement [274] or radioactive decay [267], the noise in such processes is often strongly autocorrelated. The quality of random number generation is critical to cryptographic security; e.g. the Digital Signature Algorithm (DSA) and the Elliptic Curve Digital Signature Algorithm (ECDSA) signature algorithms are particularly sensitive to the quality of the random number generation, allowing full recovery of the private key when even a few bits of randomness are known [197, 196]. Therefore, random number generation typically happens in two phases: 1) generation, and 2) refinement, as illustrated in Figure 6.1.

In the second phase, raw random data is postprocessed to remove bit-to-bit correlations and other imperfections, producing a high-quality random bitstream. The post-processing step is also important to protect against fault-injection attacks against the physical source of randomness [20].

The random output is then used for key generation, for generating unique identifiers or secrets that will be provisioned into the device during manufacture, nonces, initial counter values, and challenges for challenge-response authentication [195], among other cryptographic uses.

Testing and validating random number generator implementations for their security properties is a difficult mathematical endeavor. Several international and national standards have been written to support such evaluations, e.g., NIST SP800-90b [255], ISO 18031 [144], and BSI AIS-31 [69].

In some cases, certification against these standards results in the publication of information on the design of cryptographic systems. An example of this in a mobile chipset is is the Qualcomm SPU [216], which takes advantage of electronic noise that causes electronic oscillators to drift apart from each other, allowing their values to be combined to produce a stream of random output.

This output is post-processed using a pseudorandom number generator [45] to produce the final output for use by software.

Chapter 7

Run-time protection mechanisms

On mobile platforms applications are often strongly isolated within their own sandboxes (Chapter 3) and written in memory safe languages such as Kotlin for Android [126] or Swift for iOS devices [14]. Nonetheless, both the platform itself and various application components are written in "low-level" languages such as C/C++, that make them vulnerable to memory safety errors. To address this, the research literature has since the late 90s been rife with various approaches to address security vulnerabilities stemming from memory errors. A typical approach is to retrofit programs written in C/C++ with mechanisms that prevent memory errors or their exploitation. Such schemes range from approaches that only prevent some memory errors, such as stack canaries that detect out-of-bounds memory writes on the stack [84], to mitigation techniques that prevent specific exploitation techniques, e.g. control-flow integrity (CFI) solutions that prevent code-reuse attacks [1]. A notable early success story is the widespread deployment of W^X memory policies that prevent an attacker from directly injecting new code into the executable memory space of a program [185, 206]. Unfortunately, the deployment of other comprehensive run-time protection mechanisms has been limited due to large performance overheads or compatibility issues. Consequently, many software-only protection schemes—for instance, sanitizers such as Address Sanitizer [234]—are only used during development, in order to detect (and correct) memory safety violations during testing. This is poised to change with the recent influx of hardware-assisted memory safety primitives geared towards efficient realization of protection schemes in consumer hardware. One example is the ARM pointer authentication that is deployed in all recent Apple devices [13]). These should be efficient enough to be deployed in production. We will next look at various existing and upcoming hardware primitives for run-time memory protection.

7.1 Intel 64 architecture

The Intel 64 architecture has introduced two major run-time protection extensions in recent years: Intel Memory-Protection Extension (MPX) and Control-flow Enforcement Technology (CET). First, we will discuss Intel MPX, which was introduced in the Skylake architecture in 2015 and is one of the first run-time memory protection extensions in contemporary CPUs. MPX prevents memory safety issues by providing new instructions for run-time detection of buffer overflows and other spatial memory errors in legacy C/C++ programs. CET is narrower in scope, and specifically addresses code reuse attacks (Section 8.3) by providing CFI [1]. Although static (compile-time) CFI policies are not exact and can be circumvented [72], they are well understood and are applicable to legacy software with minimal impact on performance and compatibility. While CET is only now being deployed in the Tiger Lake architecture [162], the specification was released already in 2016. CET is supported by both the Windows and GNU/Linux operating systems (OSs). We will next explore these two architectures in more detail.

7.1.1 Intel Memory-Protection Extension

Intel MPX provides hardware-assisted instrumentation to detect spatial memory errors. Buffer overflows, and spatial memory errors in general, are a prevalent security issue in programs written in C/C++ [250]. The research literature has proposed several approaches for mitigating these issues by introducing run-time bounds checking for pointers. The approaches either leverage *fat pointers*, that include information about the bounds of pointers in the pointers themselves, or store bounds information separately from the pointers. While fat pointer schemes can be efficient, they typically require changes to the pointer layout and size in order to accommodate the included bounds information. The changed layout can cause compatibility issues, for instance with pointer arithmetic, or break assumptions on the memory layout of structures. MPX takes the latter option and stores bounds information separately and thus avoids compatibility issues inherent to fat pointers.

Prior schemes implemented in software—e.g. Baggy Bounds Checking [3] and SoftBound [193]—suffer from performance issues due to slow bounds lookup. MPX mitigates the overhead of the lookup by providing dedicated instructions for storing and retrieving pointer bounds. The bounds access is accelerated by a table walk performed in microcode, similar to the virtual memory page table walk used by the processor's memory management facilities. In addition, MPX provides new registers for storing bounds information and instructions for checking pointers against the bounds information in those registers.

The store and load instructions for bounds information use a two-level addressing scheme (Figure 7.1) based on a pointer's storage address (i.e. the address at which the pointer itself is stored, not the pointed-to address). The first lookup table is the Bound Directory (BD) which is used to lookup a Bound Table (BT) that is then the index needed to fetch the correct entry. On 64-bit

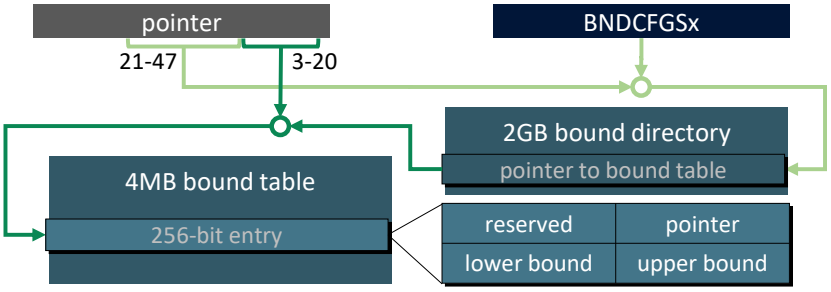


Figure 7.1: MPX bounds metadata addressing in 64-bit mode.

systems the BD is 2GB in size, whereas each individual BT is 4MB. Reserving physical memory for these is unwieldy, and so, MPX supports on-demand memory mapping by the OS. Whenever a memory page of the BD or a BT is first accessed, it will cause a specific fault, that allows the OS kernel to step in, map the necessary memory, and then resume process execution. While this approach is feasible in user space, it significantly hinders the use of MPX in environments that cannot handle page faults, e.g. the kernel itself [220].

Similarly to the other hardware extensions discussed in this chapter, MPX is used by the compiler instrumenting a program with explicit instructions for manipulating and checking bounds. Compiler support for MPX was introduced in GCC v5.0, but was later dropped in v9.1 [105]. In addition to bounds store, load, and check instructions, the compiler must also allocate registers for storing the bounds for pointers in registers. The calling convention is then modified to propagate the bounds of pointer function arguments by using the new bounds registers. Many allocation-based schemes cannot *narrow* bounds, that is, create bounds to a smaller portion of an allocation. This can lead to security issues by allowing within-allocation memory corruption [108]. In contrast, MPX is pointer based and does allow bounds narrowing. However, aggressive bounds narrowing is incompatible with common programming patters, such as iterators and retrieving a pointer to the enclosing structure from a pointer to an element of the structure. Consequently, narrowing was sparsely deployed in the GCC MPX instrumentation.

As indicated by the dropped GCC support, Intel has deprecated MPX [251]. The Linux kernel followed suit with the removal of MPX support in v5.6 [163]. This is likely due to multiple factors, including compatibility and performance issues [201]. As reported by [201], MPX is outperformed by the software-only AddressSanitizer (although direct comparison is problematic in that, in contrast to MPX, AddressSanitizer is allocation-based and does not support narrowing but does detect use-after-free errors). The metadata storage scheme of MPX can, depending on the program, incur substantial memory overhead; increasing memory consumption by 100% on average. In order to avoid this overhead, proposed uses of MPX in research literature typically do not leverage the built-in bounds storage but only use the MPX registers and check in-

structions. For instance, SGXBounds [158] provides bounds checking inside Software Guard Extensions (SGX) enclaves (Section 5.2.1). The smaller address space of an enclave allows the bounds to be encoded in the higher bits of pointers without increasing pointer size. Other projects use MPX for coarse-grained enforcement, e.g. to provide in-process memory isolation [155], to accelerate CFI [189], or to realize execute-only memory [212].

7.1.2 Intel Control-flow Enforcement Technology

Intel CET provides hardware-assisted control-flow integrity [141] for mitigating control-flow hijacking attacks. CET consists of two mechanisms: a hardware-protected shadow stack that securely stores return addresses, and Indirect Branch Tracking (IBT) that constrains the control-flow of forward branches. In contrast to MPX, CET—and CFI in general—does not attempt to address the underlying memory error, instead it restricts how control data (such as corrupted function pointers) can be used at run-time. First, the shadow stack prevents an attacker from altering function returns by modifying the unprotected return address on the program stack. Second, IBT provides coarse grained CFI to ensure that function calls or jumps always branch to valid addresses. This ensures that function calls always target valid function entry points, thus limiting the utility of attacks that alter code pointers. These defenses significantly limit an adversary's capability to exploit memory errors to achieve their goal.

The CET shadow stack works by adding a new attribute to memory pages that identifies shadow stack pages and enables the memory management hardware to mediate accesses to those pages. Any explicit access of shadow stack pages is prohibited. Instead, call and return instructions implicitly write and read it via a new shadow stack pointer (SSP) register. When CET is enabled, a function call not only pushes the return address to the normal stack, but also to the shadow stack pointed-to by the SSP. Subsequently, as a return is executed, the hardware loads the return address from both stacks, and in case the addresses differ, issues a control protection exception (which indicates that the return address has been corrupted).

IBT is implemented as a state machine with two states (Figure 7.2). It starts in the `IDLE` state until it encounters a jump or call instruction and sets the state to `WAIT_FOR_ENDBRANCH`. To use IBT, the program must have been instrumented such that any valid jump or call targets—such as function entry points—are marked with the `endbranch` end branch instruction. When an end branch instruction is executed in the wait state, the state is reset back to idle. Any other instruction executed in the wait state causes a control protection exception that indicates an invalid control-flow transfer. While not included in the simplified state machine, IBT also allows the feature to be disabled or suppressed, for instance to support legacy code.

The GCC compiler introduced support for CET shadow stacks in version 8 [122, Section 3.18.46]. At the time of writing, Microsoft is introducing CET in the Windows operating system via hardware-enforced stack protection [167]. Similarly, many Linux distributions already include CET support, and patches

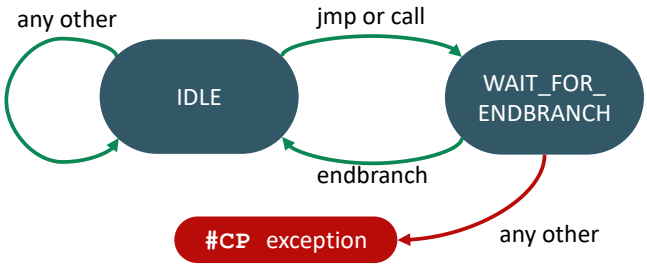


Figure 7.2: Intel CET IBT state machine.

are being reviewed for the mainline Linux kernel [268]. Hardware support for CET was deployed in the Intel Tiger Lake architecture in October 2020 [162] and AMD Zen 3 since November 2021 [2].

7.2 ARMv8-A architecture

Similar to Intel MPX and CET, ARM provides Memory Tagging Extension (MTE) that targets memory safety itself, and pointer authentication and Branch Target Identification (BTI) that addresses CFI. The ARMv8.3-A architecture first introduced pointer authentication that allows "signing" and verifying pointers, in practice enforcing CFI by protecting the integrity of function pointers and return addresses when stored in memory. The BTI extension introduced in ARMv8.5-A, provides a CFI mechanism similar to the IBT feature of Intel CET. Meanwhile, ARMv8.5 MTE uses memory tagging to allow efficient detection of both temporal and spatial memory errors. While this hardware support is still not widely deployed, these extensions have gained traction on iOS and recent Apple system on chips (SoCs) supporting pointer authentication and Google working on supporting these extensions on the Android platform.

7.2.1 ARMv8.3 pointer authentication

As discussed in Section 7.1.2, pointers are an attractive target in run-time attacks because they can allow exploits to hijack the normal operation of a vulnerable program, including enabling the execution of arbitrary malicious code. Memory-safety mechanisms, such as Intel MPX, try to prevent memory errors from happening, and mitigation techniques such as Intel CET mitigate the exploitation of corrupted pointers. Another approach is to try and determine whether a pointer has been corrupted *before* it is used. The ARM pointer authentication extension, available in ARMv8.3-A SoCs and later, adopts this approach by providing hardware primitives for *signing* and *verifying* pointers [214, 21]. This allows protected systems to sign pointers before storing them in memory, where they might be corrupted by an attacker, and then verify the pointers before they are subsequently used. The idea of cryptographically pro-

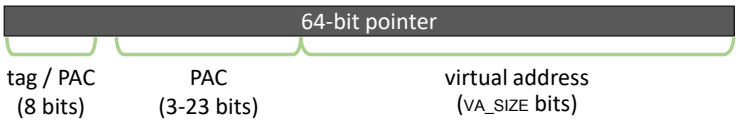


Figure 7.3: The PAC is stored in the sign-extension bits of a pointer.

tecting pointers is not new in research, in particular different software-based encoding or masking approaches have been proposed for protecting function return addresses [213]. However, with hardware support, pointer authentication can typically offer better performance and security. A more recent precursor is Cryptographic CFI, which realizes a scheme similar to pointer authentication using AES acceleration on Intel systems to generate message authentication codes (MACs) [176].[1]

Pointer authentication "signs" pointers using a tweakable MAC to generate a short MAC—called the pointer authentication code (PAC)—and embeds it in the sign-extension bits of the pointer itself (Figure 7.3). Depending on system configuration, specifically the size of the virtual address (VA) space and use of pointer tagging, this allows a MAC ranging from 3 to 31 bits in size. A typical 64-bit ARM Linux configuration would use a 16-bit PAC. The PAC is calculated based on the pointer's virtual address, a hardware-protected 128-bit key, and an application defined 64-bit *modifier*. The choice of MAC algorithm is implementation defined, with the QARMA algorithm by Qualcomm being one possibility. The algorithm is realized in hardware and can be designed for dedicated pointer authentication use. For instance, the QARMA algorithm is optimized for hardware implementation and a pointer authentication operation realized with it is expected to complete in four cycles on a 1.2GHz core [37]. This efficiency will likely allow the security mechanism to be used outside testing environments, and indeed, the Apple A-series SoCs starting from A12 support pointer authentication [13].

Pointer authentication use

The pointer authentication extension provides a set of key registers to hold the pointer authentication keys, and a set of instructions for signing and verifying pointers. A common set of key registers are shared between all exception levels (ELs), but accesses to key registers can be configured to trap into a higher exception level to facilitate key management across different protection domains. Consequently, the firmware must configure pointer authentication to allow desired use. For instance, in order to allow the OS kernel to manage keys for user-space processes, pointer authentication must be configured to allow access to key registers in EL1. In order to use pointer authentication, an application is then instrumented with pointer authentication instructions to sign and verify

[1] While [176] mention the possibility of embedding the MAC into the pointer itself, their actual implementation elects to use a separate hash-table to store the MACs. This allows longer MACs without modifying pointer layout.

pointers where needed. This approach is quite flexible and allows different protection schemes to be implemented by altering how the 64-bit modifier is assigned to pointers (e.g. modifiers based on pointer type [166] or pointer storage address [87]).

Pointer authentication use in user space has been supported by the Linux kernel since v5.0 [223], with initial support for protecting the kernel itself introduced in v5.7 [175]. The GCC 9.2 and Clang 9.0 (and later) compilers support the use of pointer authentication to sign function return addresses [104, 77]. The Apple ecosystem, starting with Xcode 10.1, has also started to roll out support for using pointer authentication to protect function pointers in iOS applications [13]. However, adoption, in particular outside relatively closed ecosystems such as iOS, is challenging because pointer signing breaks application binary interface (ABI) compatibility if applied beyond the local scope by instrumenting pointers other than return addresses. Nonetheless, support for ABI versioning and other mechanisms are currently planned, to accommodate more widespread use of pointer authentication in heterogeneous systems [56].

ARMv8.6 pointer authentication extensions

On successful PAC verification, the PAC bits are cleared from the pointer, allowing it to be used. This is necessary as the memory management unit (MMU) expects the PAC bits to be sign-extended, and will issue a fault when translating a malformed pointer. Pointer authentication verification failures leverage this to detect corrupted pointers without immediately raising a fault. Instead, on failure, the hardware first clears the PAC bits and only then flips a predetermined bit to ensure that the pointer cannot be used. Signing works similarly, pointer authentication always first writes the correct PAC into the pointer, and only then checks whether the input pointer was malformed, and if so, flips a single bit to ensure the resulting signed pointer fails verification. Unfortunately this allows an attacker to generate arbitrary signed pointers when a verification is followed by a signing without intermediate use of the pointer [40].

To address this vulnerability, ARMv8.6-A introduces two enhancements to pointer authentication: PAuth2 and FPAC [245]. First, the PAuth2 feature modifies the scheme for embedding PAC by replacing the overwriting and clearing of the PAC with a exclusive-or operation. With PAuth2, a failed verification will thus cause an incorrect PAC to be generated, which is then exclusive-ORed into the PAC bits of the corrupted pointers, resulting in a garbled PAC. Subsequent signing first generates the correct PAC for the corrupted pointer but then exclusive-ORs it into the now garbled PAC bits, thus ensuring that the pointer cannot be used. Second, the FPAC feature allows the verification instructions to be configured such that they immediately fault on verification failure. This not only prevents the attack described above, but also helps debugging by ensuring that authentication failures cause an immediate effect.

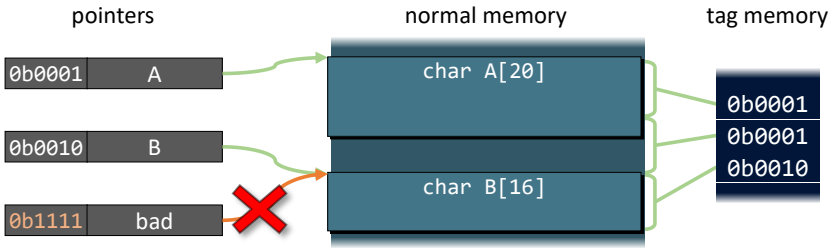


Figure 7.4: MTE provides a lock-and-key mechanism checks that pointers accessing tagged memory have an address tag that matches the allocation tag of each accessed 16-byte memory granule.

7.2.2 ARMv8.5 Memory Tagging Extension

Memory tagging has been around for various purposes since the 1970s, e.g. in the Burroughs machine that used memory tags to implement typed memory [159]. Recently, ARMv8.5 MTE introduces memory tagging for run-time memory safety by realizing a "lock and key" type scheme for memory accesses [21]. Using MTE, memory can be marked with a 4-bit *allocation tag* such that any *checked access* of the memory must be performed with an address marked with the corresponding *address tag* (Figure 7.4). Similar to pointer authentication, tags are embedded into pointers. Consequently, the use of MTE decreases the possible PAC size by one byte. MTE allows probabilistic detection and prevention of both spatial and temporal memory errors [235]. By assigning a random tag to each memory allocation an out-of-bounds or use-after-free memory dereference has only a $16^{-1} \approx 6\%$ probability of succeeding. This can allow efficient error detection both during development and in deployed applications [23].

Similar to the pointer authentication extensions, MTE does not automatically protect programs. Instead, a protected program must be instrumented with added MTE instructions to tag memory and pointers where needed. MTE provides new instructions for manipulating address tags in pointers, including an instruction for generating random address tags from a subset of the 16 possible tags. Other instructions can then be used to write allocation tags into memory based on the tag in given input pointers. Stack variables, for instance, can thus be conveniently tagged by generating a first random tag and then incrementing the tag for each subsequent variable stored in a stack-frame slot.

Allocation tags must be explicitly set and always apply to a naturally aligned 16-byte *granule*. The alignment and size restrictions can necessitate changes to program memory layout, e.g. that variables stored in tagged memory always occupy at least 16 bytes irrespective of actual variable size. A naive approach could thus result in significant memory overhead; however, in practice MTE can be combined with static analysis to decrease unnecessary coloring and thus decrease the overhead [244]. In practice, MTE also requires support from memory allocators to tag dynamic allocations, with support being made available in the Scudo allocator of LLVM [170]. At present, MTE is supported by

Clang compiler through hardware-assisted AddressSanitizer, HWASAN [81]. It is supported on the Android platform for both testing through HWASAN [8] and as a mitigation in the Scudo memory allocator [9].

7.2.3 ARMv8.5 Branch Target Identification

The BTI extension for ARMv8.5-A realizes a coarse-grained forward-edge CFI scheme [21] similar to Intel CET's IBT (Section 7.1.2). The scheme can be selectively enforced by marking specific memory pages with the `GP` flag to indicate that they are *guarded pages*. It adds the new `bti` instruction that can be used to mark valid targets for indirect branch instructions. Whenever an indirect branch instruction with link is executed (e.g. a call using a function pointer), the processor state register `PSTATE.BTYPE` flag will be set to indicate the type of branch instruction. Similarly, the state register is also set whenever jump or other branch instruction is executed, but only within a guarded page. When executing an instruction within a guarded page, the hardware faults unless the executed instruction is a valid BTI target for the `PSTATE.BTYPE`.

BTI target types are encoded only in two bits, allowing only coarse grained CFI. Only two separate CFI labels are used: `c` for function calls, `j` for branch without link, or alternatively `jc` for either. For comparison, Intel's IBT only supports one class of valid branch targets. The hardware is designed to cooperate with pointer authentication by recognizing pointer signing instruction as valid targets, i.e. if pointer authentication is used to sign the return address immediately on function entry, then no additional BTI instruction is needed. Although BTI is relatively simple, it should effectively mitigate many attacks by restricting the use of corrupted code pointers.

7.3 CHERI

The CHERI instruction set architecture (ISA) extension for architectural capabilities [263, 261] have recently gained significant traction in the research community. Capability-based addressing replaces regular raw pointers with *capabilities* that are integrity protected and include pointer bounds and additional access permissions. In contrast to pointer-based schemes such as Intel MPX, capabilities are unforgeable, and can only be created in specific circumstances using dedicated instructions. Capability-based addressing was already used in systems such as the IBM System/38 [137]. However, these systems typically employed slow lookup tables for book-keeping. In contrast, CHERI, and the M-Machine before it [74], encode all relevant information in the capability itself and thereby forgo costly metadata lookup.

A CHERI capability is double the architecture pointer size, e.g. 128 bits on 64-bit systems. By exploiting redundancy in the information, the added bits are sufficient to encode a type, permissions and both bounds (Figure 7.5). In addition, the hardware tracks a separate 1-bit capability tag for each 128-bit chunk of memory to mark capabilities, and separate them from other data stored in

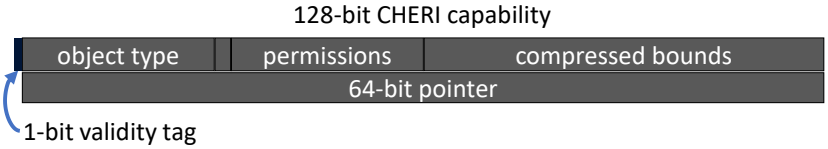


Figure 7.5: The memory layout of a CHERI capability. In addition to the 128-bit capability stored in memory, CHERI also maintains an unforgeable 1-bit validity tag to mark valid capabilities.

memory. Most importantly, the capability tag cannot be set though normal memory manipulation. Instead capability *provenance* is guaranteed by only allowing their creation and modification through specific instructions. Furthermore, capability derivation is guaranteed to be *monotonically decreasing in power*, i.e. the permissions of a derived capability must always be a subset of the source capability's permissions and bounds. There are exceptions, such as the initial capability that must be created at boot to allow access to system resources. The CHERI type system also provides a mechanism for controlled non-monotonicity, which can be used to realize CFI and compartmentalization.

Although capabilities do not directly address temporal memory errors, the unforgeability and architectural nature of CHERI facilities efficient implementation of delayed re-allocation schemes. Specifically, because capabilities can be unambiguously detected in memory, it is possible to accurately and efficiently sweep the memory to verify the absence of references to freed memory and invalidate any such references by invalidating the capability. Such a sweep can be further optimized by minor architectural additions for detecting tag presence efficiently, e.g. new instructions for reading tags without reading the associated memory, and new page table attribute to identify pages that contain capabilities [100].

The first published CHERI implementation was for the MIPS ISA [263]. However, CHERI was intentionally designed as an extension that can be applied to different architectures. It is also designed as a hybrid system, i.e. it can be deployed in systems that mix raw pointers with capabilities. This is possible by providing a default data capability that is used when addressing via raw pointers. A single process can thus be partially instrumented without necessarily compromising the security of CHERI protected code sections. At present, a CHERI implementation is also available for the 32- and 64-bit RISC-V architectures [261]. Furthermore, the ARM Morello program is set to introduce CHERI support for the ARMv8-A 32- and 64-bit architectures [30].

Part III

What can go wrong?

Chapter 8

Software-level attacks

In Part II, we discussed the design and implementation of hardware platform security in contemporary mobile devices. In Part III, we will explore the potential vulnerabilities in, and attacks against hardware platform security in mobile devices.

The traditional design of platform security architectures were based on the assumption that hardware platform security is inviolable. But recent developments have questioned this implicit assumption. Although hardware platform security is based on hardware-based building blocks such as TrustZone extensions, their trusted computing base (TCB) still consists of software and firmware components to a significant extent. Like any low-level software, such components are vulnerable to software attacks such as those that exploit memory vulnerabilities. To date, these are, by far, the most frequent source of vulnerabilities found in hardware platform security architectures. In Chapter 7, we already encountered some hardware security mechanisms intended to defend against such software attacks. In this chapter, we will discuss different types of software attacks against trusted execution environments (TEEs).

Although software vulnerabilities feature prominently in attacks, the underlying hardware mechanisms may also come under attack. Modern hardware architectures are complex. Their features can interact in unexpected ways leading to new types of vulnerabilities. In Chapter 9, we will take a look at different types of CPU-level attacks: these are attacks exploiting microarchitectural hardware features such as caching, pipeline speculation, and out-of-order execution. In this chapter, our focus will be on software attacks, i.e. attacks that mostly leverage the imperfect interaction or protections between software and memory. Such exploits can be mounted by a remote adversary and allow for potentially widely applicable attacks.

In some settings, a local adversary may have physical access to the device. In Chapter 10, we will consider these types of attacks. Examples include side-channel attacks that require physical access (such as monitoring power consumption or electromagnetic emissions from the device) as well as fault injection attacks, i.e. physically inducing faults in hardware components.

8.1 Memory vulnerabilities

Modern mobile operating systems employ stringent isolation between applications to limit the reach of compromised or misbehaving programs and users. However, software is also becoming increasingly complex and thus more prone to software defects that might constitute vulnerabilities. Such vulnerabilities can be broadly categorized into "high-level" design errors and "low-level" memory vulnerabilities. The former materialize when the abstract program logic or design is faulty regardless of specific implementations. The latter are typically implementation-specific and materialize from the specific way a program is realized as machine code for specific hardware. In this book we focus on the latter; these are also the vulnerabilities that the defenses presented in Chapter 7 address. We call such vulnerabilities *memory vulnerabilities* or *memory errors*.

These errors can further be divided into *spatial* and *temporal* memory errors [250]. An example of the former category are the well-known buffer overflows. More generally, this category includes any memory error that violates the memory boundaries of an object. The latter are errors that violate the lifetime of objects. An example of this category are *use-after-free* errors where a pointer to an object is used after the object has been freed. Use-after-free errors become exploitable when the underlying memory is reallocated such that the pointer to the freed object points to, and allows modification of, another object's memory that has been allocated at the same memory address. A memory vulnerability alone could be used to blindly launch a denial-of-service attack by corrupting the internal state of the program and causing it to misbehave. However, the attacker might have different goals, ranging from information leaks—e.g. as in Heartbleed [249]—to system takeover.

8.2 Code-injection attacks

One of the early examples of exploits utilizing memory-vulnerabilities is the Morris worm from 1988 [203, 243]. It exploited multiple vulnerabilities in Unix utilities, including a spatial memory error in the `fingerd` daemon. The specific error was missing input validation that could be exploited to cause a stack buffer overflow. The worm exploits this to inject executable code onto the program stack and overwrite the function return address so that the program executes the injected code upon return form the `main` function. Such injected code is called *shellcode* because they often invoke a system call to launch an attacker-controlled shell (typically `/bin/sh`).

To construct the payload, an attacker needs some knowledge of the target memory layout. In the case of the Morris worm, the payload must be offset correctly such that the injected return address is written into the expected memory location, in this case, the frame record of the target function. Moreover, the attacker must also know the shellcode's memory address because that is used as the injected function return address. At the time, this could

be achieved by simply analyzing common binaries that would, at run-time, be mapped to identical addresses in memory across different systems. Commonly deployed randomization defenses today, such as address-space layout randomization (ASLR), may force the attacker to find another information leak or resort to active probing in order to figure out the correct addresses and offset. While W^X policies (Chapter 7) prevent most code-injection attacks, they are still possible, for instance when attacking just-in-time compilers that must write new executable code into memory [219]. Nonetheless, the prevalence of defenses against code injection has motivated new advanced attack techniques that can bypass W^X.

8.3 Code-reuse attacks

Code-reuse attacks utilize pre-existing program code to achieve desired attack behavior. The first example of such attacks was the *return-into-libc* attack demonstrated in 1997 [209]. As the name suggests, the attack works by corrupting the return address of a function such that it jumps into the beginning of a `libc` function. By injecting the desired function arguments onto the stack, this technique effectively calls the chosen function with attacker-controlled arguments. The `libc` library is a useful target for code-reuse attacks because it is often mapped into a known memory range and provides functions with general-purpose functionality. Although return-into-libc is limited to existing functions, it allows the injection of multiple bogus stack frames to induce a sequence of attacker-controlled function calls. This technique has later been expanded and refined into return-oriented programming (ROP) attacks.

Return-oriented programming: Introduced in 2003 [236], ROP is an advanced code-reuse technique that allows more flexible attacks without being constrained to only chaining calls to pre-existing functions. It is similar to return-into-libc, but instead of functions it uses small program fragments—called *gadgets*—consisting of only one or a few instructions. Each gadget ends in an indirect branch instruction, typically a return, that is used to load the address of the next attacker-chosen gadget. This allows the gadgets to be chained such that they effectively form an alternate instruction set that the attacker can use to realize desired functionality. In fact, ROP attacks exploiting gadgets in `libc` have been shown to provide a set of gadgets that is expressive enough to support arbitrary computation; often referred to as a *Turing-complete* set of instructions.

As with code-injection, randomization defenses can mitigate ROP attacks by randomizing the run-time addresses of gadgets. However, techniques for probing the memory layout of a victim process have been shown to be sufficient to realize *just-in-time ROP* that can circumvent even run-time (re)randomization by making gadget discovery part of the attack chain [242]. Return-address protection, such as Intel Control-flow Enforcement Technology (CET) or shadow stacks (Section 7.1.2), can prevent return address corruption. But the same technique can be used by corrupting code pointers, such as plain function pointers or virtual function tables [59]. While forward-edge control-flow integrity (CFI)

schemes also protect function calls and jumps, they are often coarse-grained and allow multiple targets for a given call site. Nonetheless, defenses like Intel CET (Section 7.1.2) and ARM pointer authentication (Section 7.2.1) that can be used to enforce integrity of control flows, effectively mitigate code-reuse attacks by restricting the availability of gadgets. The prevalence of such defenses has prompted research into advanced *data-only attacks*.

8.4 Data-only attacks

Data-only attacks do not alter control-flow data such as function pointers or return addresses. Regardless, they can still be leveraged to mount denial-of-service attacks by corrupting the data of a program or to leak sensitive data. A prominent contemporary data-leakage attack is the Heartbleed vulnerability that utilized an over-read to leak sensitive program data [249]. Heartbleed exploited TLS/DTLS heartbeat messages that cause the server to respond by echoing a message back to the client. The vulnerable implementation relied on a user supplied length-field that allowed an attacker to supply an incorrect length to cause the response to include uninitialized memory possibly still containing sensitive data. This could be exploited to read up to 64 kilobytes of unrelated server memory. Even without control over the over-read memory, Heartbleed has been shown to enable leakage of confidential information such as private keys.

Although data-only attacks do not directly manipulate control data, they can influence the behavior of a program by altering the decision-making data used by a program, for instance, by overwriting the current user ID or file paths. Moreover, recent work on data-oriented programming (DOP) has shown that data-only attacks are powerful enough to achieve Turing-complete exploitation similar to ROP [138]. This type of attack has begun to appear in the wild, as demonstrated by the FORCEDENTRY exploit [49] against the iMessage messaging application. Because no control-flow data is directly manipulated, the individual control flows of the program remain within the control-flow graph (CFG) and the attack remains unhindered by defenses such as CFI. Demonstrated DOP attacks still depend on data-pointer manipulation using memory vulnerabilities and can therefore be prevented using pointer-protection schemes. Another insight is that current DOP attacks alter program behavior in detectable ways, which can be leveraged for protection or attack detection [80].

8.5 Attacks on TEEs

As with all software, implementations of trusted applications (TAs) or the TEE operating system (OS) can exhibit software defects and architectural weaknesses that may lead to vulnerabilities. Recall that the rationale for TEE isolation is to shield trusted code from malicious behavior by the rich execution

environment (REE) or its applications. By isolating only the security-critical functionality, the software implementation of the TEE can be kept relatively small—compared to REE kernels that can consist of millions of lines of code—allowing its correctness to be more easily verified. However, the codebases for modern TEE implementations, especially in mobile devices using processor secure environments such as ARM TrustZone, have grown in size to the point were confidence in the correctness of code in the TCB is weakened [75]. This is exemplified by a long history of critical implementation bugs in TEE software, including:

- *Input validation errors* [221, 237, 48, 51, 53, 54], where an attacker exploits undefined behavior in a TA, for example by a buffer overflow or use-after-free.
- *Confused-deputy vulnerabilities* [172], where a client application (CA) abuses the valid behavior of a TA in order to violate some security policy.
- *Insecure software installation* [46, 52], enabling loading of arbitrary software, or the rollback of already patched TAs or the TEE OS version to a previous, vulnerable version.

There are two reasons why TEEs are prime targets for software attacks: Firstly, they expose a large attacks surface through REE interfaces [75]. Hardening REE system services and kernel interfaces can help limit the exposure, but ultimately TAs rely on their own input validation and that of the TEE OS for protection against malicious inputs. Secondly, TEE software often lacks defenses against run-time attacks [46, 237], which are commonplace for REE OSs. This may be due to resource constraints within the TEE [199], or due to inherent hardware limitations. For instance, in the first generation of Intel SGX-capable processors, memory permissions cannot be changed after enclave initialization [57]. This makes it difficult to combine countermeasures based on address space randomization with policies for data execution prevention. This limitation was lifted in the second generation Software Guard Extensions (SGX) hardware [182].

8.6 Attacks on hardware-assisted memory defenses

Run-time memory-protection mechanisms are a promising step forward but are not a panacea. Prior research has also shown many existing of defenses are either incomplete or have theoretical limitations. Allocation based memory-protection, such as ARM Memory Tagging Extension (MTE), can be worked around by corrupting other memory objects within the same allocation [109]. Coarse-grained CFI mechanisms, such as Intel CET, prevent many attacks, but still allow *control-flow bending* attacks that alters intended functionality while adhering to the CFI policy [72]. Practical deployment and performance considerations further limit defense effectiveness, even for hardware-assisted mech-

anisms; for instance, Intel CET offers only coarse-grained CFI and ARM MTE only supports 4-bit tags.

Phasing in new hardware to a disparate set of customers and devices is also a slow process. Meanwhile, new features prompt research into attacks that circumvent them. One recent example is the Project Zero attack that circumvent ARM pointer authentication [40]. The attack exploits the specifics of how authentication failures are handled in order to realize a gadget for signing arbitrary pointers. It can be partially prevented in software; ARM has since updated the pointer authentication specification in ARM-A v8.6. Nonetheless, this exemplifies a challenge of applying hardware-based mitigation: a new hardware fix does not help already deployed hardware, and the generally slow hardware update cycle means old, vulnerable, devices remain in use..

Theoretical limitations coupled with implementation-specific flaws indicate that memory vulnerabilities will continue to be a problem for the foreseeable future. Nonetheless, the increasingly sophisticated attacks indicate that runtime protection mechanisms (Chapter 7) indeed make exploitation of modern systems more challenging. Controlled application ecosystems and use of hardware-assisted isolation mechanisms (Chapter 5) also serve to restrict the vulnerabilities that do occur. As such, although we might not be rid of software-level attacks any time soon, there is an increasing range of tools and mechanisms that allow the mitigation and management of associated risks.

Chapter 9

CPU-level attacks

In Chapter 8, we discussed attacks against vulnerable software. In this chapter we will discuss attacks that take advantage of vulnerable hardware.

We consider two kinds of attack: side-channel attacks, where an attacker violates the confidentiality of a security boundary by examining the effect of the victim code on shared resources, and fault-injection attacks, where an attacker violates the integrity a victim behind a hardware-enforced security boundary.

9.1 Side-channel attacks

Side channel attacks take advantage of resources that are shared between multiple processes, allowing an attacker to make inferences about the state of other processes beyond what is allowed by their defined interfaces. A side-channel attack has two main stages: first, the data to be taken must find its way into the shared resource. This may occur accidentally, or with some encouragement by the attacker. Secondly, the attacker must have a way to read out the contents of the shared resource in question. There may be more than one way to read data from a given resource, providing increased bandwidth and accuracy by taking advantage of features of particular conditions in which a side channel may occur.

9.1.1 Cache-timing side channels

Cache-timing side channels allow an attacker to infer the memory access patterns of a program; if this pattern is dependent on secret data, then it can lead to a vulnerability. Cache-timing side channels have been a major concern for implementations of the Advanced Encryption Standard (AES) encryption algorithm, which was designed to be efficiently computed using table lookups [55]. The reliance on table lookups allows cache-timing attacks to determine the indices that are looked up, which can then be used to infer the key.

Central processing unit (CPU) caches are generally *set-associative* [204, p. 497]. Physical memory is divided into *blocks*, and the cache into *sets* of cache lines that can each store a single block. The lowest-order bits of each physical address provide an offset into a block; the next-lowest bits are used to associate an address with a cache set. Then, when a block is loaded into the cache, the cache selects a cache line from its associated set, and uses it to store a copy of the block, along with a tag used to disambiguate it from other blocks assigned to the same set.

Processes with separate address spaces can still share cache lines. As a result, *memory access times* can be used as a readout process for this side channel. Loads from more-recently accessed memory address will complete faster, as they can be loaded from caches nearer to the CPU.

The *Prime+Probe* attack [202] is based on the principle that, since blocks from different processes map to the same cache sets, an attacker can detect whether another process—the *victim*—accesses a given address by loading similarly-aligned regions of their own memory, thereby filling the cache set corresponding to the target address with their own data. Then, after the victim process has run, the attacker reads back the same memory again. If the victim has not accessed the target address, then this will be fast; otherwise, the attacker's data will have been evicted from the cache, and the load will be slow. This allows the attacker to make inferences about the memory access pattern of the victim. Prime+Probe is not perfectly reliable, as the blocks used to probe a given cache set can be evicted by loading *any* block associated with same cache set, resulting in false-positives.

The *Flush+Reload* attack [266] works in the more restricted case where the attacker shares a memory page with the victim. Then they can use the more precise Flush+Reload cache-timing attack, which tests for the presence of a specific block. In order to do this, the attacker and victim processes must access the same physical address. This can occur if the same shared library is mapped into both their memory spaces, or if pages are deduplicated by a hypervisor. In this case the attacker can load from the same physical address as the victim, and if this completes quickly, then this specific block is in the cache and must definitely have been loaded. This is in contrast to Prime+Probe, which can detect memory access only at the level of cache-sets rather than blocks.

While the above attacks were initially demonstrated on the desktop-centric x86 architecture, later work has shown that the attacks can also be conducted on a ARM architectures and mobile operating systems [168, 273]. Such attacks have also been shown to be capable of leaking secrets from ARM TrustZone, even when launched from a non-privileged Android application [271].

9.1.2 Transient execution vulnerabilities

Even where a program's memory access patterns are not secret-dependent, modern processors use *speculative execution* [204, p. 434], allowing execution to continue while the CPU waits for inputs to arrive from uncompleted instructions or from distant banks of memory. When the CPU guesses wrongly

what the inputs will be, the speculative execution will be erroneous, and when the inputs eventually arrive, the CPU will roll back the speculative execution and start again with the correct inputs. Nevertheless, the speculative execution has *microarchitectural* side effects that are not rolled back. In particular, speculatively-executed memory accesses affect the state of the cache, resulting in a cache-timing side channel between the speculative and non-speculative executions. If an attacker can trick the processor into speculatively executing secret-dependent memory accesses, then they can extract secrets using one of the previous attacks.

The Meltdown attack [169] takes advantage of the fact that kernel memory was mapped into the process address space, and protected by memory access control mechanisms. Intel CPUs would speculatively execute past a load from kernel memory, allowing the instructions immediately following the load to leak the secret data using a cache side-channel before the access control bits are obtained and an exception is raised. A variation on Meltdown, known as Foreshadow [71], uses the same underlying vulnerability to also read data from Software Guard Extensions (SGX) enclaves.

Spectre [149] vulnerabilities rely on branch misprediction to create secret-dependent side channels in software whose memory access pattern is normally independent of all secret values.

One approach is to use speculative execution to bypass a bounds check. An attacker may be able to obtain secret data through an apparently non-vulnerable data-dependent memory access by means of a buffer over-read, using speculative execution to ignore any bounds checks that prevent the over-read from being executed non-speculatively.

Another method is for the attacker to poison the indirect branch predictor by repeatedly jumping between appropriately aligned addresses in the attacker's address space. This will make the CPU speculatively jump to an arbitrary attacker-controlled address when it reaches the targeted branch instruction. These speculative mispredictions can be chained together to realize any computation desired by the attacker, similarly to a return-oriented programming (ROP) attack, described in Section 8.3.

The Load Value Injection attack [256] combines aspects of the Meltdown and Spectre attacks. Whereas in a Meltdown attack, the attacker speculatively loads data from the victim's memory space, Load Value Injection reverses this, tricking the victim into speculatively loading data belonging to the attacker. This is then used to trick the victim into revealing secret data through a side channel.

9.1.3 Mitigations

Cache-timing side channels are avoided by eliminating secret-dependent memory accesses. This means that not only must secret-dependent access data always follow the same pattern, but the control flow must also follow the same pattern so that instruction fetches do not reveal secret data. Some languages provide tools that can assist. For example, the HACL* formally-verified

cryptographic library [276] defines a *secure integer* interface that prevents non-constant-time operations and usage as an array index.

Trusted execution environments (TEEs) can also isolate the cache usage of their trusted applications (TAs) from other applications. For example, the Sanctum [83] TEE architecture flushes the per-core caches whenever a core transition from enclave mode to non-enclave mode takes place. Shared caches are partitioned so that TAs have dedicated cache sets that are not shared with untrusted code, preventing the cache-timing attacks presented above.

More 'forensic' approaches have been suggested that attempt to detect the microarchitectural behaviors characteristic of cache-timing attacks. CloudRadar [272] uses CPU performance counters to identify the execution of cryptographic code, and then attempts to correlate these events with anomalously large numbers of cache hits or misses that are indicative of a cache-timing attack against cryptographic code.

9.2 Fault-injection attacks

Whereas a side-channel attack allows an attacker to infer state that they should not be able to access through normal interfaces, a fault-injection attack allows an attacker to modify state that they should be unable to modify through normal interfaces.

9.2.1 Rowhammer

Rowhammer [148] takes advantage of the electromagnetic coupling between dynamic random access memory (DRAM) cells in high-density memory chips. Each row of cells has a *wordline* that is used to activate it when a memory access occurs. Repeatedly accessing two different rows causes the wordline to be toggled high and low repeatedly, causing the charge in some adjacent cells to leak more quickly than normal, to the point of flipping between DRAM refreshes. Rowhammer can be used across the TrustZone normal-/secure-world boundary, and this has proven to be exploitable in practice [73].

9.2.2 CLKSCREW

The CLKSCREW [252] attack induces faults by using the the control that the rich execution environment (REE) has over power-management functionality to introduce glitches through the clock and power inputs. The attacker takes advantage of the fact that different cores have independent voltage and frequency domains, allowing software running on one core to introduce glitches in another process without causing itself to fail. The effectiveness of this attack was demonstrated by using it to extract AES keys from a TA, and to load a self-signed TA.

9.2.3 Mitigations

Fault-injection attacks can be mitigated in two main ways: first, by preventing the fault from being injected, and secondly, by making faults difficult or impossible to exploit when they do occur.

Where the shared resource is a well-defined entity, fault-injection attacks can be prevented with appropriate access control. For example, one defence against Rowhammer attacks is to physically separate security domains from one another, as in [65] or [61], which place a certain number of guard rows between regions of memory used by software in each security domain, making attacks less likely to succeed.

Error correction and other forms of redundancy can also be used to mitigate fault-injection attacks, since they force an attacker to correctly modify several pieces of state simultaneously. One example is the use of error correcting memory, which can detect and correct the random memory errors caused by Rowhammer attacks. Another is to perform computations more than once and compare the results, as suggested by [252]. An alternative to redundancy is randomization, which aims to limit the probability that an attack will be successful. For example, by inserting random loops of no-op instructions into the program, the attacker can modify the timing such that a CLKSCREW attack is unlikely to be correctly targeted, and therefore preventing reliable exploitation.

Chapter 10

Physical attacks

In Chapter 8 and Chapter 9, we showed how a potentially-remote attacker can exploit vulnerabilities in both software and hardware. In this chapter, we will show what more an attacker can achieve with physical access to the device.

An attacker with some level of physical access to a device is able to interact with the device in ways that do not form part of a defined interface such as a network protocol. This is necessary due to the tendency of designers to isolate security elements such as subscriber identity modules (SIMs) and Titan-M [50] in separate devices with a relatively small attack surface; this prevents the CPU-level attacks from Chapter 9, forcing attackers to find new attack surfaces to exploit.

Some of these attacks are non-invasive—e.g., observing electromagnetic emanations even at some distance from the device—while others are highly invasive—e.g., applying laser pulses to the exposed circuitry of a chip.

We will characterize attacks along a spectrum of invasiveness:

- Non-invasive: The attack can be mounted without physically tampering with the device in any way.
- Non-destructively invasive: The attack requires some physical tampering such as opening the device's enclosure, but does not cause any permanent damage and allows the device to be returned to its original form.
- Physically-evident: The attack causes irreversible physical damage to the device, and it is afterwards physically evident that an attack has taken place.
- Destructive: The device is permanently disabled or degraded by the attack.

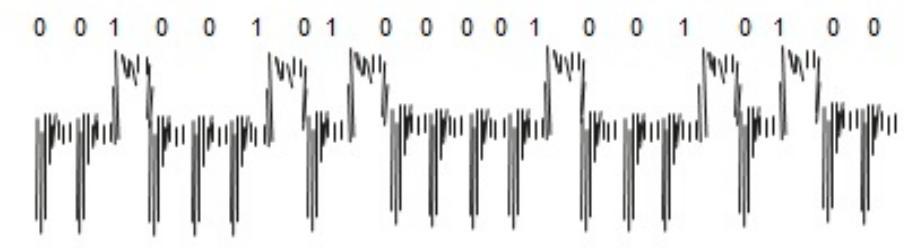


Figure 10.1: Illustration of a power trace of a modular exponentiation operation, adapted from [151].

10.1 Power analysis

Power analysis attacks uncover confidential information by analyzing a device's power consumption pattern over time. Logic gates consume energy while their transistors are in the process of switching, meaning that the operations being executed by the device and the data being operated on have an effect on power consumption.

Depending on where the power consumption is measured, such an attack may be non-invasive—e.g. if the device runs from external power through a line which can be tapped—or non-destructively invasive—e.g. if the attacker needs to open up the device in order to gain access to the power rails of a particular component of the device.

10.1.1 Simple power analysis

The power consumption of a device varies most noticeably according to the type of operation that is taking place. By observing power consumption patterns where instruction flow depends on data, attackers can identify which branch has been taken at a given point in the program, allowing an attacker to identify secret-dependent operations.

As an example of an operation that can be targeted by an simple power analysis (SPA) attack, consider an implementation of the Rivest-Shamir-Adleman (RSA) encryption and signing cryptosystem. It has decryption and signing operations of the form

$$message \equiv ciphertext^{key} \pmod{n}$$

using a naïve implementation of the square-and-multiply algorithm [82, p. 956]. Careful observation of the power trace collected during the operation can yield the secret key value. Figure 10.1 illustrates the power consumption patterns of a hypothetical square-and-multiply operation, showing how each bit in the exponent can be reconstructed from the trace.

A straightforward countermeasure against SPA attacks is to ensure that that the same sequence of instructions will always be executed, no matter the values of the data being processed. This normally incurs a performance cost, as it requires that all possible branches to be computed every time. However, it is not a perfect countermeasure, as the same instruction may consume different amounts of energy when provided with different data.

An alternative is to take the opposite approach: rather than trying to eliminate all secret data from the power traces, a program can perform additional random operations in order to create noise in the power trace, at the cost of computational overhead. SPA attacks are practical only for large power variations or high quality traces, but not on noisy measurements [151]. This is especially important for embedded devices like smart cards, that execute operations sequentially and so are more vulnerable to SPA than devices with a more advanced central processing unit (CPU) or ones based on an field programmable gate array (FPGA) [153].

10.1.2 Differential power analysis

Whereas SPA focuses on identifying computational characteristics in the shape of an individual power trace, differential power analysis (DPA) uses the statistical properties of many traces in order to identify subtle fluctuations that, within any individual trace, are overshadowed by noise and other measurement errors [150]. This allows DPA to more easily detect data-dependent changes in power consumption.

For example, in [150], the authors identify a function D that yields a bit which will be correlated with another bit that is computed during a Data Encryption Standard (DES) encryption operation, if and only if a particular part of the key has been guessed correctly. They then compute the difference between the average power traces when D yields the bit 0, and when D yields the bit 1. Where the partial key is guessed incorrectly, these differences will approach zero as more traces are averaged. However, if the partial key is guessed correctly, then these differential traces will contain spikes where the power consumption traces depend on the computed value. This process can then be used to undo the final round of the encryption operation, allowing a similar process to be applied to another part of the key. This process is repeated until the whole key is known.

In addition to the countermeasures described in Section 10.1.1, further mitigations are needed to protect against highly-sensitive DPA attacks. One such technique is to systematically randomize sensitive data while it is being processed. For example, algorithms can be implemented so that secret data is masked with a random value during computation, and only unmasked after the relevant computation has finished. An attacker who extracts intermediate values of the computation using DPA will only obtain the useless masked values. This requires the adaptation of algorithms to work with masked input and output values [151].

10.2 Electromagnetic emissions

Electric currents produce magnetic fields. When these currents change with time, an electromagnetic (EM) wave is produced, which can travel over a long distance, or be conducted along a power cable or other output. When a device performs a computation, these waves can be detected, revealing information on the computation and its data. Many EM eavesdropping attacks can be conducted non-invasively, and unlike most of the other attacks presented in this chapter, in some cases they can even be exploited from a distance without physical access to the device.

Exploitation of unintended emissions has a long history, with the phenomenon having been observed in cipher machine components during the Second World War [60, p. 90], and having been actively exploited by the 1950s [264, p. 91] to track radio receivers being operated by the Soviet Union in Britain.

This vulnerability first appeared in the open literature in the 1980s [157], most famously by Van Eck's demonstration of eavesdropping on computer monitors from a distance of 50m [92]; the signal from the monitor was received using an antenna, amplified, and combined with an artificial synchronization signal that allowed the captured emissions to be viewed on a normal display.

One approach is to reduce the strength of the emitted signal either by electromagnetic shielding, or by increasing the distance between devices and potential attackers [60, p. 91]; this can be achieved with static infrastructure with fencing or other physical measures, or with shielding inside the device that will reduce the available signal, and force the attacker to remove it, potentially rendering the attack physically-evident.

The designer may also attempt to reduce the strength of the emitted signal. One such approach was to replace the fonts used to display sensitive text with a low-pass filtered version that produced weaker emissions when displayed [157]. Careful hardware design can also reduce EM emissions. Asynchronous hardware designs—which avoid the use of a component-wide common clock—and dual-rail logic—which represent each bit with two physical values whose EM emissions will approximately cancel each other out [217]—produce less emissions and so are less vulnerable.

10.3 Fault-injection attacks

Fault-injection attacks are active attacks in which an attacker operates a device under conditions that cause it to deviate from its normal operation. In this way, transient faults during execution are introduced, to force the device to leak sensitive data, alter its control flow, or produce incorrect computation results. For example, an attacker can cause a fault during an RSA decryption or signing operation in such a way that the result of the computation contains information that can be used to recover the RSA private key [43]. Faults can be caused by altering any of a wide variety of conditions, such as supply voltage, clock frequency or quality, ambient temperature, or radiation [218]. Different

approaches allow for different levels of precision in the faults that can be generated, but may also be more or less invasive and may require differing levels of sophistication on the part of the attacker.

10.3.1 Glitching

One way to inject faults is to temporarily alter the voltage on a processor's power supply rail, or to prematurely flip the clock signal at a precisely-determined time—this alteration is known as a *glitch*—so that the digital logic of the device no longer functions correctly, causing spurious values to be placed into registers or other parts of the processor state [64]. A common approach is to target a fault at a conditional jump or test instruction [43] with the intention of bypassing an access control check. Such faults can in many cases be injected from outside the device, making the attacks non-invasive, e.g., SIM cards rely on an external source of power which can serve as an attack point. Other devices, such as mobile phones, may not expose their clock or power supply rails outside the device, therefore requiring at least some level of disassembly and modification of the device before mounting a voltage glitching attack, thus making the same attack at least non-destructively invasive in those devices.

These glitches can be produced in many different ways; perhaps the simplest is to use a transistor connected between the power supply rail to be glitched and ground; then, the transistor is switched on for a short period, briefly reducing the power supply voltage to near zero. The voltage glitch can then be further controlled by a FPGA device that manages the attack timing [64]. Commercial voltage-glitching tools are available, such as ChipWhisper [1], allowing these attacks to be performed without expert knowledge.

As a countermeasure, some embedded processors include a voltage regulator that attempts to maintain the stability of the power supply, independently of the input voltage [43]. Voltage regulation requires energy storage, normally in the form of a capacitor. These cannot be easily incorporated into a chip, and must be connected externally. This provides an opportunity for an attacker to bypass the regulator by connecting the voltage glitch source to the capacitor pin [64], but nevertheless prevents the attacker from carrying out the attack noninvasively. On-chip clock generators can be used in the same way to protect against clock glitches.

Alternatively, these attacks can be mitigated through *duplication* [43], with protected operations being executed multiple times in order to compare their results, thereby allowing glitches to be detected.

10.3.2 Focussed fault injection

Semiconductors are sensitive to light and charged particles, which can be used to introduce faults into a chip [43]. When a photon or ion hits a semiconductor with sufficient energy, it can ionize a region of the chip [239]. This causes a

[1] `https://www.newae.com/chipwhisperer`

non-conducting transistor to conduct, which can flip the output of a gate or a memory cell.

In order to exploit this phenomenon, it is necessary to precisely target a burst of light or ions at a specific region of the chip. In one experiment [239], this was achieved by directing a flash of light into the camera port of a semiconductor probing station set to 1500× magnification. Both a laser and white light—such as from a flashbulb—can be used as a light source, but lasers can produce a better-collimated beam that is more easily targeted. Best of all are focused ion beams, which have a very short wavelength and so can be focused on very small regions of the chip. The energy needed to induce a bit-flip depends on what is being attacked: the trapped charge used to set the value of a flash memory cell is much less than the charge that must be displaced in order to flip a static RAM (SRAM) cell. This attack is highly invasive, and will likely be physically-evident, as it is necessary to remove part of the packaging from the chip in order to expose the die.

There are two main ways to defend against this attack: the first is to shield sensitive regions of the die from external sources of light by covering them with a metal layer. This can be defeated by attacking with infrared or X-ray light, both of which possess some penetrating ability [239]. The other is to design circuitry so that its security cannot be compromised with only a single fault, and to include duplication and error-detection mechanisms that will allow the chip to detect or correct faults once they have been introduced [239]. Similar methods are also proposed as defenses against more the coarse-grained power and clock glitching attacks described in Section 10.3.1.

10.4 Microscopy and probing attacks

The most invasive attacks involve destructive modification to device components or even the dies of chips. For example, it is possible to remove metal layers from a die by soaking it in hydrofluoric acid for several seconds [154], thereby allowing photographs to be taken of different layers of the chip, combined with depth measurements that can be taken using a confocal microscope [187]. This allows much of the original design to be reconstructed. In some cases, this reconstruction is sufficient for the attacker's goals: for example, programming some types of read-only memory (ROM) results in visible changes to the structure of the chip, allowing data to be read out from microscope images [79]. Publicly available tools can process these images to read the data in the ROM [164]. Such attacks can be mitigated by choosing memory technologies whose visible structure does not change after being written. For example, antifuse memory does not visibly change after programming, unlike eFuse memories whose broken metal links are clearly visible [79].

More advanced attacks can use microscopic probes to interact with the circuit electrically, to read out the contents of buses or insert faults directly [239]. Techniques such as bus encryption can provide some protection, but the data must be decrypted somewhere in order to perform computation, and does not

prevent an attacker from injecting faults. Another defence is to insert a 'sensor mesh', one or more long space-filling wires that cover sensitive circuitry. If the wire is broken by an attacker seeking access to the functional layers of the device, then this can be detected by the chip, which can respond with measures such as overwriting any sensitive data [154]. However, even with the strongest countermeasures, it is impossible to completely prevent a determined attacker from compromising a system's hardware. For this reason we must design high-security systems so that they can tolerate at least some level of hardware compromise.

Part IV

What Next?

Chapter 11

Dealing with hardware compromise

In the previous three parts of this book, after exploring the "What?", "Why", and "How?" of mobile platform security (Parts I and II), we examined a range of attacks (Part III). In particular, the spate of software and side-channel attacks against hardware-assisted trusted execution environments (TEEs) have made it clear that placing unconditional trust in the inviolability of hardware-assisted security mechanisms is not reasonable. Modern processor hardware is necessarily complex in order to meet performance requirements essential for their viability in a competitive market. The complexity has led to characterizations like "hardware is the new software" [47]. This implies that design and implementation vulnerabilities are likely to continue.

In this part, we take a brief look at various possible strategies in designing and using TEEs in the face of possible compromise. We will discuss both *preventive* and *reactive* techniques.

First, in this chapter, we will look at possible approaches that can be taken to mitigate the impact of hardware compromise in current TEE architectures. In Chapter 12, we will discuss some novel hardware extensions, both from academia and industry, that can strengthen the security of next generation TEEs.

11.1 Multiple TEEs

Recall from Figure 5.1 that there are different architectural design patterns that can be applied concretely realizing a TEE. The dominant design pattern today is the processor secure environment found in both ARM TrustZone and Intel Software Guard Extensions (SGX). In this style, the same processor executes both rich execution environment (REE) and TEE software with logical isolation achieved via hardware support. The reason why this design pattern has

become dominant is economic: it avoids the increased bill-of-materials cost of a physically distinct TEE realization [178]. Since logical separation implies a greater degree of sharing between the REE and TEE than does physical separation, this approach leads to a greater attack surface, especially for side-channel attacks.

An alternative is to resort to physically distinct external security co-processors despite the increased cost. As we noted in Chapter 5, there is already a clear trend in the market in this direction with Google's Titan and Apple's Secure Enclave Processor (SEP) which complement ARM TrustZone in their respective smart devices.

One mitigation is to use a second TEE in the form of such a co-processor to ensure the integrity of remote attestation even in the event of an attack that compromised confidentiality (but not integrity) of the primary TEE. Recent inclusion of specialized hardware units (like Google Titan), support such approaches by introducing a co-processor in commodity hardware. When ensuring attestation integrity, all sensitive key material is stored within the co-processor, such that if the larger general-purpose TEE is compromised, the keys remain confidential thus preventing an adversary from impersonating a valid machine.

Although these additional TEEs are currently used to protect systems resources only, it is possible to design schemes using multiple TEEs so that even in the event of hardware compromise or failure in some of them, some security guarantees could still be maintained [4].

11.2 Application-specific techniques

TEE functionality is used to design solutions that provide security and privacy guarantees in various applications. If there is a vulnerability in the TEE itself, and an adversary successfully exploits it to circumvent these guarantees, it may leave behind tell-tale cues in the state of the particular application under consideration. By monitoring for such cues, it may be possible to detect and/or account for potential TEE compromise.

An example of such application-specific mitigation can be seen in the design of *proof of elapsed time (PoET)* [143], a Sybil-resistance mechanism [91] proposed by Intel as a replacement for proof of work (PoW) [145] used in Bitcoin-like blockchains. PoW is used to choose which participant in the blockchain (playing the role of block "miners") wins the right to append the next block in the blockchain. To do this in a fair manner, PoW requires miners to solve a cryptographic puzzle for a given block. The miner who finds the solution first is declared the winner. The probability of winning is proportional to the amount of computational work that the miner has at its disposal. The scheme is fair in the sense the available computational resources is the only factor that influences the chance of a miner winning. However, the computation necessary is wasteful, leading to the extravagant energy consumption associated with PoW.

PoET aims to achieve the same probability distribution for a miner winning the right to append the next block. But instead of relying on a computational puzzle, PoET relies on having the processor wait for a certain duration of time (chosen randomly from the desired distribution), and issue a certificate from the TEE attesting that the processor has indeed waited for the specified time. The miner with the least waiting time is the winner.

The designers of PoET wanted to mitigate the impact of an adversary compromising a small number of TEEs. Rather than attempting to detect compromise, their approach is not allow a TEE to win the next block if its rate of winning in the recent past has exceeded a certain threshold. A TEE may exceed the threshold either because the adversary has compromised it or because of sheer good luck. By suitably choosing the threshold, the latter possibility can be made sufficiently small. While this strategy does not account for all possible ways in which an adversary who compromises a small set of nodes can take advantage of the system, it does prevent an attacker with a single compromised machine from winning arbitrarily many blocks.

Although PoET is an example in a non-mobile setting, similar application-specific mitigation may also be possible in mobile device settings. Application-specific mitigation can be seen as an instance of *cross-layer design for security*, a term coined by Frank Piessens [210]. Abstractions used in computer science to cleanly isolate one layer or building block from another can deprive the designer of the ability to design and deploy novel mitigations. Piessens notes that application-specific information (e.g., about which data is confidential) is typically lost during compilation. By making such information available at run-time, large classes of attacks can be defended against. In other words, while "leaky abstractions" [210] can lead to side-channel attacks, carefully mediating useful information across abstraction boundaries can also help the defender. An example of this approach is *HardScope* [200] where a small number of additional processor instructions (and accompanying logic) combined with compiler support for instrumentation, can enforce variable visibility (scope) information at run-time and can thus be used to prevent various data-oriented attacks that exploit memory vulnerabilities.

Chapter 12

Towards next-generation TEEs

In this chapter, we will discuss potential approaches that can be used to strengthen the security guarantees in the next generation trusted execution environments (TEEs). We will cover both academic proposals, including some early ones, as well as recent industry developments.

12.1 CPU-based TEE architectures

12.1.1 AEGIS

AEGIS [246] was the first published example of an enclave-based trusted execution architecture, with processes being provided with a *tamper-evident* or *tamper-resistant* processing environment.

The AEGIS architecture follows a "critical section"-like approach: a program will signal to the processor that it should enter tamper-evident mode with an `enter_aegis` instruction, which also specified the number of subsequent bytes to measure before continuing with their execution. This measured code section can then perform additional checks of other code sections and the execution context. While in tamper-evident mode, the program can use a `sign_msg` instruction to remotely attest the code section marked by the `enter_aegis` instruction executed to enter tamper-evident mode. Once the sensitive computation is complete, the program executes an `exit_aegis` instruction to return the processor to its normal execution mode.

[246] also explores the trade-offs that need to be made when deciding how to apportion trusted functionality between the processor and the operating system. The authors argue that a security kernel is more flexible and requires less modification to the processor, but that this approach requires the verification of more trusted code and may reduce performance.

12.1.2 Late-launch-based TEEs

Flicker [180] uses the late launch functionality of modern Intel x86 central processing units (CPUs) to provide a TEE. These CPUs implement the `skinit` instruction, which stops all other execution before measuring and executing a designated piece of code in cached memory. This measurement process is integrated with the Trusted Platform Module (TPM), allowing persistent sensitive data associated with that piece of code to be protected by sealing.

Flicker uses late launch to execute a trusted application, whose data is protected by virtue of the fact that no other code runs concurrently with it, and that at completion the application erases any sensitive data and returns the system to its original state.

SMART [97] provides similar functionality for non-x86 devices, describing hardware modifications needed to provide late launch on an embedded CPU.

12.1.3 Sanctum

Sanctum [83] modifies the RISC-V architecture to provide enclave-type TEEs, with the explicit goal of mimicking the functionality of Software Guard Extensions (SGX) as closely as possible. Sanctum provides memory isolation, with isolation also extending to the last-level cache; this prevents the use of cache-timing side-channel attacks against enclaves.

12.1.4 Keystone

Keystone [165] is another enclave-type TEE for the RISC-V architecture. However, unlike Sanctum, Keystone is built as a software layer atop an unmodified RISC-V architecture, allowing its functionality to be configured according to the needs of the application in question. However, though the architecture itself is unmodified, Keystone requires that the hardware platform provide trusted boot and hardware randomness; and that the manufacturer perform key provisioning (Section 4.1.1). Nevertheless, realizing these is significantly less burdensome than implementing a fully hardware-based TEE.

12.1.5 MI6

The MI6 processor [63] is designed to provide secure enclaves that are resistant to microarchitectural side-channel attacks despite aggressively speculative out-of-order execution. MI6 partitions the entire memory hierarchy which, for instance, facilitates protection against cache bandwidth side-channels not covered by Sanctum. Thus, MI6 enclaves are guaranteed not to share cache lines with other software. Other MI6 changes eliminate a number of smaller timing differences in the cache-access pipeline. A new `purge` instruction is used after protection domain transitions, emptying the pipeline, resetting the branch predictor, and flushing the L1 cache.

12.1.6 TrustLite & TyTAN

TrustLite [152] provides enclaves for embedded devices without a memory management unit (MMU). Each enclave is configured so that its working memory can be read only when the CPU is executing an instruction from a permissible memory region, and its code cannot be overwritten at all.

This approach protects enclaves from the operating system (OS), with only slightly greater hardware complexity than a split-world architecture, but without the need for a security kernel. This makes TrustLite suitable for small embedded devices that lack the resources to run a normal security kernel. TyTAN [67] builds a complete FreeRTOS [38]-based system around TrustLite on top of the Intel Siskiyou Peak platform.

12.1.7 Sanctuary

Sanctuary [66] provides user-space enclaves using TrustZone, taking advantage of access control features included in Arm's TZC-400 TrustZone Address Space Controller (TZASC). Rather than basing access control on just a secure/normal-world bit, the TZC-400 allows access control based on the source device. Sanctuary dedicates cores of a multi-core processor to the execution of enclave code, with access to the memory allocated to an enclave being restricted to this core only. This protects enclaves even from a compromised OS. Only one enclave can run at a time on a given core, thereby protecting enclaves from each other.

Unlike normal TrustZone-based systems, Sanctuary trusted applications do not run in secure mode, providing a greater level of isolation between enclave code and the security kernel; this significantly reduces its attack surface.

12.1.8 TIMBER-V

TIMBER-V [262] provides enclave functionality with highly efficient dynamic memory management, and suitable for implementation on low-end processors. In addition to typical enclaves, supported by trusted user mode, TIMEBER-V also adds a trusted supervisor mode that can be used to realize trusted system services within an untrusted operating system. TIMBER-V is based on a memory-tagging architecture, where each word of memory contains several "tag bits" that can be used for access control. TIMBER-V uses two-bit tags, representing normal memory, enclave memory, trusted entry point, and trusted supervisor memory.

These tags allow applications to interleave regions of memory whose access is restricted to different security domains: client applications (CAs) can only access normal memory, trusted applications (TAs) can access enclave and normal memory, and the trusted supervisor can access everything. In addition, the CPU switches between different security domains according to the tag applied to the instruction being loaded.

This approach allows memory to be transferred between an application and its enclave very efficiently, by simply adjusting the tag on each word.

12.1.9 CURE

The CURE enclave architecture [41] attempts to collect all previously proposed techniques for enclave hardware support into one design, prototyped on RISC-V. Enclave memory (temporally dedicated to CPU cores) is configured with dedicated cache ways, thereby mitigating the risk of side-channel attacks. Enclave identifiers are not only included into core memory management, but also added to system bus arbiters for both CPU and direct memory access (DMA) controller; this allows peripherals to be temporally associated with enclaves. Such an approach has not been fully explored in prior designs.

The authors claim that CURE is flexible enough to implement different kinds of hardware-assisted enclaves (as viewed from the OS), including kernel-space enclaves for isolating individual parts of kernel operations such as memory management, virtual machine enclaves for traditional workload separation, as well as *subspace enclaves*, which can be seen as an isolation/fragmentation mechanism usable at the highest privilege levels to protect against memory corruption vulnerabilities or for trusted computing base (TCB) minimization. The opportunity to be flexible in what types of enclaves to support stems from that the trust boundaries of an enclave is memory and not specific to privilege levels or applications; thus it is possible to construct enclave boundaries within traditional execution units, similarly to what can be achieved with hardware capabilities such as CHERI [263].

12.1.10 Arm Confidential Compute Architecture

In 2021, Arm released its plan for its next generation architecture (ARMv9.3A). As part of this hardware update, the isolation features that constitute the foundation of Arm TrustZone have been extended to include two new memory isolation contexts, one to be used for the EL3 firmware, and the other, more noteworthy, is the *realm mode*, which is a new "split world" approach proposed to be used for enclaves, in the form of more dynamic isolation contexts with a minimal TCB. This new architecture is named Confidential Compute Architecture (CCA), a clear reference to the ongoing enclave specification work at the Linux Consortium for Intel X86 instruction architecture (X86) hardware (AMD Secure Encrypted Virtualization (SEV) and Intel SGX/Total Memory Encryption (TME)). Arm notes that CCA can be used both in cloud computing as well as at the edge (in terminal devices) [29].

The CCAs architecture is depicted in Figure 12.1. The technical details of the realm mode are to be released over the next few years, but the isolation of the Realms is achieved by a new memory firewall, the Granule Protection Table (GPT) applied after memory translation [76]. The GPT is configured by EL3, and partitions physical memory for Realm mode, Secure mode, Monitor (EL3) and Non-Secure mode address spaces.

Similar to the Non-Secure (NS) bit in Arm TrustZone, now two bits (corresponding to four possible worlds) will permeate throughout the memory access framework, and especially caches, to map access control configuration to

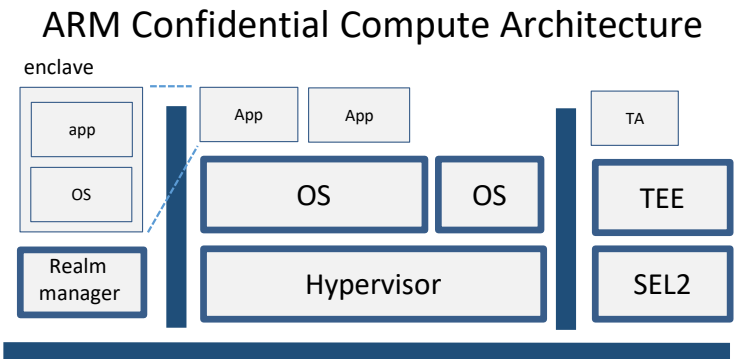


Figure 12.1: Arm CCA: Extending the Arm split-world architecture towards enclave support

the firewalling. At the moment it remains unclear whether the extended configuration is visible on the internal bus for external hardware components to react to the distinction. Just like with AMD SEV and Intel TME, the partitioned address spaces will be assigned their individual memory encryption keys, and page encryption will be tweaked per address when being encrypted as stored to dynamic random access memory (DRAM) [76]. This makes the realm memory contents inaccessible to parallel or higher privilege levels, even for such components that may have configuration rights to the aforementioned firewall.

In contrast to x86 enclave designs, in Arm CCA, enclave key and lifecycle management, attestation and device authentication are implemented in software, with the realm manager, i.e. the software running in realm hypervisor mode, playing a central role. It is likely that ARM will propose a common, open-source base for this software, as was done in the case of Arm Trusted Firmware which is today the EL3 reference code and definition for e.g. power management control and world-switching logic in Arm devices. It can be expected that original equipment manufacturers (OEMs) and chip vendors will adapt such a future common realm manager to their own needs, resulting in a common baseline for functionality like enclave setup and attestation, but whether the trust root (keys) will be rooted with Arm or OEMs is unclear at the moment.

12.2 Beyond TEEs

The TEEs in modern CPUs provide tight integration between TAs and rich execution environment (REE) applications, allowing high performance compared to the use of secure co-processors. However, in many computationally-bound applications such as machine learning, the bulk of the work is offloaded to a co-processor such as a graphics processing unit (GPU) outside of the CPU

itself. The goal of the following works is to allow generic computation to be offloaded to the accelerator with minimal hardware modifications and without sacrificing the isolation provided by a TEE.

12.2.1 Slalom

For existing TEEs, an example of the current state of the art in combining TEEs with offloaded computation is Slalom [254], which uses cryptography to allow the bulk of the computation for neural network evaluation to be safely offloaded to an untrusted GPU. Each layer of a neural network has a linear and a nonlinear component, taking as input a vector $\overrightarrow{x}$ and outputting a value $y = f(W\overrightarrow{x})$, where W is a weight matrix. The key observation is that most of the computational complexity is in the linear part $W\overrightarrow{x}$. The TEE can therefore outsource this computation to the GPU, and use Freivalds' algorithm [102] to probabilistically verify the result with lower complexity than the computation itself. Furthermore, since the computation being offloaded is linear, the TEE can send cryptographically blinded data to the GPU, at the cost of some precomputation. It does this by taking pseudo-random blinding vector $\overrightarrow{r} \in \mathcal{Z}_n$, then precomputing and securely storing an 'unblinding vector' $W\overrightarrow{r}$, where W has been discretized to an element of $\mathcal{Z}_n$. Then, for the online part, the TEE has the GPU compute $W \cdot (\overrightarrow{x} + \overrightarrow{r})$, uses Freivalds' algorithm to verify the result, and subtracts the precomputed unblinding vector $W\overrightarrow{r}$.

Despite the overhead of the verification process, Slalom provides a significant speedup relative to computation in the TEE: 6–20× without blinding, and 4–11× with blinding. Nevertheless, this is slower than evaluating the neural network using a GPU in the normal way, and so other works consider how to extend the trust boundary to the GPU.

12.2.2 Secure GPU accelerators

Heterogeneous Isolated EXecution (HIX) [147] allows isolated computation using SGX, an unmodified GPU and minor changes to the I/O interconnect. Suppose an application enclave wishes to send private data to the GPU for computation. First, it communicates with a special-purpose SGX enclave, called the GPU enclave, that alone is able to control the GPU. The GPU enclave mediates between the application enclave and the GPU, establishing a shared key that will be used to encrypt the data. The application enclave encrypts its data and stores it in a protected region of memory; it then instructs the GPU enclave to send an encrypted command to the GPU to copy the data to the GPU's internal memory via DMA. The data is then decrypted using the normal computational facilities of the GPU, allowing the desired processing to take place.

As encryption and decryption occur only when data is transferred to and from the GPU, the overhead is quite low for tasks whose workload is dominated by computation within the GPU. This is the ideal situation, since less computationally-demanding tasks can be carried out efficiently using the CPU.

However, because the GPU is unmodified, it is impossible to perform remote attestation of its workloads. This means that an adversary that can manipulate the PCI Express (PCIe) link or replace the GPU itself can defeat the security of the TEE. Consequently, the TEE as a whole cannot perform meaningful remote attestation unless the operator of the system is trusted *a priori*.

A later research project presents rack-scale heterogeneous TEEs [275]. This work specifically targets rack-scale server deployments, where one node on the rack acts as a secure controller that mediates access to the PCIe fabric. Compute resources are then completely dedicated to specific users and hard-rebooted between uses. The secure controller can be remote attested, and due to the full reboot, the computer resources are known to be in a known state on use. Whereas other heterogeneous TEEs require hardware changes, [275] can be realized on existing hardware but is restricted to a server setting.

In contrast to HIX and rack-scale heterogeneous TEEs, Graviton [258] adds TEE functionality to the GPU, allowing remote attestation and explicit support for encrypted data transfers. This mitigates the greatest weakness of HIX, namely that it cannot protect against an attacker capable of interfering with the system hardware, at the cost of no longer being compatible with commodity GPUs.

12.2.3 Morpheus

Morpheus is a run-time security mechanism with goals similar to memory-safety features in contemporary processors (Chapter 7), but takes a starkly different approach [103]. The underlying idea of Morpheus is that correct program behavior depends on defined program behavior specified by the program code, whereas exploitation of memory vulnerabilities depends on undefined behavior. Consequently, Morpheus aims to randomize all undefined operations of a program implementation, such as memory layout and instruction encoding, in order to prevent an attacker from introducing reliable attacks against the program. Specifically, Morpheus implements a RISC-V extension for *domain tagging*, *pointer displacement*, *domain encryption*, *churn*, and *attack detection*.

Program memory is split into four domain: code, data pointers, code pointers, and other memory. Initial domain assignments are specified by the compiler. The memory and registers are extended with 2-bit domain tags such that the hardware can track the tags during computation. The pointer displacement feature adds a level of memory address translation such that all addresses are displaced based on a periodically-changing displacement value. Domain encryption is applied to all code and pointers. Churn, i.e. re-randomization, is periodically applied such that all encrypted domains are re-encrypted with a new key and the displacement value is set to a new random value. While an attacker could leak specific randomization values, those values are valid only until the next churn takes place. Finally, attack detection attempts to detect probing attempts that aim to find the randomization values and triggers continuous churn when a possible attack is detected.

The churn is implemented in hardware such that the program needs not be stopped to perform re-randomization. Instead, the hardware does a linear sweep of memory and keeps track of the progress such that it can transparently use either the new or old displacement and key where appropriate during re-randomization. Due to the hardware implementation with domain-tagging and concurrent churn Morpheus achieves good performance, incurring only a 0.8% overhead on average when measured with the SPEC CPU 2006 C-language benchmarks. The security of Morpheus relies on the churn period being smaller than the time an attacker needs to break the randomization. Consequently, Morpheus might not be effective against attacks that exploit a direct information leak to break address-space layout randomization (ASLR). Nonetheless, even the suggested churn period of 50ms, with the above overhead, is more than sufficient to prevent known ASLR probing mechanisms.

12.2.4 Blinded computing

One way to securely process data without having direct access to it is homomorphic encryption, which allows computation to be performed on the encrypted data directly, without ever revealing the plaintext to the party performing the computation [257]. Slalom, described in Section 12.2.1, is a special example of this approach. Homomorphic encryption offers strong guarantees as it protects the data both against side-channels and even malicious programs that might intentionally attempt to leak the data. Unfortunately, in many cases homomorphic encryption incurs unacceptable overhead when used for general computation.

Another approach is to use traditional TEE architectures that provide TA attestation and allow confidential processing of data. However, programs today need to process an ever increasing amount of confidential data. Meanwhile, TEE context switches are expensive and increased functionality increases the likelihood of security-related bugs within the TEE software. Blinded computation presents an alternative that minimizes the need for a separate execution environment or costly cryptographic measures.

Blinded computation combines software-guided instrumentation with dynamic taint-tracking to provide practical side-channel resistant computation on blinded data, with a minimal TCB restricted to a small single-purpose TEE. Its adversary model is similar to a traditional TEE but with strong protection against side-channels; unlike previous taint-tracking architectures [269], the adversary can also corrupt any code with run-time attacks such that the adversary can run arbitrary code. A small TEE is used to only provide initial blinded inputs to a program. The hardware then enforces a policy that prevents any computation from directly leaking data, or exposing it via side-channels. Unlike static analysis, this approach protects data from being exposed even when an attacker exploits a software vulnerability to run their own code. Since the blindedness of the the data is ensured by hardware-enforced access control rather than cryptographic blinding, this approach avoids the overhead of approaches based on homomorphic encryption.

Since no blinded data can be written to external peripherals, all output is passed back to the small single-purpose TEE which then can securely re-encrypt the computation results before they are sent back to the client that initially provided the input.

Dynamic dataflow tracking has similarities to prior approaches that, for instance, use it to prevent confidentiality by tracking information flows from *taint sources* to *sinks* (for instance, passwords and output functions, repsectively) [247]. Blinded computation relies on a compiler instrumentation and software data-flow verification to prevent the copious false-positives that plague some naive taint-tracking policies [240].

Best-practices for writing side-channel resistant code show that it is possible to construct code that is side-channel resistant. Recent compiler-based approaches have further demonstrated that it is also feasible to construct such code automatically [62]. Blinded computation uses similar techniques to automate software transformation and provide compile-time policy verification. Note that purely software-based approaches do not prevent memory errors from directly addressing the confidential data, nor do they provide protection from a malicious OS, both of which the hardware-enforced policy of blinded computation protects against. Recent work has also shown that taint-tracking can be used to prevent micro-architectural attacks such as Spectre [270, 149]; allowing a hardware-based to avoid restrictive policies while still allowing speculation. Taken together, blinded computation provides security comparable to homomorphic encryption, with some hardware modification, but minimal development and performance overhead.

12.2.5 In-memory computing

For many decades, computing efficiency has been increasing exponentially in accordance with Moore's law, but in recent years the continued miniaturization and speed-up of CPUs has hit against the limitations of physics. At the same time, computing has become increasingly data-centric, placing a performance burden especially on memory I/O.

One emerging, architectural research direction is *in- or near-memory computing* [192],[230] where memory cells can directly targeted by algorithm computation in programmable logic that either is co-located with the memory framework or physically nearer to it than the CPU on the other end of the memory bus. Although in-memory computing can have obvious performance benefits when processing large amounts of data in use cases such as video filtering or some machine learning algorithms, some obvious security applications can be envisioned for such architectures also: Physical unclonable functions (PUFs) could leverage the physical randomness of memory cells and provide error-correction functionality co-located with the memory, in essence providing device-local key material. Similarly, true random number generators (TRNGs) are an obvious use-case that could be implemented, based on memory cells as a source of randomness.

The more architecturally diverse the programmability in devices becomes, the more difficult it will be to argue about attack vectors and isolation: E.g. if an attack can be embedded into the memory framework (firmware), can it then circumvent all existing memory and firewall protections that we rely on today? If memory is programmable, can we leverage the physical layout of memory cells as a guaranteed isolation boundary for separating security domains in the device (in a similar fashion to memory tagging). The next 5-10 years will tell us this.

12.3 Conclusion

The evolution of hardware security mechanisms on mobile devices is significantly more brisk than it has been in the past. In parallel with the introduction of new coarse-grained isolation mechanisms like Arm CCA that are suitable for constructing TEEs and enclave architectures, architectural additions for integrity and memory safety (like Arm pointer authentication (PA), Memory Tagging Extension (MTE), Branch Target Identification (BTI) and Intel Control-flow Enforcement Technology (CET)) are already being deployed at scale, and academic research continues to produce new, even better concepts.

Further hardware innovations are awaiting imminent adoption. With the breakdown of Moore's law, speed of computation is increasingly achieved via *heterogenity*, incorporating dedicated hardware cores sharing memory access with general-purpose cores. All forms of digital signal processors (DSPs), including neural processing units (NPUs) and GPUs are good examples of components in such designs. It remains a partially open issue how secure domains are built across such components in heterogeneous hardware, particularly because such purpose-built cores are usually not equipped with built-in protection mechanisms such as privilege rings or isolation domains that are now commonly found in general-purpose processors. Another potential design approach for mobile device processors is *hardware reconfigurability*. Already today, network equipment such as base stations or routers incorporate integrated field programmable gate array (FPGA) or Coarse Grained Reconfigurable Array (CGRA) hardware for optimizing packet processing speed, routing or cryptographic operations. But the fact that these components are re-programmable, with very low-level access to memory and arithmetic logic units (ALUs), also makes them potential entry points for malware and attacks. On one hand, such hardware can serve as a re-configurable way to implement hardware-assisted security mechanisms, but on the other hand, they may pose a threat to overall device integrity.

Finally, new memory technologies such as magnetoresistive random-access memory (MRAM) and phase-change random-access memory (PRAM) point us towards a future where system memory will be *persistent*, retaining state even across power-cycles. This, too, is a double-edged sword: it can significantly shorten the boot-up time for mobile devices, and can improve security by allowing security mechanisms (such as system counters in memory) to remain

unconditionally stateful; but at the same time successful system attacks may end up becoming memory resident. This may require us to revisit many of our integrity (and stability) solutions we use today that are based on the presumption that random access memory (RAM) is volatile, such as the role of boot-up integrity metrics or firmware updates vs. live system patching from the perspective of memory.

All in all, the field of (mobile) platform security is now an interesting, fast moving field of research and engineering. We hope this book has provided you with an overview of the contemporary security architectures in mobile devices, explaining the de-facto mechanisms and tools that has kept our personal devices safe during the last two decades, outlining what is changing and improving in the immediate future on the platform, and identifying the expected challenges ahead.

Appendix A

Commercial TEE deployments

Since TEE technology, and in particular TrustZone, has been deployed in large scale, a number of TEE vendors have emerged over the years. The majority of these are with proprietary implementations of the TEE software stack. Table A.1 lists TEE vendors for TrustZone, TrustZone-M and the RISC-V architecture.

[1] https://globalplatform.org/wp-content/uploads/2019/07/01-CR-1.0_GP180004-Certificate-and-Certification-Report_20190712.pdf

[2] https://www.commoncriteriaportal.org/files/epfiles/CC%20Huawei%20iTrustee%20Software%20V2.0%20Security%20Target%202.1.pdf

[3] https://www.trustonic.com/solutions/iot-security/

[4] https://www.op-tee.org/

[5] https://www.provenrun.com/products/provencore/

[6] https://www.qualcomm.com/products/features/mobile-security-solutions

[7] https://www.rockycore.cn/index.html

[8] https://optimumdesk.com/it-solutions/data-loss-prevention-privacy

[9] https://www.sierraware.com/open-source-ARM-TrustZone.html

[10] https://www.trustkernel.com/en/products/tee/t6.html

[11] https://developer.samsung.com/teegris

[12] https://source.android.com/security/trusty

[13] https://www.taf.net.cn/Tee_detail.aspx?_ID=2349283b-6311-4617-862c-112234544354

[14] https://globalplatform.org/certified-products/watchtrust-2-1-1-on-sc9860-2/

[15] https://globalplatform.org/wp-content/uploads/2019/09/GP-SE-2019_02-CR-1.0_GP190006-Certificate-and-Certification-Report_20190909.pdf

[16] https://www.trustonic.com/technical-articles/kinibi-m/

[17] https://www.st.com/en/embedded-software/provencore-m.html

[18] https://hex-five.com/first-secure-iot-stack-riscv/

TEE	Vendor	GP Certification Functional	GP Certification Security	License	Note
		TrustZone			
Cloud Link TEE[1]	Pingtouge Semiconductor	✓	✓	proprietary	
iTrustee[2]	Huawei			proprietary	Formerly Secure Core
Kinibi[3]	Trustonic	✓		proprietary	Formerly Mobicore and <t-base
OP-TEE[4]	Linaro			BSD 2-Clause	
ProvenCore[5]	Prove & Run			proprietary	
Qualcomm TEE[6]	Qualcomm			proprietary	Formerly QSEE
Rocky Core[7]	Suzhou Rong Card Intelligent Technology	✓		proprietary	
SecuriTEE[8]	Solacia			proprietary	
SierraTEE[9]	SierraWare			proprietary	
T6[10]	TrustKernel			GNU GPL	
TEEGRIS[11]	Samsung		✓	proprietary	GP certification for Mediatek MT6737T
TLK [198]	Nvidia			MIT	
Trusty[12]	Google			MIT	
TURBOTEE[13]	Eastcompeace Technologies	✓		proprietary	
Yunos TEE	Taobao Software	✓		proprietary	
WATCHTRUST[14]	Watchdata	✓	✓	proprietary	
Upteq NFC422 v1.0[15]	Gemalto (Thales Group)	✓		proprietary	
		TrustZone-M			
Kinibi-M[16]	Trustonic	N/A		proprietary	
ProvenCore-M[17]	Prove & Run	N/A		proprietary	
		RISC-V			
MultiZone[18]	Hex Five Security	N/A		MIT	

Table A.1: Commercial TEE implementations. The GP Certification column indicates TEEs that have received TEE Initial Configuration v1.1 functional certification and/or TEE Security certification according to `https://globalplatform.org/certified-products/`.

Bibliography

[1] Martín Abadi et al. "Control-Flow Integrity". In: *Proceedings of the 2005 ACM Conference on Computer and Communications Security*. 2005. DOI: 10.1145/1102120.1102165.

[2] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual*. 2021. URL: https://www.amd.com/system/files/TechDocs/40332.pdf (visited on 01/16/2022).

[3] Periklis Akritidis et al. "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-bounds Errors". In: *Proceedings of the 2009 USENIX Security Symposium*. 2009.

[4] Fritz Alder. "TEE^2 – Combining Trusted Hardware to Enhance the Security of TEEs". https://falder.org/tee2-thesis.pdf. Technical University of Darmstadt, 2019.

[5] Alibaba. *Alibaba Cloud Released Industry's First Trusted and Virtualized Instance with Support for SGX 2.0 and TPM*. 2020. URL: https://www.alibabacloud.com/blog/alibaba-cloud-released-industrys-first-trusted-and-virtualized-instance-with-support-for-sgx-2-0-and-tpm_596821 (visited on 01/29/2021).

[6] Tiago Alves and Don Felton. "TrustZone: Integrated Hardware and Software Security". In: *Information Quarterly* 3.4 (2004), pp. 18–24.

[7] Android Open Source Project. *Android 6.0 Compatibility Definition*. 2015.

[8] Android Open Source Project. *HWAddressSanitizer*. 2020. URL: https://source.android.com/devices/tech/debug/hwasan (visited on 10/27/2020).

[9] Android Open Source Project. *Scudo*. 2020. URL: https://source.android.com/devices/tech/debug/scudo (visited on 10/27/2020).

[10] Apple. *Data protection*. 2018. URL: https://support.apple.com/guide/security/data-protection-overview-secf6276da8a/web (visited on 03/06/2022).

[11] Apple Inc. *Apple T2 Security Chip Security Overview*. https://www.apple.com/mac/docs/Apple_T2_Security_Chip_Overview.pdf. 2018.

[12] Apple Inc. *iOS Security — iOS 12*. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf. 2018.

[13] Apple Inc. *Preparing Your App to Work with Pointer Authentication*. 2020. URL: https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication (visited on 10/27/2020).

[14] Apple Inc. *Swift*. 2020. URL: https://developer.apple.com/swift/ (visited on 10/25/2020).

[15] *Apple Platform Security*. Apple, 2020.

[16] Apple. Inc. *About FaceID Advanced technology*. 2020. URL: https://support.apple.com/en-gb/HT208108 (visited on 03/06/2022).

[17] William A Arbaugh, David J Farber, and Jonathan M Smith. "A secure and reliable bootstrap architecture". In: *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE. 1997.

[18] ARM Ltd. *ARM Security Technology - Building a Secure System using TrustZone Technology*. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c. 2009.

[19] ARM Ltd. *Arm TrustZone technology for ARMv8-M Architecture, Version 2.1*. https://static.docs.arm.com/100690/0201/armv8_m_architecture_trustzone_technology_100690_0201_01_en.pdf. 2018.

[20] ARM Ltd. *Arm® Platform Security Architecture Trusted Base System Architecture for Arm®v6-M, Arm®v7-M and Arm®v8-M 1.0*. ARM, 2018. URL: https://armkeil.blob.core.windows.net/developer/Files/pdf/PlatformSecurityArchitecture/Architect/DEN0083-PSA_TBSA-M_1.0-bet1.pdf.

[21] ARM Ltd. *Armv8-A architecture reference manual, DDI 0487F.c*. DDI 0487F.c. 2020.

[22] ARM Ltd. *Armv8-M Architecture Reference Manual, Version A.k*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0553a.k/DDI0553A_k_armv8m_arm.pdf. 2019.

[23] ARM Ltd. *Armv8.5-A Memory Tagging Extension*. Whitepaper. 2019.

[24] ARM Ltd. *Hardware Accelerated Crypto | Mbed OS 5 Documentation*. 2021. URL: https://os.mbed.com/docs/mbed-os/v6.15/porting/hardware-accelerated-crypto.html.

[25] ARM Ltd. *Isolation using virtualization in the Secure world: Secure world software architecture on Armv8.4, Version 1.0*. https://developer.arm.com/-/media/Files/pdf/Isolation_using_virtualization_in_the_Secure_World_Whitepaper.pdf. 2019.

[26] ARM Ltd. *Power State Coordination Interface*. http://http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0022d. 2017.

[27] ARM Ltd. *SMC Calling Convention*. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c. 2016.

[28] ARM Ltd. *Software Delegated Exception Interface*. http://http://infocenter.arm.com/help/topic/com.arm.doc.den0054a. 2017.

[29] ARM Ltd. *Unlocking the power of data with ARM CCA*. 2021. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/unlocking-the-power-of-data-with-arm-cca (visited on 03/06/2022).

[30] Arm Ltd. *Arm Morello Program*. Arm Developer. 2020. URL: https://developer.arm.com/architectures/cpu-architecture/a-profile/morello (visited on 11/29/2020).

[31] Arm Ltd. *FIPS 140-2 Non-Proprietary Security Policy*. ARM, 2018. URL: https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp3263.pdf.

[32] *Arm® Platform Security Architecture Security Model 1.0*. ARM Ltd, Feb. 2020.

[33] *Arm® Platform Security Architecture Trusted Boot and Firmware Update 1.0*. ARM Ltd, 2019.

[34] Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. 1st. Berkely, CA, USA: Apress, 2015. ISBN: 9781430265832.

[35] N. Asokan et al. *Mobile Platform Security*. Vol. 9. Synthesis Lectures on Information Security, Privacy, & Trust. Morgan & Claypool Publishers, 2014. ISBN: 9781627050975. URL: http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=688023&site=ehost-live&authtype=sso&custid=ns192260.

[36] Atsec information security corporation. *Cryptographic Module for Intel® vPro™ Platforms Security Engine Chipset*. 2016. URL: https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp2720.pdf (visited on 07/18/2016).

[37] Roberto Avanzi. "The QARMA Block Cipher Family. Almost MDS Matrices over Rings with Zero Divisors, Nearly Symmetric Even-Mansour Constructions with Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes". In: *IACR Transactions on Symmetric Cryptology* 2017.1 (Mar. 2017), pp. 4–44. DOI: 10.13154/tosc.v2017.i1.4-44. URL: https://tosc.iacr.org/index.php/ToSC/article/view/583.

[38] AWS. *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. 2021. URL: https://www.freertos.org/ (visited on 09/19/2021).

[39] Ahmed M. Azab et al. "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World". In: *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS)*. 2014, pp. 90–102. DOI: 10.1145/2660267.2660350.

[40] Brandon Azad. *Project Zero: Examining Pointer Authentication on the iPhone XS*. Feb. 1, 2019. URL: https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html (visited on 02/12/2020).

[41] Raad Bahmani et al. "{CURE}: A Security Architecture with CUstomizable and Resilient Enclaves". In: *Proceedings of the 2021 USENIX Security Symposium*. 2021.

[42] Xiaolong Bai. *The last line of defense: understanding and attacking Apple File System on iOS*. 2018. URL: https://i.blackhat.com/eu-18/Thu-Dec-6/eu-18-Bai-The-Last-Line-Of-Defense-Understanding-And-Attacking-Apple-File-System-On-IOS.pdf (visited on 03/06/2022).

[43] H. Bar-El et al. "The Sorcerer's Apprentice Guide to Fault Attacks". In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382. DOI: 10.1109/JPROC.2005.862424.

[44] Hagai Bar-El. *Security implications of hardware vs Software cryptographic modules*. Tech. rep. Discretix Technologies, Jan. 2002.

[45] Elaine Barker and John Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. National Institute of Standards and Technology, 2015. DOI: 10.6028/NIST.SP.800-90Ar1.

[46] Frédéric Basse. *Amlogic S905 SoC: bypassing the (not so) Secure Boot to dump the BootROM*. 2016. URL: https://fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html (visited on 02/27/2022).

[47] Andrew Baumann. "Hardware is the New Software". In: *Proceedings of the 2017 Workshop on Hot Topics in Operating Systems (HotOS)*. Whistler, BC, Canada: ACM, 2017, pp. 132–137. ISBN: 978-1-4503-5068-6. DOI: 10.1145/3102980.3103002. URL: http://doi.acm.org/10.1145/3102980.3103002.

[48] Sean Beaupre. *TRUSTNONE*. http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf. 2015.

[49] Ian Beer and Samuel Groß. *A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution*. Dec. 15, 2021. URL: https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html.

[50] Bellom, Maxime Rossi and Melotti, Damiano and Teuwen, Philippe. *A Titan M Odyssey*. 2021. URL: https://i.blackhat.com/EU-21/Wednesday/EU-21-Rossi-Bellom-2021_A_Titan_M_Odyssey-wp.pdf (visited on 03/06/2022).

[51] Gal Beniamini. *QSEE privilege escalation vulnerability and exploit (CVE-2015-6639)*. 2016. URL: https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html (visited on 03/01/2022).

[52] Gal Beniamini. *Trust Issues: Exploiting TrustZone TEEs*. 2017. URL: https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html (visited on 03/01/2022).

[53] Gal Beniamini. *TrustZone Kernel Privilege Escalation (CVE-2016-2431)*. 2016. URL: http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html (visited on 03/01/2022).

[54] David Berard. *Kinibi TEE: Trusted Application Exploitation*. 2018. URL: https://www.synacktiv.com/posts/exploit/kinibi-tee-trusted-application-exploitation.html (visited on 03/01/2022).

[55] David J. Bernstein. *Cache-timing attacks on AES*. 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[56] Kristof Beyls. *[Llvm-Dev] Round Table on AArch64 Pauth ABI - Minutes*. E-mail. Oct. 15, 2020. URL: http://lists.llvm.org/pipermail/llvm-dev/2020-October/145839.html (visited on 10/27/2020).

[57] Andrea Biondo et al. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1213–1227. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/biondo.

[58] Dionysus Blazakis. *The Apple Sandbox*. 2011. URL: https://developer.android.com/guide/topics/permissions/overview (visited on 02/01/2021).

[59] Tyler Bletsch et al. "Jump-Oriented Programming: A New Class of Code-Reuse Attack". In: *Proceedings of the 2011 ACM Asia Conference on Information, Computer and Communications Security (ASIACCS)*. Hong Kong, China: ACM, 2011, pp. 30–40. ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966919.

[60] David G Boak. *A History of U.S. Communications Security*. Vol. 1–2. National Security Agency, 1973.

[61] Carsten Bock et al. "RIP-RH: Preventing Rowhammer-Based Inter-Process Attacks". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (ASIACCS)*. Asia CCS '19. Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 561–572. ISBN: 9781450367523. DOI: 10.1145/3321705.3329827.

[62] Pietro Borrello et al. "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Apr. 21, 2021. DOI: 10.1145/3460120.3484583.

[63] Thomas Bourgeat et al. "MI6: Secure Enclaves in a Speculative Out-of-Order Processor". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Columbus, OH, USA, 2019, pp. 42–56. DOI: 10.1145/3352460.3358310.

[64] C. Bozzato, R. Focardi, and Francesco Palmarini. "Shaping the Glitch: Optimizing Voltage Fault Injection Attacks". In: *IACR Transactions on Cryptographic Hardware Embed. Syst.* 2019.2 (2019), pp. 199–224. DOI: 10.13154/tches.v2019.i2.199-224.

[65] Ferdinand Brasser et al. "CAn't Touch This: Software-Only Mitigation against Rowhammer Attacks Targeting Kernel Memory". In: *Proceedings of the 2017 USENIX Conference on Security Symposium*. Vancouver, BC, Canada: USENIX Association, 2017, pp. 117–130. ISBN: 9781931971409.

[66] Ferdinand Brasser et al. "SANCTUARY: ARMing TrustZone with Userspace Enclaves". In: *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*. S, Feb. 2019.

[67] Ferdinand Brasser et al. "TyTAN: Tiny Trust Anchor for Tiny Devices". In: *Proceedings of the 2015 Annual Design Automation Conference (DAC)*. San Francisco, CA, USA, 2015, 34:1–34:6. DOI: 10.1145/2744769.2744922.

[68] Ernie Brickell and Jiangtao Li. *Enhanced Privacy ID from Bilinear Pairing*. Cryptology ePrint Archive, Report 2009/095. https://eprint.iacr.org/2009/095. 2009.

[69] BSI. *Evaluation of random number generators*. Standard. BSI, 2013.

[70] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *Proceedings of the 2018 USENIX Security Symposium*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 991–1008. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[71] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[72] Nicolas Carlini et al. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity". In: *Proceedings of the 2015 USENIX Security Symposium*. Washington, DC, USA: USENIX Association, Aug. 2015, pp. 161–176. ISBN: 978-1-931971-23-2. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

[73] Pierre Carru. "Attack TrustZone with Rowhammer". Presented at GreHack. 2017. URL: https://grehack.fr/data/2017/slides/GreHack17_Attack_TrustZone_with_Rowhammer.pdf.

[74] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. "Hardware Support for Fast Capability-Based Addressing". In: *ACM SIGOPS Operating Systems Review* 28.5 (Nov. 1, 1994), pp. 319–327. DOI: 10.1145/381792.195579.

[75] David Cerdeira et al. "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems". In: *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P)*. San Francisco, CA, USA, 2020. DOI: 10.1109/SP40000.2020.00061.

[76] Charles Garcia-Tobin, ARM Ltd. *ARM CCA Hardware Architecture*. 2021. URL: https://static.linaro.org/connect/armcca/presentations/CCATechEvent-210623-CGT-2.pdf (visited on 03/06/2022).

[77] Luke Cheeseman. *D51429 [AArch64] Return Address Signing B Key Support*. Sept. 2019. URL: https://reviews.llvm.org/D51429 (visited on 10/26/2020).

[78] Lily Chen, Joshua Franklin, and Andrew Regenscheid. *Guidelines on HardwareRooted Security in Mobile Devices*. SP 800-16. Gaithersburg, MD, United States, 2012.

[79] Nick Chen. "The Benefits Of Antifuse OTP". In: *Semiconductor Engineering* (Dec. 19, 2016). URL: https://semiengineering.com/the-benefits-of-antifuse-otp/ (visited on 03/07/2022).

[80] Long Cheng et al. "Exploitation Techniques and Defenses for Data-Oriented Attacks". In: *Proceedings of the 2019 IEEE Cybersecurity Development (SecDev)*. Tysons Corner, VA, USA, 2019, pp. 114–128. DOI: 10.1109/SecDev.2019.00022.

[81] Clang team. *Hardware-Assisted AddressSanitizer Design Documentation*. 2020. URL: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html (visited on 09/06/2020).

[82] Thomas H Cormen et al. *Introduction to Algorithms*. 3rd. MIT Press, 2009. ISBN: 9780262533058.

[83] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation". In: *Proceedings of the 2016 USENIX Security Symposium*. Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[84] Crispin Cowan et al. "Protecting Systems from Stack Smashing Attacks with StackGuard". In: *Linux Expo*. 1999.

[85] Cryptomathic. *EMV Key Management – Explained*. 2017. URL: https://www.cryptomathic.com/hubfs/docs/cryptomathic_white_paper-emv_key_management.pdf.

[86] Don Davis, Ross Ihaka, and Philip Fenstermacher. "Cryptographic Randomness from Air Turbulence in Disk Drives". In: *Proceedings of Advances in Cryptology — CRYPTO '94*. Ed. by Yvo G. Desmedt. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 114–120. ISBN: 978-3-540-48658-9.

[87] Remi Denis-Courmont et al. "Camouflage: Hardware-Assisted CFI for the ARM Linux Kernel". en. In: *Proceedings of the 2020 ACM/IEEE Annual Design Automation Conference (DAC)* (2020).

[88] Jack B Dennis. "Segmentation and the design of multiprogrammed computer systems". In: *Journal of the ACM (JACM)* 12.4 (1965), pp. 589–602.

[89] Alexander W. Dent. *Secure Boot and Image Authentication*. SP 800-16. Aug. 2012.

[90] Alexandra Dmitrienko et al. "Market-driven code provisioning to mobile secure hardware". In: *Proceedings of the 2015 International Conference on Financial Cryptography and Data Security (FC)*. Springer. 2015, pp. 387–404.

[91] John R. Douceur. "The Sybil Attack". In: *Proceedings the 2002 International Workshop on Peer-to-Peer Systems (IPTPS)*. Ed. by Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron. Vol. 2429. Lecture Notes in Computer Science. Cambridge, MA, USA: Springer, 2002, pp. 251–260. DOI: 10.1007/3-540-45748-8_24.

[92] Wim van Eck. "Electromagnetic radiation from video display units: An eavesdropping risk?" In: *Computers & Security* 4.4 (1985), pp. 269–286. DOI: 10.1016/0167-4048(85)90046-x.

[93] J.-E. Ekberg, K. Kostiainen, and N. Asokan. "The Untapped Potential of Trusted Execution Environments on Mobile Devices". In: *IEEE Security & Privacy Magazine* 12.4 (July 2014), pp. 29–37. ISSN: 1540-7993. DOI: 10.1109/MSP.2014.38.

[94] Jan-Erik Ekberg. "Securing Software Architectures for Trusted Processor Environments; Programvarusystem för säkra processorarkitekturer". en. Ph.D Thesis. 2013, 91 + app. 139. ISBN: 978-952-60-3632-8. URL: http://urn.fi/URN:ISBN:978-952-60-3632-8.

[95] Jan-Erik Ekberg and N Asokan. "External authenticated non-volatile memory with lifecycle management for state protection in trusted computing". In: *Proceedings of the 2009 International Conference on Trusted Systems (INTRUST)*. Springer. 2009, pp. 16–38.

[96] Jan-Erik Ekberg and N. Asokan. "External Authenticated Non-volatile Memory with Lifecycle Management for State Protection in Trusted Computing". In: *Proceedings of the 2010 International Conference on Trusted Systems (INTRUST)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 16–38. DOI: 10.1007/978-3-642-14597-1_2.

[97] Karim Eldefrawy et al. "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust". In: *Proceedings of the 2012 Annual Network and Distributed System Security Symposium (NDSS)*. S, Feb. 2012. URL: http://www.eurecom.fr/publication/3536.

[98] ETSI. *UICC Application Programming Interface for Java Card, Release 11*. https://www.etsi.org/deliver/etsi_ts/102200_102299/102241/11.00.00_60/ts_102241v110000p.pdf. 2012.

[99] Robert S. Fabry. "Dynamic verification of operating system decisions". In: *Communications of the ACM* 16.11 (1973), pp. 659–668.

[100] Nathaniel Wesley Filardo et al. "Cornucopia: Temporal Safety for CHERI Heaps". In: *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA, May 2020.

[101] *Security Requirements for Cryptographic Modules*. NIST, 2019. DOI: 10.6028/NIST.FIPS.140-3.

[102] Rusins Freivalds. "Probabilistic Machines Can Use Less Running Time". In: *Proceedings of the 1977 IFIP Congress*. Toronto, Canada, 1977. ISBN: 0-7204-0755-9.

[103] Mark Gallagher et al. "Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn". In: *Proceedings of the 2019 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: Association for Computing Machinery, Apr. 4, 2019, pp. 469–484. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304037. URL: https://doi.org/10.1145/3297858.3304037 (visited on 02/24/2021).

[104] GCC Team. *GCC 9 Release Series — Changes, New Features, and Fixes*. Aug. 2018. URL: https://gcc.gnu.org/gcc-9/changes.html (visited on 10/26/2020).

[105] GCC Wiki. *Intel® Memory Protection Extensions (Intel® MPX) Support in the GCC Compiler*. June 2018. URL: https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler (visited on 10/28/2020).

[106] General Dynamics Mission Systems. *Advanced INFOSEC Machine (AIM)*. 2015. URL: https://gdmissionsystems.com/-/media/General-Dynamics/Cyber-and-Electronic-Warfare-Systems/PDF/Brochures/cyber-advanced-infosec-machine-aim-datasheet.ashx (visited on 03/06/2022).

[107] General Dynamics Mission Systems. *AIM II — Embeddable Programmable Security)*. https://gdmissionsystems.com/-/media/General-Dynamics/Cyber-and-Electronic-Warfare-Systems/PDF/Brochures/cyber-aim2-embeddable-programmable-security-datasheet.ashx. 2015.

[108] Ronald Gil, Hamed Okhravi, and Howard Shrobe. "There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection". In: *Proceedings of the 2018 IEEE Cybersecurity Development*. SecDev '18. Cambridge, MA, USA, Sept. 2018, pp. 102–109. ISBN: 978-1-5386-7662-2. DOI: 10.1109/SecDev.2018.00021.

[109] Ronald Gil, Hamed Okhravi, and Howard Shrobe. "There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection". In: *Proceedings of the 2018 IEEE Cybersecurity Development*. 2018 IEEE Cybersecurity Development. SecDev '18. Cambridge, MA, USA: IEEE, Sept. 2018, pp. 102–109. ISBN: 978-1-5386-7662-2. DOI: 10.1109/SecDev.2018.00021. URL: https://ieeexplore.ieee.org/document/8543393/ (visited on 03/25/2019).

[110] GlobalPlatform. *Card Specification v2.3.1*. 2018. URL: https://globalplatform.org/specs-library/card-specification-v2-3-1/.

[111] GlobalPlatform. *Root of Trust Definitions and Requirements v1.1*. 2018. URL: https://globalplatform.org/specs-library/globalplatform-root-of-trust-definitions-and-requirements/.

[112] GlobalPlatform. *Secure Channel Protocol '03' - Amendment D v1.1.2*. 2019. URL: https://globalplatform.org/specs-library/secure-channel-protocol-03-amendment-d-v1-1-2/.

[113] GlobalPlatform. *TEE Client API Specification, Version 1.0*. https://globalplatform.org/specs-library/tee-client-api-specification/. 2010.

[114] GlobalPlatform. *TEE Internal Core API Specification, Version 1.3.1*. 2021. URL: https://globalplatform.org/specs-library/tee-internal-core-api-specification/.

[115] GlobalPlatform. *TEE Management Framework, Version 1.0*. 2016. URL: https://globalplatform.org/specs-library/tee-management-framework-including-asn1-profile/.

[116] GlobalPlatform. *TEE Management Framework: Open Trust Protocol (OTrP) Profile v1.0*. GPD_SPE_123. 2019. URL: https://globalplatform.org/specs-library/tee-management-framework-open-trust-protocol.

[117] GlobalPlatform. *TEE Secure ElementAPI v1.1.2*. 2021. URL: https://globalplatform.org/specs-library/tee-secure-element-api/.

[118] GlobalPlatform. *TEE Sockets API Specification v1.0.1, 1.0.2 & 1.0.3*. 2017. URL: https://globalplatform.org/specs-library/tee-sockets-api-specification/.

[119] GlobalPlatform. *TEE System Architecture, Version 1.2*. 2018. URL: https://globalplatform.org/specs-library/tee-system-architecture-v1-2/.

[120] GlobalPlatform. *TEE Trusted User Interface API v1.0*. 2013. URL: https://globalplatform.org/specs-library/trusted-user-interface-api-v1/.

[121] GlobalPlatform. *TEE Trusted User Interface Lowe-level API v1.0.1*. 2018. URL: https://globalplatform.org/specs-library/globalplatform-technology-tee-trusted-user-interface-low-level-api-v1-0-1/.

[122] GNU. *GCC 8.4 Manual*. 2018. URL: https://gcc.gnu.org/onlinedocs/gcc-8.4.0/gcc (visited on 11/10/2020).

[123] Google. *Android permissions*. 2020. URL: https://developer.android.com/guide/topics/permissions/overview (visited on 02/01/2021).

[124] Google. *DM-verity*. 2020. URL: https://source.android.com/security/verifiedboot/dm-verity (visited on 03/06/2022).

[125] Google. *FS-verity*. 2020. URL: https://www.kernel.org/doc/html/latest/filesystems/fsverity.html (visited on 03/06/2022).

[126] Google. *Kotlin and Android*. 2020. URL: https://developer.android.com/kotlin (visited on 10/25/2020).

[127] Google. *Pixel Security: Better, Faster, Stronger*. 2016. URL: https://blog.google/products/android-enterprise/pixel-security-better-faster-stronger/ (visited on 03/06/2022).

[128] Google. *Security-Enhanced Linux in Android*. 2021. URL: https://source.android.com/security/selinux (visited on 02/01/2021).

[129] Robert M Graham. "Protection in an information processing utility". In: *Communications of the ACM* 11.5 (1968), pp. 365–369.

[130] GSMA. *Generic Overlay SIM Security Assessment?* https://www.gsma.com/publicpolicy/wp-content/uploads/2014/08/GSMA-Security-Group-Overlay_SIM_Security_Assessment_August_18_2014.pdf. 2014.

[131] GSMA. *IMEI Blacklisting*. Apr. 22, 2019. URL: https://www.gsma.com/security/resources/imei-blacklisting/.

[132] GSMA. *Understanding SIM Evolution*. 2015. URL: https://www.gsmaintelligence.com/research/?file=81d866ecda8b80aa4642e06b877ec265 (visited on 03/06/2022).

[133] Shay Gueron. *A Memory Encryption Engine Suitable for General Purpose Processors*. Cryptology ePrint Archive, Report 2016/204. https://eprint.iacr.org/2016/204. 2016.

[134] J. Alex Halderman et al. "Lest We Remember: Cold-boot Attacks on Encryption Keys". In: *Communications of the ACM* 52.5 (May 2009), pp. 91–98. DOI: 10.1145/1506409.1506429.

[135] Shai Hasarfaty and Yanai Moyal. *Behind the Scenes of Intel Security and Manageability Engine*. 2019. URL: https://i.blackhat.com/USA-19/Wednesday/us-19-Hasarfaty-Behind-The-Scenes-Of-Intel-Security-And-Manageability-Engine.pdf (visited on 03/06/2022).

[136] W.t. Holman, J.a. Connelly, and A.b. Dowlatabadi. "An integrated analog/digital random noise source". In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 44.6 (1997), pp. 521–528. DOI: 10.1109/81.586025.

[137] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. "IBM System/38 Support for Capability-Based Addressing". In: *Proceedings of the 1981 Annual Symposium on Computer Architecture (ISCA)*. Washington DC, USA: IEEE Computer Society Press, May 12, 1981, pp. 341–348. DOI: 10.5555/800052.801885.

[138] Hong Hu et al. "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks". In: *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA, 2016, pp. 969–986. DOI: 10.1109/SP.2016.62.

[139] Huawei. *Huawei EMUI Security Whitepaper*. 2020. URL: https://consumer-img.huawei.com/content/dam/huawei-cbg-site/common/campaign/privacy/whitepaper/emui-10-security-technical-white-paper-v1.pdf (visited on 03/06/2022).

[140] IBM. *3845/3846 Data Encryption Devices*. 1977. URL: http://ed-thelen.org/comp-hist/IBM-ProdAnn/3845.pdf (visited on 03/06/2022).

[141] Intel. *Control-Flow Enforcement Technology Specification (Revision 3.0)*. May 2019, p. 358. URL: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf (visited on 11/09/2020).

[142] Intel Corporation. *Intel Atom Processor Z2760 Datasheet*. 2012. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/atom-z2760-datasheet.pdf (visited on 03/06/2022).

[143] Intel Corporation. *Proof of Elapsed Time*. 2019. URL: https://github.com/hyperledger/sawtooth-poet (visited on 03/06/2022).

[144] ISO. *Information technology — Security techniques — Random bit generation*. Standard. ISO, 2011.

[145] Markus Jakobsson and Ari Juels. "Proofs of Work and Bread Pudding Protocols". In: *Proceedings of the 2019 IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS)*. Ed. by Bart Preneel. Vol. 152. Leuven, Belgium: Kluwer, 1999, pp. 258–272.

[146] Markus Jakobsson et al. "Implicit authentication for mobile devices". In: *Proceedings of the 2009 USENIX Conference on Hot topics in Security*. Vol. 1. USENIX Association. 2009, pp. 25–27.

[147] Insu Jang et al. "Heterogeneous Isolated Execution for Commodity GPUs". In: *Proceedings of the 2019 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Providence, RI, USA: ACM, 2019, pp. 455–468. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304021.

[148] Yoongu Kim et al. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *Proceeding of the 2014 Annual International Symposium on Computer Architecuture (ISCA)*. 2014.

[149] P. Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*. May 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[150] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Proceedings of 1999 Advances in Cryptology (CRYPTO)*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[151] Paul Kocher et al. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering* 1 (Apr. 2011), pp. 5–27. DOI: 10.1007/s13389-011-0006-y.

[152] Patrick Koeberl et al. "TrustLite: A Security Architecture for Tiny Embedded Devices". In: *Proceedings of the 2014 European Conference on Computer Systems (EuroSys)*. Amsterdam, The Netherlands: ACM, 2014, 10:1–10:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592824.

[153] François Koeune and François-Xavier Standaert. "A Tutorial on Physical Security and Side-Channel Attacks". In: *Foundations of Security Analysis and Design III: FOSAD 2004/2005 Tutorial Lectures*. Ed. by Alessandro Aldini, Roberto Gorrieri, and Fabio Martinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 78–108. ISBN: 978-3-540-31936-8. DOI: 10.1007/11554578_3. URL: https://doi.org/10.1007/11554578_3.

[154] Oliver Kömmerling and Markus G Kuhn. "Design Principles for Tamper-Resistant Smartcard Processors". In: *Proceedings of the 1999 USENIX Workshop on Smartcard Technology*. 1999. URL: https://www.usenix.org/conference/usenix-workshop-smartcard-technology/design-principles-tamper-resistant-smartcard.

[155] Koen Koning et al. "No Need to Hide: Protecting Safe Regions on Commodity Hardware". In: *Proceedings of the 2017 European Conference on Computer Systems*. Belgrade, Serbia: ACM, 2017, pp. 437–452. ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064217.

[156] Iggy Krajci and Darren Cummings. "The Intel Mobile Processor". In: *Android on x86: An Introduction to Optimizing for Intel® Architecture*. Berkeley, CA: Apress, 2013, pp. 33–46. ISBN: 978-1-4302-6131-5. DOI: 10.1007/978-1-4302-6131-5_5. URL: https://doi.org/10.1007/978-1-4302-6131-5_5.

[157] Markus G Kuhn and Ross J Anderson. "Soft tempest: Hidden data transmission using electromagnetic emanations". In: *Proceedings of the 1998 International Workshop on Information Hiding*. Springer. 1998, pp. 124–142.

[158] Dmitrii Kuvaiskii et al. "SGXBOUNDS: Memory Safety for Shielded Execution". In: *Proceedings of the 2017 European Conference on Computer Systems (EuroSys)*. Belgrade, Serbia: ACM, 2017, pp. 205–221. ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064192.

[159] C. A. Lakos. "Implementing BCPL on the Burroughs B6700". In: *Software: Practice and Experience* 10.8 (1980), pp. 673–683. ISSN: 1097-024X. DOI: 10.1002/spe.4380100806.

[160] Butler W Lampson. "Protection". In: *ACM SIGOPS Operating Systems Review* 8.1 (1974), pp. 18–24.

[161] Nikolaus Lange. "Single-chip implementation of a cryptosystem for financial applications". In: *Financial Cryptography*. Ed. by Rafael Hirschfeld. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 135–144. ISBN: 978-3-540-69607-0.

[162] Michael Larabel. *Intel Confirms CET Security Support For Tiger Lake*. July 15, 2020. URL: https://www.phoronix.com/scan.php?page=news_item&px=Intel-CET-Tiger-Lake (visited on 11/15/2020).

[163] Michael Larabel. *Intel MPX Support Is Dead With Linux 5.6*. Jan. 31, 2020. URL: https://www.phoronix.com/scan.php?page=news_item&px=Intel-MPX-Is-Dead (visited on 11/15/2020).

[164] Adam Laurie et al. *rompar: Masked ROM optical data extraction tool*. 2013. URL: https://github.com/AdamLaurie/rompar (visited on 08/12/2020).

[165] Dayeol Lee et al. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *Proceedings of the 2020 European Conference on Computer Systems (EuroSys)*. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387532.

[166] Hans Liljestrand et al. "PAC It up: Towards Pointer Integrity Using ARM Pointer Authentication". en. In: *Proceedings of the 2019 USENIX Security Symposium*. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 177–194. ISBN: 978-1-939133-06-9.

[167] Jin Lin. *Developoer Guidance for Hardware-enforced Stack Protection — Microsoft Tech Community*. Feb. 24, 2021. URL: https://techcommunity.microsoft.com/t5/windows-kernel-internals-blog/developer-guidance-for-hardware-enforced-stack-protection/ba-p/2163340 (visited on 01/16/2022).

[168] Moritz Lipp et al. "ARMageddon: Cache Attacks on Mobile Devices". In: *Proceedings of the 2016 USENIX Security Symposium*. 25th USENIX Security Symposium. Austin, TX, USA, 2016, pp. 549–564. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp (visited on 03/06/2022).

[169] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *Proceedings of the 2018 USENIX Conference on Security Symposium*. Baltimore, MD, USA: USENIX Association, 2018, pp. 973–990. ISBN: 978-1-931971-46-1.

[170] LLVM. *Scudo Hardened Allocator*. 2020. URL: `https://llvm.org/docs/ScudoHardenedAllocator.html` (visited on 10/27/2020).

[171] Peter Loscocco and Stephen Smalley. "Integrating Flexible Support for Security Policies into the Linux Operating System." In: *Proceedings of the 2001 USENIX Annual Technical Conference*. 2001, pp. 29–42.

[172] Aravind Machiry et al. "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments". In: *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. S, Feb. 2017. DOI: 10.14722/ndss.2017.23227.

[173] Tarjei Mandt, Mathew Solni, and David Wang. *Demystifying the Secure Enclave Processor*. 2016. URL: `http://mista.nu/research/sep-paper.pdf` (visited on 03/06/2022).

[174] Tarjei Mandt, Mathew Solni, and David Wang. *Demystifying the Secure Enclave Processor*. 2016. URL: `https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf` (visited on 03/06/2022).

[175] Kristina Martsenko. *Arm64: Compile the Kernel with Ptrauth Return Address Signing*. Mar. 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=74afda4016a7437e6e425c3370e4b93b47be8ddf` (visited on 10/26/2020).

[176] Ali Jose Mashtizadeh et al. "CCFI: Cryptographically Enforced Control Flow Integrity". en. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Denver, CO, USA: ACM Press, 2015, pp. 941–951. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813676.

[177] Daniel Maslowski. "Look at ME! Intel ME Firmware Investigation". FOSDEM 2020. 2020. URL: `https://archive.fosdem.org/2020/schedule/event/firmware_lam/attachments/slides/3872/export/events/attachments/firmware_lam/slides/3872/look_at_me_fosdem20.pdf` (visited on 03/07/2022).

[178] Saara Matala, Thomas Nyman, and N. Asokan. *Historical insight into the development of Mobile TEEs*. 2019. URL: `https://blog.ssg.aalto.fi/2019/06/historical-insight-into-development-of.html` (visited on 03/06/2022).

[179] Isaac Mbiti and David N. Weil. *Mobile banking: The impact of MPesa in Kenya*. Working Paper 011-13. Brown University, Department of Economics, 2011. URL: `http://hdl.handle.net/10419/62662`.

[180] Jonathan M McCune et al. "Flicker: An Execution Infrastructure for TCB Minimization". In: *Proceedings the 2008 ACM SIGOPS/EyroSys European Conference on Computer Systems*. Glasgow, UK, 2008, pp. 315–328. DOI: 10.1145/3246965.

[181] Frank McKeen et al. "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2013 International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. Tel-Aviv, Israel: Association for Computing Machinery, 2013. ISBN: 9781450321181. DOI: 10.1145/2487726.2488368. URL: https://doi.org/10.1145/2487726.2488368.

[182] Frank McKeen et al. "Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave". In: *Proceedings of the 2016 International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. HASP 2016. Seoul, Republic of Korea: ACM, 2016, 10:1–10:9. ISBN: 978-1-4503-4769-3. DOI: 10.1145/2948618.2954331.

[183] Michael McReynolds. *Azure announces next generation Intel SGX confidential computing VMs*. 2021. URL: https://techcommunity.microsoft.com/t5/azure-confidential-computing/azure-announces-next-generation-intel-sgx-confidential-computing/ba-p/2839934 (visited on 01/29/2021).

[184] Masoud Mehrabi Koushki et al. "Is Implicit Authentication on Smartphones Really Popular? On Android Users' Perception of "Smart Lock for Android"". In: *Proceedings of the 2020 International Conference on Human-Computer Interaction with Mobile Devices and Services*. 2020, pp. 1–17.

[185] Microsoft. *A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*. 2006. URL: https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in (visited on 09/05/2019).

[186] Markus Miettinen et al. "Conxsense: automated context classification for context-aware access control". In: *Proceedings of the 9th ACM Asia Symposium on Information, Computer and Communications security (ASI-ACCS)*. 2014, pp. 293–304. DOI: 10.1145/2590296.2590337.

[187] Marvin Minsky. "Memoir on Inventing the Confocal Scanning Microscope". In: *Scanning* 10 (1988), pp. 128–138.

[188] Nagendra Modadugu and Bill Richardson. *Building a Titan: Better security through a tiny chip*. 2018. URL: https://security.googleblog.com/2018/10/building-titan-better-security-through.html (visited on 03/06/2022).

[189] Vishwath Mohan et al. "Opaque Control-Flow Integrity". In: *Proceedings of the 2015 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2015. ISBN: 978-1-891562-38-9. DOI: 10.14722/ndss.2015.23271. URL: https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/opaque-control-flow-integrity/ (visited on 10/07/2019).

[190] Mondato. *Skin SIM Technology: A Serious Challenge for Safaricom?* 2014. URL: https://blog.mondato.com/skin-sim-safari/ (visited on 03/06/2022).

[191] James Morris, Stephen Smalley, and Greg Kroah-Hartman. "Linux Security Modules: General security support for the Linux kernel". In: *Proceedings of the 2002 USENIX Security Symposium*. ACM Berkeley, CA. 2002, pp. 17–31.

[192] Onur Mutlu et al. "A Modern Primer on Processing in Memory". In: *arXiv preprint arXiv:2012.03112* (2020).

[193] Santosh Nagarakatte et al. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Dublin, Ireland: ACM, 2009, pp. 245–258. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542504.

[194] National Geographic. *First computer bug*. 1947. URL: https://www.nationalgeographic.org/thisday/sep9/worlds-first-computer-bug/.

[195] Dana Neustadter. *True Random Number Generators for Heightened Security in Any SoC*. 2020. URL: https://www.synopsys.com/designware-ip/technical-bulletin/true-random-number-generator-security-2019q3.html.

[196] Phong Q. Nguyen and Igor E. Shparlinski. "The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces". In: *Designs, Codes and Cryptography* 30.2 (Sept. 2003), pp. 201–217. DOI: 10.1023/A:1025436905711.

[197] Nguyen and Shparlinski. "The Insecurity of the Digital Signature Algorithm with Partially Known Nonces". In: *Journal of Cryptology* 15.3 (June 2002), pp. 151–176. DOI: 10.1007/s00145-002-0021-3.

[198] NVIDIA Corporation. *Trusted Little Kernel (TLK) for Tegra: FOSS Edition*. 2015. URL: http://nv-tegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git;a=blob_plain;f=documentation/Tegra_BSP_for_Android_TLK_FOSS_Reference.pdf;hb=HEAD.

[199] Thomas Nyman et al. "CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers". In: *Proceedings of the 2017 Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Ed. by Marc Dacier et al. Cham: Springer International Publishing, 2017, pp. 259–284. ISBN: 978-3-319-66332-6. DOI: 10.1007/978-3-319-66332-6_12.

[200] Thomas Nyman et al. "HardScope: Hardening Embedded Systems Against Data-Oriented Attacks". In: *Proceedings of the 2019 Annual Design Automation Conference (DAC)*. Las Vegas, NV, USA: ACM, 2019, p. 63. DOI: 10.1145/3316781.3317836.

[201] Oleksii Oleksenko et al. "Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2.2 (June 2019), 28:1–28:30. ISSN: 2476-1249. DOI: 10.1145/3224423.

[202] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Proceedings of the 2006 CT-RSA*. Red. by David Hutchison et al. Vol. 3860. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20. ISBN: 978-3-540-31033-4 978-3-540-32648-9. DOI: 10.1007/11605805_1. (Visited on 02/27/2022).

[203] Bob Page. *A Report on the Internet Worm*. 1988. URL: https://www.ee.ryerson.ca/~elf/hack/iworm.html (visited on 03/06/2022).

[204] David A Patterson and John L Hennessy. *Computer Organization and Design*. 3rd ed. Morgan Kaufmann, 2005. ISBN: 1-55860-604-1.

[205] Greig Paul and James Irvine. "Take Control of Your PC with UEFI Secure Boot". In: *Linux Journal* 2015.257 (Sept. 2015). ISSN: 1075-3583.

[206] PaX Team. *PaX PAGEEXEC Documentation*. 2006. URL: https://pax.grsecurity.net/docs/pageexec.txt (visited on 11/15/2020).

[207] Siani Pearson. *Trusted Computing Platforms: TCPA Technology in Context*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002. ISBN: 0130092207.

[208] Mingliang Pei et al. *The Open Trust Protocol (OTrP)*. Internet-Draft draft-ietf-teep-opentrustprotocol-03. IETF Secretariat, May 2019. URL: http://www.ietf.org/internet-drafts/draft-ietf-teep-opentrustprotocol-03.txt.

[209] Alexander (Solar Designer) Peslyak. *Getting around Non-Executable Stack (and Fix)*. Aug. 1997. URL: https://seclists.org/bugtraq/1997/Aug/63 (visited on 09/05/2019).

[210] Frank Piessens. "Security across abstraction layers: old and new examples". In: *Proceedings of the 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. Genoa, Italy, 2020. DOI: 10.1109/EuroSPW51379.2020.00043.

[211] Sandro Pinto and Nuno Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM Computing Surveys* 51.6 (Jan. 2019), 130:1–130:36. ISSN: 0360-0300. DOI: 10.1145/3291047.

[212] Marios Pomonis et al. "kR^X: Comprehensive Kernel Protection against Just-in-Time Code Reuse". In: *Proceedings of the 2017 European Conference on Computer Systems (EuroSys)*. Belgrade, Serbia: ACM, 2017, pp. 420–436. ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064216.

[213] Changwoo Pyo and Gyungho Lee. "Encoding Function Pointers and Memory Arrangement Checking against Buffer Overflow Attack". en. In: *Information and Communications Security*. Vol. 2513. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 25–36. ISBN: 978-3-540-00164-5. DOI: 10.1007/3-540-36159-6_3.

[214] Qualcomm. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. 2017.

[215] Inc. Qualcomm Technologies. *Qualcomm Secure Processing Unit SPU230 Core Security Target Lite*. 80-NU430-6 Rev. B. 2019. URL: https://www.commoncriteriaportal.org/files/epfiles/1045b_pdf.pdf.

[216] Inc. Qualcomm Technologies. *Qualcomm SPU FIPS 140-2 Non-Proprietary Security Policy V1.3*. 80-NU430-6 Rev. B. 2019. URL: https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp3549.pdf.

[217] J. Quisquater and David Samyde. "ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards". In: *Proceedings of the 2001 International Conference on Research in Smart Cards*. 2001.

[218] S. Ravi, A. Raghunathan, and S. Chakradhar. "Tamper resistance mechanisms for secure embedded systems". In: *Proceedings of the 2004 International Conference on VLSI Design*. 2004, pp. 605–611. DOI: 10.1109/ICVD.2004.1260985.

[219] Elena Reshetova, Filippo Bonazzi, and N. Asokan. "Randomization Can't Stop BPF JIT Spray". In: *Proceedings of the 2017 International Conference on Network and System Security*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 233–247. ISBN: 978-3-319-64701-2. DOI: 10.1007/978-3-319-64701-2_17.

[220] Elena Reshetova et al. "Toward Linux Kernel Memory Safety". In: *Software: Practice and Experience* 48.12 (Dec. 2018), pp. 2237–2256. ISSN: 00380644. DOI: 10.1002/spe.2638.

[221] Dan Rosenberg. *QSEE Trustzone Kernel Integer Overflow Vulnerability*. Presented at Black Hat 2014. 2014. URL: https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-On-Trusting-TrustZone-WP.pdf.

[222] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. 1st. Berkely, CA, USA: Apress, 2014. ISBN: 9781430265719.

[223] Mark Rutland. *Arm64: Enable Pointer Authentication*. Dec. 2018. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=04ca3204fa09f5f55c8f113b0072004a7b364ff4 (visited on 10/26/2020).

[224] Samsung. *Samsung Knox*. 2020. URL: https://docs.samsungknox.com/admin/whitepaper.

[225] Samsung. *Samsung Knox file encryption*. 2020. URL: https://www.samsungknox.com/en/blog/samsung-knox-file-encryption-1-0-the-first-certified-integrated-dual-data-at-rest-solution-for-mobile-devices (visited on 03/06/2022).

[226] Ravi S Sandhu and Pierangela Samarati. "Access control: principle and practice". In: *IEEE Communications Magazine* 32.9 (1994), pp. 40–48.

[227] Stefan Santesson et al. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. RFC 6960. June 2013. DOI: 10.17487/RFC6960. URL: https://rfc-editor.org/rfc/rfc6960.txt.

[228] Uday Savagaonkar et al. *Titan in depth: Security in plaintext*. 2017. URL: https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext (visited on 03/06/2022).

[229] Michael D. Schroeder and Jerome H. Saltzer. "A Hardware Architecture for Implementing Protection Rings". In: *Proceedings of the 1971 ACM Symposium on Operating Systems Principles (SOSP)*. Palo Alto, CA, USA: ACM, 1971, pp. 42–54. DOI: 10.1145/800212.806498. URL: http://doi.acm.org/10.1145/800212.806498.

[230] Abu Sebastian et al. "Memory devices and applications for in-memory computing". In: *Nature Nanotechnology* 15.7 (2020), pp. 529–544. DOI: 10.1038/s41565-020-0655-z.

[231] ARM Ltd. *Security IP*. 2020. URL: https://developer.arm.com/ip-products/security-ip (visited on 10/22/2020).

[232] Arm Ltd. *Security IP | CryptoCell-300 family – Arm Developer*. 2020. URL: https://developer.arm.com/ip-products/security-ip/cryptocell-300-family (visited on 10/22/2020).

[233] Arm Ltd. *Security IP | CryptoCell-700 Family – Arm Developer*. 2019. URL: https://developer.arm.com/ip-products/security-ip/cryptocell-700-family (visited on 12/01/2019).

[234] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. Boston, MA, USA: USENIX, 2012, pp. 309–318. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[235] Kostya Serebryany. "ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety". In: *USENIX ;login:* 44.2 (2019), pp. 12–16.

[236] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)". In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS)*. Alexandria, Virginia, USA: ACM, 2007, pp. 552–561. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313. URL: http://doi.acm.org/10.1145/1315245.1315313.

[237] Di Shen. *Attacking your "Trusted Core" Exploiting TrustZone on Android*. Presented at Black Hat 2015. 2015. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android.pdf.

[238] SIMalliance Ltd. *Device Implementation Guidelines version 1.1*. 2013. URL: https://simalliance.org/wp-content/uploads/2015/03/SIMalliance_UICC_Device_Implementation_Guidelines-1.1.pdf.

[239] Sergei P. Skorobogatov and Ross J. Anderson. "Optical Fault Induction Attacks". In: *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–12. DOI: 10.1007/3-540-36400-5_2.

[240] Asia Slowinska and Herbert Bos. "Pointless Tainting? Evaluating the Practicality of Pointer Tainting". In: *Proceedings of the 2009 ACM European Conference on Computer Systems (EuroSys)*. New York, NY, USA: Association for Computing Machinery, Apr. 1, 2009, pp. 61–74. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519073.

[241] Stephen Smalley and Robert Craig. "Security Enhanced (SE) Android: Bringing Flexible MAC to Android." In: *Proceedings of the 2013 Network and Distributed Systems Symposium (NDSS)*. Vol. 310. 2013, pp. 20–38.

[242] K. Z. Snow et al. "Just-in-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. SP '13. San Francisco, CA, USA, May 2013, pp. 574–588. DOI: 10.1109/SP.2013.45.

[243] Eugene H. Spafford. "The Internet Worm Program: An Analysis". In: *ACM SIGCOMM Computer Communication Review*. SIGCOMM 19.1 (Jan. 1989), pp. 17–57. ISSN: 0146-4833. DOI: 10.1145/66093.66095. URL: http://doi.acm.org/10.1145/66093.66095.

[244] Evgenii Stepanov et al. "Memory Tagging in LLVM and Android". 2020 Virtual LLVM Developers' Meeting. Oct. 2020.

[245] Nigel Stephens. *Developments in the Arm A-Profile Architecture: Armv8.6-A*. Sept. 25, 2019. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-architecture-developments-armv8-6-a (visited on 04/20/2020).

[246] G. Edward Suh et al. "AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing". In: *Proceedings of the 2003 Annual International Conference on Supercomputing (ICS)*. San Francisco, CA, USA: ACM, 2003, pp. 160–171. ISBN: 1-58113-733-8. DOI: 10.1145/782814.782838.

[247] G. Edward Suh et al. "Secure Program Execution via Dynamic Information Flow Tracking". In: *ACM SIGPLAN Notices* 39.11 (Oct. 7, 2004), pp. 85–96. ISSN: 0362-1340. DOI: 10.1145/1037187.1024404. (Visited on 10/25/2021).

[248] Harini Sundaresan. *OMAP platform security features*. 2003. URL: https://www.ti.com/pdfs/wtbu/omapplatformsecuritywp.pdf (visited on 03/07/2022).

[249] Synopsys Inc. *Heartbleed Bug*. 2014. URL: https://heartbleed.com/ (visited on 06/21/2021).

[250] Laszlo Szekeres et al. "SoK: Eternal War in Memory". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.13.

[251] Ady Tal. *Using Intel® MPX with the Intel® Software Development Emulator*. July 23, 2020. URL: https://www.intel.com/content/www/us/en/develop/articles/using-intel-mpx-with-the-intel-software-development-emulator.html (visited on 11/15/2020).

[252] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *Proceedings of the 2017 USENIX Security Symposium*. Vancouver, BC: USENIX Association, 2017, pp. 1057–1074. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[253] Bill Toulas. *New Intel chips won't play Blu-ray disks due to SGX deprecation*. 2021. URL: https://www.bleepingcomputer.com/news/security/new-intel-chips-wont-play-blu-ray-disks-due-to-sgx-deprecation/ (visited on 01/29/2021).

[254] Florian Tramer and Dan Boneh. "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware". In: *Proceedings of the 2019 International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJVorjCcKQ.

[255] Meltem Sönmez Turan et al. *Recommendation for the Entropy Sources Used for Random Bit Generation*. National Institute of Standards and Technology, 2018. DOI: 10.6028/NIST.SP.800-90B.

[256] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*. 2020.

[257] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. "SoK: Fully Homomorphic Encryption Compilers". In: *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, pp. 1092–1108. DOI: 10.1109/SP40001.2021.00068.

[258] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. "Graviton: Trusted Execution Environments on GPUs". In: *Proceedings of the 2018 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 681–696. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/volos.

[259] John Von Neumann. "First Draft of a Report on the EDVAC". In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75.

[260] Robert Watson et al. "Design and implementation of the Trusted BSD MAC framework". In: *Proceedings of the 2003 DARPA Information Survivability Conference and Exposition*. Vol. 1. IEEE. 2003, pp. 38–49.

[261] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.html (visited on 11/30/2020).

[262] Samuel Weiser et al. "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V". In: *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*. Feb. 2019. DOI: 10.14722/ndss.2019.23068.

[263] J. Woodruff et al. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *Proceedings of the ACM/IEEE 2014 International Symposium on Computer Architecture (ISCA)*. ACM/IEEE ISCA '14. June 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201. URL: https://doi.org/10.1109/ISCA.2014.6853201 (visited on 12/03/2020).

[264] Peter Wright. *Spycatcher*. Heinemann Publishers Australia, 1987.

[265] Leslie Xu. *Secure the Enterprise with Intel® AES-NI*. White Paper. Intel, 2010. URL: https://www.intel.com/content/www/us/en/enterprise-security/enterprise-security-aes-ni-white-paper.html.

[266] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 2014 USENIX Security Symposium*. 2014.

[267] Yasukazu Yoshizawa et al. "Physical random numbers generated by radioactivity". In: *Journal of the Japanese Society of Computational Statistics* 12.1 (1999), pp. 67–81. DOI: 10.5183/jjscs1988.12.67.

[268] Yu-cheng Yu. *[PATCH v30 00/32] Control-flow Enforcement: Shadow Stack*. Aug. 30, 2021. URL: https://lore.kernel.org/linux-mm/20210830181528.1569-3-yu-cheng.yu@intel.com/T/ (visited on 01/16/2022).

[269] Jiyong Yu et al. "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing". In: *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2019. ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23061. URL: https://doi.org/10.14722/ndss.2019.23061 (visited on 09/23/2021).

[270] Jiyong Yu et al. “Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data”. In: *Proceedings of the 2019 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, Oct. 12, 2019, pp. 954–968. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358274.

[271] Ning Zhang et al. *TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices*. Cryptology ePrint Archive 2016/980. 2016. URL: https://eprint.iacr.org/2016/980 (visited on 03/06/2022).

[272] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. “CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds”. In: *Proceedings of the 2016 International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Paris, France, 2016. DOI: 10.1007/978-3-319-45719-2_6.

[273] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. “Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices”. In: *Proceedings of the 2016 ACM Conference on Computer and Communications Security (ACM)*. New York, NY, USA: Association for Computing Machinery, Oct. 24, 2016, pp. 858–870. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978360.

[274] Qing Zhou et al. “True Random Number Generator Based on Mouse Movement and Chaotic Hash Function”. In: *Information Sciences* 179.19 (Sept. 2009), pp. 3442–3450. ISSN: 0020-0255. DOI: 10.1016/j.ins.2009.06.005.

[275] Jianping Zhu et al. “Enabling Rack-Scale Confidential Computing Using Heterogeneous Trusted Execution Environment”. In: *Proceedings of the 2020 IEEE Symposium on Security and Privacy*. 2020 IEEE Symposium on Security and Privacy (SP). San Francisco, California, USA, May 2020, p. 16. DOI: 10.1109/SP40000.2020.00054.

[276] Jean-Karim Zinzindohoué et al. “HACL*: A Verified Modern Cryptographic Library”. In: *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS)*. 2017. DOI: 10.1145/3133956.3134043.