Trusted Execution Environments

Assaf Rosenbaum

Technion - Computer Science Department - M.Sc. Thesis MSC-2019-03 - 2019

Trusted Execution Environments

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Assaf Rosenbaum

Submitted to the Senate of
the Technion — Israel Institute of Technology
Kislev 5779 Haifa November 2018

This research was carried out under the supervision of Prof. Eli Biham and Dr. Sara Bitan, in the Faculty of Computer Science.

Acknowledgements

First and foremost, I wish to thank my advisors, Prof. Eli Biham and Dr. Sara Bitan, for their incredible support during the work on thesis. Thank you for providing me the freedom to find my own path while constantly guiding and directing me. I learned a lot from both of you.

I would also like to thank my parents, siblings and my parents-in-law for their help and support during this stressful time. Thank you for being there whenever I needed. A special thanks is due to my three lovely children, you sparkle my life with moment of pure joy.

Most importantly, I would like to thank my beautiful wife, Lior. Thank you for your endless support, the long hours in which you worked hard to give me the time I needed to work and for always pushing me forward. You ar the light of my life and I love you very much.

The generous financial help of the Technion, the Hiroshi Fujiwara cyber security research center and the Israel cyber directorate is gratefully acknowledged.

Contents

Hebrew Abstract ה

List of Figures

List of Tables

Listings

Abstract

Our dependence on computer systems is constantly growing. We rely on them for almost every aspect of our lives and and trust them with our most sensitive information, and critical operations. For example, a standard mobile phone may store personal data such as biomteric data or credit card number, as well as private information such as browsing history or medical records.

From an attacker's stand point, the potential profit from a successful attack on these devices is quite considerable. Therefore, malicious players are willing to invest increasing efforts to devise highly sophisticated attacks. These malicious players use various technologies to constantly attack a wide range of targets such as governments, institutes and individuals.

One particularly interesting type of attack is aiming the victim's operating system (OS) kernel. A modern OS kernel is an extremely complex software with millions of lines of code, making it error prone and therefore easier to be exploited by attackers. A successful attack on the OS kernel is devastating. An attacker with kernel privileges may bypass the security policy which is supposed to be enforced by the kernel. Moreover, attackers who succeed to take over the kernel have total control over the system, including all user programs, disk files and I/O devices. Successful attackers can even bypass protection mechanisms like anti-virus or anti-malware kits, which rely on kernel services for their integrity and availability. With these facts in mind, we realized the OS kernel is a high value target for attackers.

However, we observed that the OS kernel lacks some of the protection mechanisms that are used in user space. If the OS is compromised, the victim's system can no longer protect itself. Moreover, the system may not be aware it is compromised.

If we wish to maintain some level of security even in the case the OS is compromised, an isolated trusted component must be added to the system. One such component is called a *Trusted Execution Environment* (TEE). A TEE can be used to protect the OS kernel in the case of an attack, or to store sensitive data and prevent its leakage if the OS is compromised.

Our work focuses on the ways TEEs can improve the security of computer systems. We assess the current status of various TEE implementations and identify weak points in their security design. Based on our assessment we develop *TROOS – Trusted Open Operating System*, a trusted OS for TEE, designed to bridge some of the found flaws. We

also develop *TKRFP – Trusted Return Flow Protection*, which is a security mitigation that utilizes the TEE to eliminate the threats posed by return flow hijacking attacks (such as return oriented programming) on the OS. TKRFP provides another layer of security to the OS, thus it increases the entire system's security. Finally, we introduce the *Context Switch Oriented Programming* attack, which is a new attacking technique that can potentially bypass shadow-stack-based mitigations against return flow hijacking. We also discuss countermeasures against this new attack, including how we protected TKRFP.

Chapter 1

Introduction

The ever growing complexity of computer systems is a stumbling block when considering the system's security. A complex system is prone to bugs, which leads to vulnerabilities that can be exploited by attackers. Needless to say that operating systems are among the most complex software, and that their complexity increases with time. Experience shows that a determined attacker is likely to be able to subvert any modern operating system. This claim is supported by data about critical OS vulnerabilities reported by NVD [39] during 2017 as shown in Figure 1.1. It is easy to see that dozens of critical vulnerabilities were reported for each OS in a period of twelve months.

The problem is that the OS is usually the main component which enforces the system security policy. Therefore, once the OS is compromised the system can no longer effectively protect itself, nor the user processes and data.

It is therefore that we cannot solely trust the OS to enforce the system's security. Thus, a solution that can withstand attacks from a compromised OS is required. Such a solution is required to maintain isolated resources, such as memory location or sensitive operations, out of the OS's reach.

A *Trusted Execution Environment* (TEE) offers this kind of protection. A TEE is an isolated, security oriented and compact environment that can withstand attacks

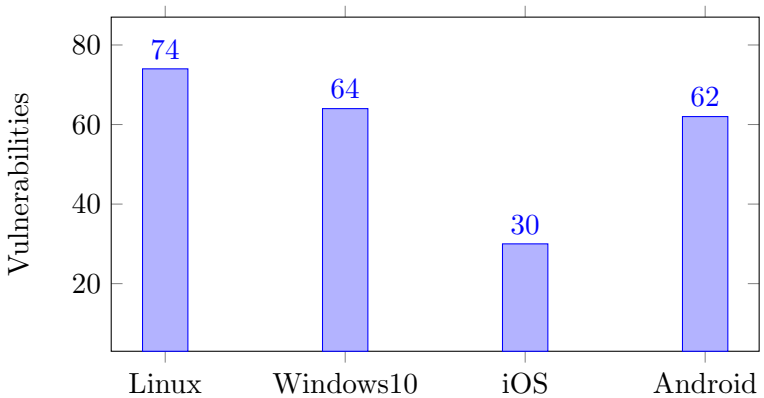


Figure 1.1: Critical Vulnerabilities in Major OSes (Jan17-Dec17)

from a compromised system. A TEE provides security-sensitive services to the main system, and preforms the most sensitive operations.

One of the most widespread TEE technologies is ARM TrustZone. Various TEE instances were implemented based on TrustZone. As a matter of fact, almost every mobile phone sold nowadays is equipped with some variation of a TrustZone-based TEE.

1.1 Our Contributions

In this thesis we focus on TrustZone-based TEEs. We study the security level of these TEEs and offer ways to improve it. We also show how TEE can take an active role in MEE protection and prevent attacks against its OS.

Our contribution can divide to three major parts:

1. We evaluate the security level of three leading TrustZone-based TEEs: QSEE, Kinibi and OP-TEE. This evaluation shows that all of them lacks key security features, such as security mitigation. Moreover, these TEEs API is too broad and not sufficiently controlled. Various attacks on the TEE can be mounted through this broad API. The results of our evaluation appear in Chapter 4.

2. We design TROOS: TRusted Open OS. TROOS is an OS for TrustZone-based TEEs that mitigates the security flaws that we found. The TROOS API consists of only four commands, and is designed in a way that prevents attackers from using it as an attack vector. In addition, TROOS also incorporates security concepts, such as minimal TCB and least privileges, that were not yet utilized in TrustZone-based TEEs. TROOS is described in Chapter 4.

3. We design TKRFP: Trusted Kernel Return Flow Protection. TKRFP is a TEE based security mechanism that protects the MEE OS from return flow hijacking attacks (such as ROP). Unlike existing mitigations, TKRFP offers complete protection even against a relatively powerful attacker who is able to write to arbitrary kernel memory addresses. TKRFP is described in Chapter 5.

4. We present CSOP: Context Switch Oriented Programming attack. CSOP is a new attacking techniques that enables an attacker with kernel write privileges to bypass shadow-stack-based security mitigations (such as TKRFP), and achieve arbitrary code execution by utilizing return flow hijacking attacks. CSOP is described in Chapter 6, along with our proposed countermeasures.

1.2 Theoretical Background

There are many definitions for the security of computer systems. For the purpose of this thesis, a computer system is considered secure if it follows a pre-defined set of security requirements. These requirements are called the *security policy* of the system.

A security policy consists of a collection of *assets* and requirements regarding these assets. The security policy may cover a wide range of requirements regarding various aspects of the system. Note that a system may be secure with respect to one policy but insecure with respect to another security policy, depending on the specific assets and the specific requirements of these policies.

For example, the security policy of a web server typically states its data base as one of the system's assets, and requires restricted access to this data base. On the other hand, some web servers allow access to the general public, and others restrict access to a small set of authorized users. The security policy may also require limitations on physical access to the server, such as access by at least two people simultaneously to the hardware, or that the hardware should be protected by a fireproof safe. Moreover, it may require limitations on the geographic location of the server or the hours in which the data is accessed.

Various mechanisms can be used for security policy enforcement. One of the most important of these mechanisms is *access control*, which enables the system to restrict access to its internal assets. Modern computer systems include various access control features. For example virtual memory restricts memory accesses by processes, the Linux OS defines access right to each file, iOS allows applications to access the camera only after user approval, etc. A secure system usually uses its access control capabilities to enforce a bulk of the security policy requirements.

When evaluating the security of a system we should also consider its adversary. The adversary is trying to attack the system and compromise its assets (i.e., violate the security policy) and might try various techniques in order to do so. We cannot know in advance who is this adversary or what are his capabilities. However, in order to focus our attention to the most significant threats, we can define a *threat model* to generally describe this adversary and his capabilities. We can also analyze our system and identify the interfaces that might be use to attack its assets. The collection of these interfaces is called the system *attack surface*. Different threat models might lead to different attack surfaces. If we assume a certain threat model, in which attacker cannot access a particular interface, this interface may be removed from the attack surface. A different threat model may assume access to the interface which leads to an larger attack surface.

We extend our discussion on these notions in more details in the following subsections.

1.2.1 Access Control

Generally speaking, a computer system consists of resources (objects), and users (subjects) which use the system to interact with these resources. In the common case, some restrictions should be enforced on these interactions. E.g., a certain user may be allowed to interact only with a subset of the resources or to preform only certain operations on them. *Access Control* is the selective restrictions of the ways users interact with the

system resources. Access control can be used to enforce the system security policy. A formal definition follows.

Objects The system resources are called objects. The prime example of an object is a file, but objects may also be processes, memory locations, interprocess messages, network packets, I/O devices, etc.

Subjects Entities that may access and manipulate objects are called subjects. The high abstraction of a subject is a user, but in practice the subjects in a system are usually processes (that may or may not be associated with users). Other kinds of subjects may be I/O controllers, and various kinds of "smart" devices.

All the activities in the system can be viewed as a sequence of accesses preformed by subjects on objects, where each access is a request to preform an operation on the object. For example, a user reading from a file. Access control selectively restricts these accesses according to some policy. Access control consists of three tasks:

1. Identification: uniquely identify each subject and object in the system.
2. Monitoring: determine underlying operation of each access request (read, write, execute, etc.).
3. Authorization: allow or decline each access according to the system's security policy.

We should mention that on a standard system the OS is the main component which enforces access control. In the common case, the subjects are processes (acting on behalf of users). The processes interact with the system's resources via the OS API and the OS enforces access control according to the system's security policy.

1.2.2 Trusted Computing Base

In order to be secure, a system must have good security policy. However, this is not enough, the system should also be able to reliably enforced it. Therefore, the components that enforce the security policy are crucial to the security of the system.

A system's *Trusted Computing Base* (TCB) [33] is the set of all components (hardware, firmware and software) that are critical to its security. Compromising any of the TCB components might jeopardize the entire system security, while compromising a component outside the TCB does not effect the global system security (an isolated part of the system is of course compromised). In other words, the TCB are the components responsible to enforce the security policy, and the ones which may be used to bypass it.

Naturally, a secure system inspires to have the smallest TCB possible. However, the TCB clearly contains the components that enforce the system's access control, as they are an important part of the security policy. As stated above, the components

that enforce access control are typically parts of the OS. Most OSes do not separate the parts which enforce access control from the rest of the OS. Therefore, the entire OS, which is a huge complex piece of software, is typically contained in the system's TCB. Including the OS in the TCB significantly increases the TCB.

1.2.3 Threat Model

The system's *threat model* defines the adversary capabilities. The threat model describes the access our attacker is assumed to have to the system, the attacker's resources, and his abilities. The security evaluation is preformed in the light of the system's threat model.

The threat model is an important tool when trying to evaluate security, as it helps us to focus the attention to the components that might be attacked. However, if not defined carefully, the threat model may lack critical parts. An incomplete threat model might cause us to miss crucial risks the system is exposed to.

For example, lets consider the security evaluation of a web server. The threat model is likely to assume that an attacker is able interact with the server via various network protocols. Thus, attacks via network are considered of a high relevance.

On the other hand, in most cases we can safely assume that the attacker does not have physical access to the server, so threats that require physical access are of lower relevance.

Another usage of the threat model is to define the attack scenarios we expect our system to be protected from, and also those we know we cannot protect against. In the previous example, the threat model suggests that attacks that require physical access to the server are only a secondary concern, and that the majority of resources should be devoted for protection from the network.

1.2.4 Attack Surface

When analyzing the security of a computer system, we need to define what are the resources we want to protect, and what might put these resources at risk. Our threat analysis model is based on a risk management scheme offered by NIST in [60].

To evaluate the system's security we define its *assets*, that are the critical resources which must be protected. Then, we find the system's *entry points*, which are the ways an attacker may interact with the system. Out of these entry points we focus on those who can be used as *attack vectors* to one of the system's assets. The set of these attack vectors are the system's *attack surface*. A formal definition follows.

Assets The *assets* of a system is the set of all objects (resources) that are critical to the system's owner and must be protected.

Entry Point A system's *entry point* is an interface through which inputs can be provided to the system.

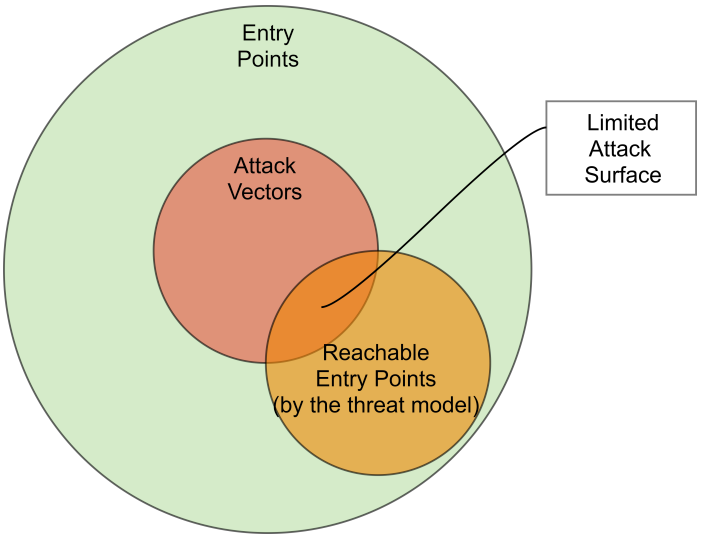


Figure 1.2: Attack Surface Limited to Reachable Entry Points

Attack Vector An *attack vector* is an entry point that might be used to mount an attack on one (or more) of the system's assets.

Attack Surface The system's *attack surface* is the set of all of its attack vectors.

The system's attack surface represents the threats the system is exposed to. When evaluating the security of a computer system, we start by finding its attack surface. Once the attack surface is identified, we can evaluate the system's ability to withstand attacks against its assets.

One of the threat model usages is to help defining the system's attacks surface. A real system can be too complex to be fully analyzed. In such cases, we can use the threat model to focus our attention to the most relevant components of the system's security. For example, a system may have so many entry points that we cannot analyze all of them to determine which of them are attack vectors.

Based on the threat model we can define a subset of the entry points, called the reachable entry points. This subset contains an entry point, if and only if the threat model defines it as reachable by attackers. By restricting our analysis only to reachable entry points (as shown in Figure 1.2), we can reduce our analysis efforts, and define a limited attack surface.

If our threat model is accurate enough, we do not neglect to analyze significant threats by reducing our interest only to the reachable entry points. On the other hand, if our threat model is not sufficient, i.e., an attacker can reach an entry point which we assume is unreachable, then we might ignore an attack vector in our analysis, leaving the system exposed to attacks from this vector.

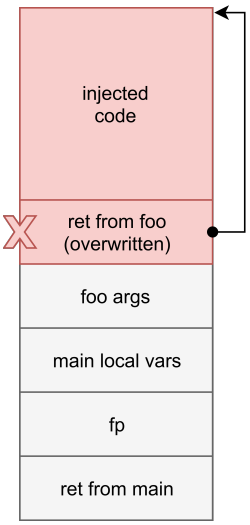


Figure 1.3: Stack Overwrite Attack

1.3 Return Flow Hijacking

Control flow hijacking is a wide family of attacks that aim to take over the victim program. In a control flow hijacking attack, the attacker overwrites data used by the program control flow, thus gaining control over the program execution. Return flow hijacking is a special case of Control flow hijacking. In return flow hijacking the attacker hijacks the program control flow by overwriting some procedure return addresses. Thus, directing the program execution to a location of his choice upon procedure return. Since usually procedure return addresses are stored on the call stack, an attacker that can overwrite the call stack may hijack the program control flow.

The lion's share of critical vulnerabilities are due to buffer overflows, and in particular stack buffer overflows. A stack buffer overflow may enable the attacker to hijack the program return flow. In the most naive form of the attack, the attacker injects code into the program's memory by overwriting the stack with the desired code. Then, the attacker overwrites a return address, which is stored on the stack, with the address of his injected code. When the procedure returns, the program is directed to the attacker's code.

For example, consider a program that contains a vulnerable procedure `foo`. By attacking `foo` the attacker is able to inject code to the stack, and to overwrite `foo`'s return address with the address of his injected code. The resulting state of the stack is illustrated in 1.3.

When `foo` executes the return instruction, the program control flow is hijacked. Instead of continuing with the program's normal execution, the attacker code is executed. From that point, the attacker has complete control over the program.

Various security mitigation exist (and widely implemented) to prevent return flow hijacking (we describe them later). However, none of them is foolproof, and even when

```
foo:
push {fp, lr}
add fp, sp, #8
sub sp, sp, #12
```

Listing 1.1: Procedure Prologue in ARM Architecture

```
sub sp, fp, #8
pop {fp, pc}
```

Listing 1.2: Procedure Epilogue in ARM Architecture

all of these mitigations are applied, return flow hijacking attacks can still be successfully mounted.

1.3.1 The Call Stack and the Calling Convention

The *call stack* is a stack data structure used to store the state of the program's procedures call hierarchy. Each procedure has its own frame on the stack which holds its local variables and the return address. It is also common to use the stack for arguments passing.

Usually, the call stack is managed by two dedicated CPU registers. The *stack pointer* which points to the head of the stack and the *frame pointer* which points to the active procedure frame on the stack. Unlike other memory regions, the stack grows towards lower memory addresses. The communication protocol between the calling procedure and the callee procedure is called *calling convention*.

The calling convention determines how the caller passes arguments to the callee and how the caller receives the output. The calling convention also dictates which registers may be modified by the callee and which it must leave unchanged.

1.3.1.1 The Calling Convention of the ARM Architecture

In ARM CPUs a procedure is called using the BL instruction. BL loads the return address to the link register (LR) and the procedure address to the PC register. To return from a procedure, LR is loaded back into the PC. In the simple case in which a procedure does not call any other procedure (in which case it is called a leaf-procedure), the stack is never used to hold the return address. However, if the procedure does call other procedures, the return address is copied from the LR to the call stack during the procedure prologue (see Listing 1.1). The returned address is loaded back to the LR during the procedure epilogue (see Listing 1.2).

1.3.1.2 Control Flow Hijacking And ARM Architecture

Contrary to popular belief, the ARM architecture is vulnerable to return flow hijacking due to stack overwrite vulnerability. Although the stack does not store the return address in case of a leaf procedure, in all other cases the call stack does store the return

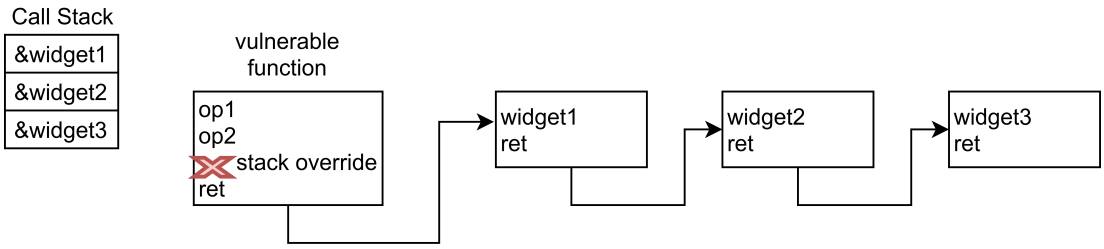


Figure 1.4: Return Oriented Programming Attack

address. By overwriting the stored LR of the current procedure frame, or of any previous frame on the call stack, the attacker may be able to hijack the program return flow.

1.3.2 Return-Oriented Programming (ROP)

Return-oriented programming (ROP) [57] is an exploitation technique designed to overcome security measures that prevents code injection. ROP enables an attacker to leverage a stack overwrite vulnerability into a full blown code execution exploit. By overwriting the stack with a set of carefully crafted addresses, the attacker takes over the program return flow. Each address is chosen so that a few desired instructions will be preformed before reaching another return instruction. Such a set of instructions is called a *ROP widget*. The addresses on the stack are set to trigger a specific sequence of widgets. We illustrate this in Figure 1.4. Various tools exist [49, 54, 45] to help build and design the widgets chain, in a way that performs an arbitrary code to the advantage of the attacker.

Although initially introduced for the x86 architecture, later work (such as [13]) showed that ROP is a general technique, which may be implemented on various architectures and operating systems.

1.4 Vulnerability Mitigations for Return Flow Hijacking

A *mitigation* of a vulnerability is a security mechanism which prevents an attacker from successfully exploiting the vulnerability. As the threat posed from return flow hijacking is high, various mitigations for this attack exist. These mitigations may be deployed by the compiler (e.g., stack canaries), provided by the CPU (e.g., NX bit), or supported by the operating system (e.g., ASLR). Modern computer systems utilize a combination of various vulnerability mitigations in order to increase the mitigations effectiveness, thus increasing the system security.

For example, the Linux kernel follows the “Kernel Self-Protection” guidelines [15]. As dictated by the guidelines, mitigations are employed in order to stop attackers with arbitrary read and write access to the kernel memory from achieving arbitrary kernel code execution. These mitigations include (but not limited to): ASLR, NX and stack canaries.

The following subsections describe some of the most widely used vulnerabilities mitigations.

1.4.1 Stack Canaries

Typically, stack overwrite attacks are caused by buffer overflows, which copy data whose size exceeds the buffer size into the buffer. The excessive data "overflows" the buffer, and overwrites a continuous memory region into the memory just adjacent to the buffer. Notice that in order to overwrite the return address, the attacker must also overwrite any value stored between the buffer and the return address.

A *stack canary* (sometimes called StackGuard) [14] is an unpredictable value stored on the stack, just before the return address. In case of a buffer overflow, the attacker must also overwrite the canary in order to overwrite the return address. Since the attacker cannot predict the canary value, its value is changed due to the attack. Before using the return address, the canary is checked. A canary value change indicates that an overwrite occurred. The code responsible to store and check the canaries may be deployed by the compiler during the compilation process.

Contemporary compilers, such as *GCC* and *Visual C* support canaries (the compiler options are named *-fstack-protector* and */GS* respectively). Moreover, by default canaries are deployed on prone to be vulnerable functions.

A canary is only effective if the attacker must overwrite it when trying to mount the attack. Therefore, canaries cannot protect against an attacker with arbitrary write ability, as the attacker will "skip" the canary and will not overwrite it. Another weakness of the canaries is that they may be somehow leaked. Attacker with read permissions may read the value of the canary, which nullifies the protection it provides.

1.4.2 No-eXecute (NX) bit

Modern CPUs provide a dedicated page table bit to mark memory regions as non-executable. The OS uses this bit to mark sensitive regions, such as the stack, as non-executable, thus, limiting attackers ability to inject their own code to the victim program memory. Intel refers to this bit by the name XD (eXecute Disable) [26], while ARM names it XN (eXecute Never) [9].

An extension of the NX bit is W^X (Write XOR eXecute), which enforces every writable memory region to be non-executable. ARM supports this feature by the WXN bit. To the best of our knowledge, Intel CPUs do not directly support W^X. However, the operating system can make sure that every time it is mapping a writable memory page it also marks it as non-executable.

While NX prevents an attacker from injecting his own code, it does not stop him from using existing code to his needs. Attacking techniques such as return-to-libc or ROP use such existing code, and are thus effective even when NX is present.

1.4.3 Address Space Layout Randomization

Address space layout randomization (ASLR) [64] randomly arranges the program's memory space by loading each of its sections (code, stack, heap, etc.) to a random base address. When applied to the kernel, this feature is called KASLR. Most of modern operating systems utilize KASLR at default configuration.

When ASLR is deployed, memory corruption vulnerabilities are harder to exploit, and in particular control flow hijacking. The attacker must find the code section load address before he may successfully utilize attacking techniques such as return-to-libc or ROP.

Unfortunately, ASLR is very sensitive to information leaks. The leakage of a single address may be sufficient to bypass ASLR. Another flaw of ASLR is the lack of sufficient entropy. Many systems do not provide good entropy source for ASLR, which weakens its protection.

1.4.4 Cross Privilege Level Access Prevention

Usually if the CPU is trying to access memory (to read data or to execute code) of a certain privilege level while operating in privilege level, malicious intentions are at place. For example, the CPU is not supposed to execute user space code while operating in kernel mode. If we allow cross privilege level execution, an unprivileged attacker who can overwrite a kernel return address, may direct the kernel to a controlled executable region in the user space memory.

Both ARM and Intel architectures provides mitigation for such cases. Intel's *Supervisor mode execution prevention* (SMEP) [26] stops the CPU from executing ring 3 code while in ring 0. ARM's architecture provides a similar feature dubbed *Privileged eXecute Never* (PXN) [9]. There are similar features to prevent cross privilege levels data accesses.

Cross domain code execution prevention does not stop an attacker from exploiting the kernel existing code to his benefit. In particular, it cannot defend against attacks such as *return to libc* or *ROP*.

1.4.5 Guard Pages

Guard pages are "empty" memory pages with specific (minimal) permissions, placed at the end of a memory region. An attacker trying to exploit out of boundary read/write vulnerability across memory region might be stopped by the guard pages.

To deploy guard pages, the loader usually places unmapped virtual memory pages (non-present bit set) "before" and "after" the protected memory region. An overflow (or underflow) of the protected memory region accesses the unmapped page and causes a page fault which stops the attack.

1.5 Outline of this Thesis

The rest of this thesis is organized as follows: Chapter 2 provides the necessary background on TEEs, in this chapter we provide a detailed description of TEE and describe the most widely used TEEs. In Chapter 3 we discuss previous related work, we describe the TEE standardization efforts, present various known attacks on TEEs and protection mechanisms for return flow hijacking. Chapter 4 presents TROOS, an OS for TEEs developed to mitigate widely spread security issues in TEEs. In Chapter 5 we present TKRFP, a TEE-based return flow protection mechanism. Chapter 6 introduces the CSOP attack, an attack that potentially may bypass shadow stacks mitigations for return flow hijacking. Finally, Chapter 7 summarizes our work.

Chapter 2

Trusted Execution Environments

Trusted isolated components are needed because the OS itself cannot provide reliable protection to critical assets (such as cryptographic keys or biometric data) and processes. The trusted components provide another layer of protection where critical assets can be stored outside of the OS reach.

A trusted component offers protected capabilities to the main execution environment (MEE). The protected capabilities consist of memory locations accessible only by the trusted component and operations preformed by the trusted component. There are various implementations of trusted components such as TPMs, secure elements, cryptographic processors, or trusted execution environments. In this work, we concentrate only in TEEs.

A *Trusted Execution Environment* (TEE) [24] is an isolated execution environment, which provides security features such as integrity of code and protection of assets. The TEE is separated from the main execution environment (MEE) of the device. In particular, the MEE operating system cannot directly access the TEE. A TEE is built in a way that maximizes the confidentiality and integrity of the relevant data and code. The TEE usually relies on some hardware features which force strict isolation from the MEE.

In order to communicate with the MEE, the TEE exposes an API. The MEE can use this API to request trusted services from the TEE. The MEE and the TEE typically communicate by a client-server protocol, where the MEE runs the client application and the TEE runs the server.

This description is too general and does not specify any concrete requirement that a TEE implementation must provide. In [67] McCune lists five key requirements that a TEE must fulfill in order be both secure and useful. McCune's work is considered milestone in the theoretical work on TEEs and these requirements are usually considered a part of the TEE definition.

Feature	Explanation
Isolated Execution	The ability to run an application in complete isolation from the MEE code (including the MEE OS).
Secure Storage	The ability to store data such that only its owner can read or modify it, in addition the integrity of the data must be ensured.
Remote Attestation	The ability to allow to remote parties to verify the state of the execution environment.
Secure Provisioning	The ability to provide data to a specific software module.
Trusted Path	The ability to preform secure I/O operations.

Table 2.1: The Main Requirements of a TEE

2.1 The Main TEE Requirements

This section provides a detailed description of McCune's main TEE requirements. A short summary is presented in Table 2.1.

2.1.1 Isolated Execution

Isolated execution provides the ability to run code and use data, such that the run-time integrity and secrecy is promised. This is the basic requirement from a TEE, and is a part of the GlobalPlatform's TEE definition.

Typically, TEEs provide two layers of isolation to their TAs. The first layer is the isolation of the TEE from the MEE. The second layer, is the isolation of a TA from others TAs in the TEE.

2.1.2 Secure Storage

Secure storage provides the ability to protect the integrity, secrecy and freshness of data at rest. By using secure storage, a TA can securely restore data stored in non-volatile memory.

The secure storage can use physically protected storage devices, or use cryptographic means to protect portions of "regular" storage devices.

It should be noted that providing data freshness is trickier than only providing integrity and secrecy. Therefore, the secure storage of most TEEs lacks proper freshness assurance.

2.1.3 Remote Attestation

Remote attestation allows remote parties to verify the state of the TEE and its TAs. The attestation is typically a report signed by the TEE. This report includes information, which can be used by the remote party to learn about the state of the TA and the TEE, such as the TA identity, a measurement of the TA code, and the TEE kernel version.

The signing key is typically a private key associated with the device, e.g., the TPM's platform endorsement key. However, in order to maintain privacy, the key is not directly used, instead the key is used with an anonymization scheme (e.g., EPID [30]).

For example, before providing a service to his clients, a cloud service provider can use remote attestation to verify that they comply with terms, such as using an updated TA version or using a certain TEE. A client can also use remote attestation before sending its credentials to a server.

2.1.4 Secure Provisioning

Secure provisioning provides the ability to send data to a specific application on a specific device, while maintaining the integrity and secrecy of this data. Typically, secure provisioning is preformed by encrypting the data with a key known only to the target application.

In the common case, a TA first proves its identity and state to a server (using the remote attestation capability), and then the server uses secure provisioning to provision secrets to the already trusted TA.

2.1.5 Trusted Path

A trusted path provides the ability to authenticate peripheral devices (such as a touchscreen or a keyboard) and to securely interact with these devices. By using the trusted path, a TA can securely use a peripheral device with assurance that the MEE cannot compromise the communication with the device. In particular, the integrity and secrecy of the communication with the device is promised. Moreover, a user that interacts with a TA via trusted path can be certain of the TA identity.

For example, the trusted path can be used to implement a secure pin reader in the TEE, which provides two important properties: the inserted pin can be read only by the TEE, and the user can be sure that the TEE presents the pin reader.

2.2 Examples of Use Cases

In order to provide services to the MEE, the TEE creates *trusted applications* (TA), which are the processes that run inside the TEE. Typically each trusted service is provided by a different TA. This section presents several TA use cases.

2.2.1 Trusted Video Player

TEE can significantly improve digital rights management (DRM) security. Consider an online media service provider, such as Netflix. The media service provider wishes to prevent illegal distribution of content. However, the service provider cannot trust the OS nor the application used to play the content, as it has no control over any of them.

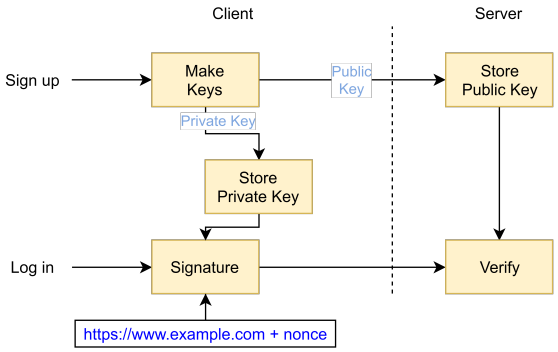


Figure 2.1: The SQRL Authentication Scheme

For example, how can the media service provider be sure that a certain client does not use a tempered OS version that records the content instead of simply playing it?

This problem can be solved using TEE. The media service provider uses a secure video player TA inside the TEE. After authenticating the secure player TA (using the TEE's remote attestation capability), the media can be securely provisioned to the secure video player. The secure player TA can then use the trusted path capability to play the content without it ever being exposed to the MEE.

2.2.2 Trusted Bitcoin Wallet

A hardware Bitcoin wallet is a dedicated external device, typically a USB stick, designed to serve only as wallet. These wallets are now considered the best practice for Bitcoin storage.

We can use the TEE to implement a secure Bitcoin wallet, which offers similar security level similar as an hardware wallet, without using any external device. Thus, getting a better user experience and saving money.

A trusted Bitcoin wallet can provide all the functionality offered by an hardware wallet. It has secure storage for long-term storage of the Bitcoin keys, an isolated execution area where these keys can be used isolated from the rest of the system, and a trusted path to interact with the user.

2.2.3 Trusted SQRL Authenticator

SQRL is a new open standard for secure website login and authentication, which is still in a draft state. In SQRL when a user wishes to sign up to a website he creates a pair of asymmetric keys. The public key is sent to the website to be used as the user's identity token for this website. On a login request a user sends his public key to the website. The website replies with a nonce (to prevent replay attacks). The user then uses his private key to sign a concatenation of the website domain and the nonce, thus proving ownership on the public key and authenticating himself. The SQRL authentication scheme is outlined in Figure 2.1.

In a naive implementation, a malware with privileged access rights may be able to access the SQRL private keys, thus to gain access to any user account which is authenticated using SQRL. A TEE can improve the security of SQRL by storing the private keys, and providing a signing service. Upon a login request, the TEE signs the login challenge, while the private key is safely stored in the TEE memory. By using a TEE we can prevent even the highest privileged MEE malware from accessing the SQRL private keys.

While the offered solution prevents a privileged malware from having the option to directly access the SQRL private keys, this malware can still preform unauthorized login by using the TEE signing service. In order to prevent this attack as well, the TEE can ask the user for a pin (using the trusted path feature). Only after the pin is verified the TEE signs the challenge.

2.2.4 Kernel Integrity Checker

The TEE may be used to provide monitoring and supervision of critical components of the MEE. A TA that offers such a service does not rely on any input provided by the MEE. Instead, it directly accesses the MEE resources in a transparent way. To make such services effective, the ability to invoke them without any MEE intervention is required.

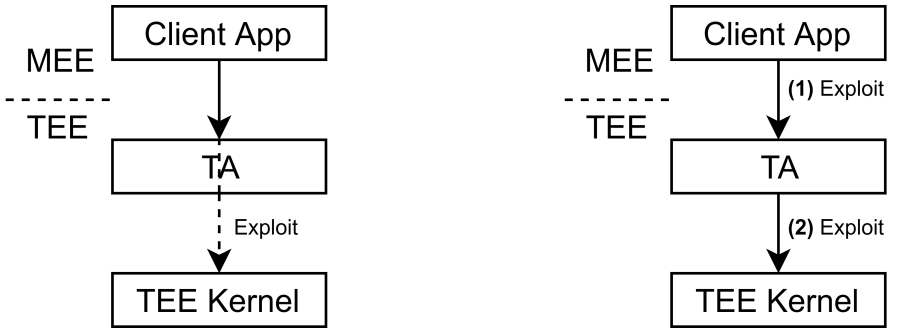
A TA acting as a kernel integrity checker [70] is invoked periodically by the TEE. Upon invocation, the TA directly accesses the MEE memory, scans the kernel code and read-only data, and preforms integrity checks.

2.3 The TEE TCB and Attack Surface

As the keen reader may figure out a TEE is not "magically" secure. It is only granted to be isolated from the MEE. An attacker might still exploit the TEE API to try to compromise it. If the TEE is not carefully designed and implemented, the attacker might succeed (as shown later in this thesis). Therefore, a TEE must have the smallest attack surface as possible. In order to reduce the TEE attack surface, its TCB must be as minimal as possible. Reducing the TEE's TCB is one of the leading guidelines in TEE design.

Our threat model assumes that the attacker has no initial foothold inside the TEE, and therefore the attacker may exploit the TEE only via its external API. Hence, the TEE's attack surface is upper bounded by the TEE external API. Since the TEE external API includes all the interfaces of the TAs, the TA's interfaces become a major part of the TEE's attack surface.

For example, the attacker can use an interface exposed by a vulnerable TA to exploit the TA, and thus gain an initial foothold in the TEE. Then, the attacker can use its foothold in the TEE to mount an attack on the TEE kernel. This two-phases attack



(a) Single-Phase Attack (b) Two-Phases Attack

Figure 2.2: TA as an Attack Vector of the TEE Kernel

requires the attacker to exploit the TA in a manner that enables him to run code within the TA context. Another example uses the TA to exploit the TEE kernel without running code within the TA context. Instead, the attacker uses an existing TA code that already interacts with the TEE kernel. In this single-phase attack, the attacker uses a TA API that allows him to control the arguments that the TA passes to the kernel API, resulting with the attacker being able to interact with the TEE kernel without running his own code within the TEE. These two types of attacks on the TEE kernel are outlined in Figure 2.2.

Other types of TEE interfaces exposed to the MEE are the management and configuration interfaces. These interfaces are used to control the TEE, rather than to interact with its TAs. For example, a TEE may allow the MEE to load new TAs (after proper authentication) in order to extend the trusted services it provides. Attackers may try to exploit any of these interfaces as well.

The MEE's security policy typically restricts access to the TEE, and allows only carefully selected entities to interact with it. However, the TEE cannot trust the MEE, and therefore must treat any MEE request as untrusted.

A well designed TEE needs to keep its external API to the bare minimum. An effort to reduce the internal API between the TAs and the TEE kernel should also be taken. As shown later in this thesis, it is not the case in existing TEEs.

2.4 ARM TrustZone

One of the most widely used TEE technologies is ARM's TrustZone. It is a security extension that provides the hardware support required to establish a TEE in the processor. TrustZone enables to create an isolated execution environment, which can be used for security purposes.

ARM has two CPU architectures: A [1, 3] and M [2, 4]. ARM Cortex-A [5] is series of CPUs implementing the A-architecture. These CPUs are intended for application use, and can be found on smartphones, tablets and even PCs. ARM Cortex-M [6] is a series of CPUs implementing the M-architecture. These CPUs are intended to be used as

micro-controllers, and are optimized for low cost and low power usage. The Cortex-M CPUs are widely used in Internet of Things devices, engine controller modules, industrial control systems, and household appliances. Naturally, they are not as powerful as the Cortex-A series CPUs.

TrustZone was initially available only in Cortex-A CPUs. Nowadays TrustZone is also available in Cortex-M CPUs, after the new version of the M-architecture, ARMv8M [4], extended it with TrustZone support. The concepts of the TrustZone are similar on both the A and M architectures. However, the underlying operations are different, and optimized to fit each architecture common use cases.

The two most significant differences are that in contrast to the A-architecture the M-architecture does not have an MMU, and that the M-architecture has higher concern for performance (especially in terms of time and power).

Since TrustZone is slightly different between the two architectures, we use the term TrustZone-A when we discuss TrustZone in the A-architecture, and the term TrustZone-M when we discuss TrustZone in the M-architecture. The term TrustZone is used when discuss on concepts that are relevant for both technologies, or when it can be understood from the context whether it is TrustZone-A or TrustZone-M.

2.4.1 ARM TrsutZone-A

ARM TrustZone security extension [8] provides the technology for splitting the CPU to two security states: A non-secure state, which is called the *normal world* (NW), and a secure state, which is called the *secure world* (SW). The hardware enforces isolation between the two states, and ensures that secure resources cannot be accessed by code running in the non-secure state. In particular, the memory (both physical and virtual) is divided to secure and non-secure regions. The secure memory regions cannot be accessed while the CPU is in its non-secure state. Each state manages its virtual space separately with its own translation tables. The CPU secure/non-secure state is determined according to a bit called *non-secure* (NS) in the *security configuration register* (SCR) of the CPU.

TrustZone provides the hardware support required in order to implement TEEs. The SW is used to run the TEE, while the NW runs the MEE. The isolation enforced by TrustZone maintains the necessary separation between the two environments.

2.4.1.1 World Switch

The CPU state switching is done by code running in a special CPU mode called *monitor mode*. This mode is accessible to the NW by a dedicated privileged instruction called *secure monitor call* (SMC). The SMC instruction triggers an exception, which is handled by the SMC handler. The handler is defined by the registered secure monitor vector.

Arguments (e.g., which service is requested) are typically passed to the monitor in registers. However, the SMC instruction also has a four-bit immediate value, which is

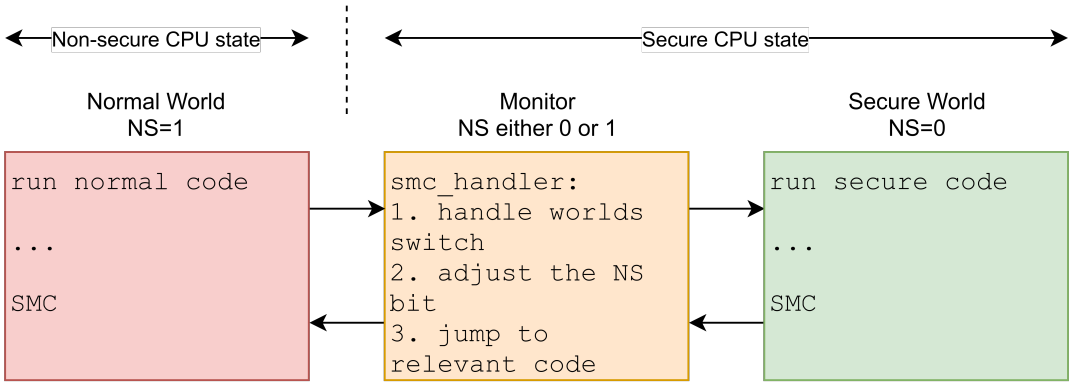


Figure 2.3: TrustZone-A World Switch

ignored by the CPU, but can be used as an argument for the SMC handler.

Notice that the SMC instruction does not flip the NS bit. However, while in monitor mode the CPU is considered to be running in its secure state regardless of the the NS bit value.

The monitor mode code typically preforms the following actions:

1. Checks which world invoked the SMC instruction (by checking the NS bit).
2. Saves the current world state.
3. Restores the state of the called world.
4. Updates the value of the NS bit according to the called world.
5. Passes control to the called world.

Figure 2.3 outlines the flow of TrustZone-A world switches. The non-secure to secure switch starts with an SMC instruction which causes the smc_handler routine in monitor mode to run. The smc_handler routine identifies that it should preform a non-secure to secure switch, saves the state of the non-secure world, restores the secure world state (if needed), clears the NS bit, and starts the secure world code execution. To return back to non-secure mode, the secure code invokes an SMC instruction and the monitor preforms the switch in a manner similar to the non-secure to secure switch.

Although the secure world can directly access and change the NS bit, it typically does not do so. Instead, the secure world also uses the monitor for world switch. The monitor mode can preform the protection measures needed before returning to the normal world (such as overwriting registers to prevent secrets leakage). In this way, there is only one controlled gateway between the two worlds.

2.4.1.2 Memory Isolation

TrustZone provides the capability to isolate the secure state memory accesses from non-secure state accesses. The isolation is done in two layers, physical and virtual.

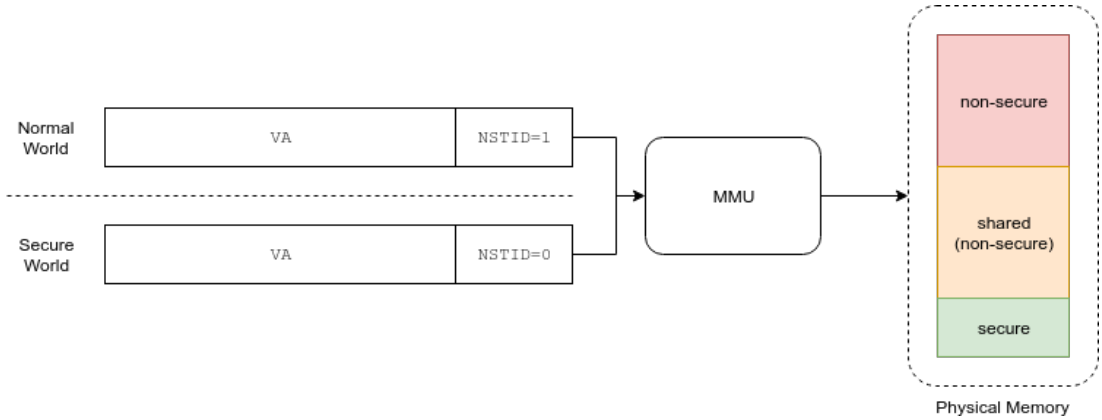


Figure 2.4: TrustZone-A Memory Isolation

For virtual memory isolation, each world uses its own virtual memory management unit (MMU), each world maintains its own local set of translation tables. In this way, each world independently controls its own virtual to physical address-mappings. The translation table descriptors are extended with an NS bit which is used by the MMU to determine if the memory access should be treated as secure or non-secure. When in normal world, the hardware ignores this bit and considers it to be set, hence all memory accesses are considered non-secure. The secure world maps its memory with the NS bit unset, hence memory accesses are secured. The secure world may also map some memory regions with NS bit set, and as a result accesses to this region are considered as non-secure creating a shared memory with the normal world.

The physical memory is protected by the memory controller. Each block of memory is configured to be either secure or non-secure. The memory controller denies any access to secure memory preformed by the CPU while in non-secure state.

To summarize, on a TrustZone enabled CPU, a virtual address (VA) is translated to a physical address by the MMU according to the CPU state, i.e., the same VA may be translated into different physical addresses depending on the CPU state. The memory controller denies any non-secure access to the secure memory. Figure 2.4 outlines this memory isolation scheme.

2.4.1.3 SoC Devices Isolation

ARM CPUs are typically used as part of a system on a chip (SoC). The SoC contains various devices (like CPU, modem and GPS) which communicate with each other via the system bus. TrustZone enables to define each of the SoC devices as a secure device or as a non-secure device.

An NS control signal is transmitted on the system bus channel. When a device issues a new transaction to the system bus, this signal is set according to the device security level. The hardware implementation must make sure that every non-secure master device in the SoC has its NS control signal set to high. The recipient of the

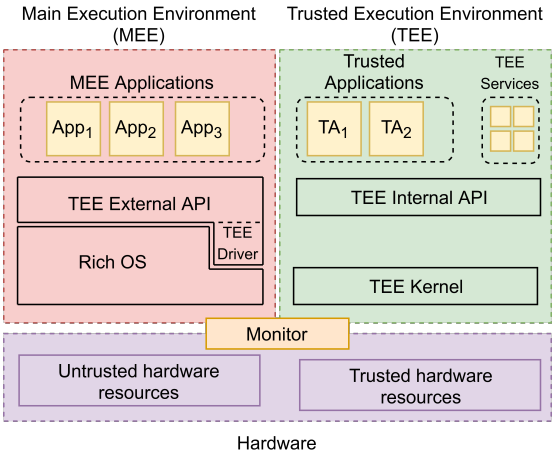


Figure 2.5: The Common Structure of a TrustZone-Based TEE

transaction is in charge of enforcing the security policy, and denies any non-secure accesses to secure devices.

2.4.1.4 Peripheral Bus

In addition to its own internal devices, a SoC usually interacts with external devices. The external devices communicate with each other and with the SoC via the peripheral bus. The SoC is connected to the peripheral bus through an AXI-to-APB bridge [7].

For backward compatibility support, an NS control signal was not added to the peripheral bus channels. Instead, the AXI-to-APB bridge is responsible to enforce the security of the peripheral devices.

2.4.2 The Typical TrustZone-A Based TEE Structure

We wish to address the typical design of a TrustZone-based TEE. Although other approaches may be considered while designing a TrustZone-based TEE, all the TEE implementations we investigate comply to this typical design. Figure 2.5 illustrates this design.

Recall that TrustZone provides two CPU states, secure and non-secure, each with its own exception level stack. The secure memory is isolated from the non-secure state, and SoC devices defined as secure are also isolated from the non-secure state. The secure state along with its resources is called the secure world, while the non-secure state along with its resources is called the normal world.

In order to completely isolate the TEE from the MEE, the TEE resides in the secure world. The TEE usually consists of a trusted OS, namely the TEE kernel, running in supervisor mode (EL1), and trusted applications (TAs) running in user mode (EL0). Like any OS, the TEE kernel may offer various services to its application. These services may be provided by system calls through its internal API, or by dedicated components. The user mode components (e.g., the TAs) interact with each other via

the IPC mechanisms offered by the TEE kernel.

The TEE kernel should enforce strict isolation between its components. To achieve this goal, the TEE kernel may use the tools provided by the ARM architecture. For example, each trusted application uses its own translation table, for isolating the address spaces of the TAs from each other.

The TAs provide services to the MEE applications through the TEE external API, typically by some kind of client–server protocol. The requests to the TAs are issued by invoking the SMC instruction. The MEE makes sure to pass the trusted application identifier as well as the request arguments to the SMC instruction. Because the SMC instruction is a privileged instruction (i.e., can be invoked only the the kernel), the MEE exposes the TEE services to its application via a driver interface.

Upon SMC invocation the monitor is responsible to preform a secure switch from the normal world to the secure world and to pass the request to the TEE kernel. Then, the TEE kernel directs the request to the relevant TA. which preforms the requested task.

2.4.3 ARM TrsutZone-M

In TrustZone-M, the separation of memory is preformed by a new hardware unit called *Security Attribution Unit* (SAU). The SAU enables the system to split the memory to secure and non-secure regions by assigning each region with a corresponding attribute.

Unlike in TrustZone-A, the CPU state is determined by the security attribute of the code's memory region. The *Memory Protection Unit* (MPU) prevents access to secure memory while in non-secure state.

2.4.3.1 World Switch

The secure world provides *secure gateways* in order to allow the CPU to switch its state from non-secure to secure. These secure gateways are implemented by executing the SG instruction in *non-secure callable* (NSC) memory regions.

The SG instruction represents a valid entry point to the secure world and prevents non-secure code from being able to jump to arbitrary addresses in the secure world code.

The NSC memory regions are secure regions with a special feature: Unlike regular secure memory, non-secure code can branch to NSC memory. However, the CPU remains in its non-secure state until the SG instruction is invoked. Therefore, if the first instruction after the branching to the NSC memory is any instruction other than the SG instruction a security exception is triggered.

Figure 2.6 exemplifies a normal world to secure world switch. The non-secure code wishes to use a trusted service named bar. The non-secure code first branches to bar's secure gateway in the NSC memory. The SG instruction switches the CPU state to

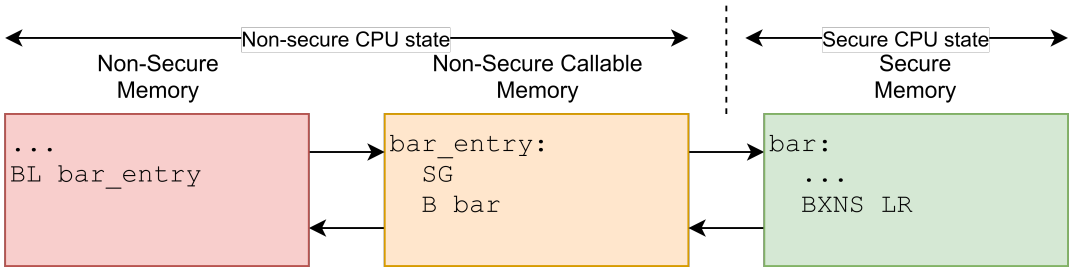


Figure 2.6: TrustZone-M World Switch

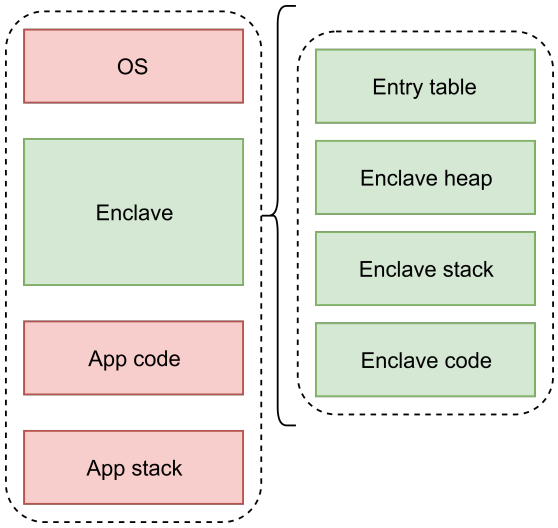


Figure 2.7: Enclave's Memory, and its Location Within the Application's Virtual Address Space

secure, and then the control is passed to the code that resides in the secure memory region.

To return to normal world the secure world uses the BXNS instruction, which branches to an address specified by a register, and causes a transition from the secure to the son-secure world. In the above example the BXNS instruction uses the LR, which was loaded during the call to bar_entry.

2.5 Software Guard Extension (SGX)

Intel Software Guard Extension (SGX) [27] is an `x86_64` instructions set architecture extension which enables unprivileged (user) process to create *enclaves* within its virtual address space. These enclaves consist of secure memory regions that cannot be accessed by any external entity. Even high privileged code (e.g., the OS code) cannot access an enclave created by a user. Figure 2.7 outlines a virtual address space containing an enclave and the enclave's memory sections. The enclave's memory regions are encrypted by the CPU, and only decrypted within the CPU perimeter. Hence, enclave code and data cannot be read by a regular process or the OS, even if they have unlimited memory

access capabilities.

In addition, SGX support attestation capabilities: special instructions enable to measure and verify the content loaded into the enclave before its execution, and provide both local and remote attestation.

2.5.1 Memory Management

The enclave's memory pages reside in the *enclave page cache* (EPC), in a dedicated area in the physical memory. Each page is associated with a specific enclave and may contain data or meta-data used by the CPU. The OS manages the EPC resources using dedicated CPU instructions. For example, when a new enclave is created the OS is responsible to find free EPC pages ans assign them to the new enclave. Nonetheless the content of the EPC is out of the OS reach, and is cryptographically protected by the CPU.

2.5.1.1 Enclave Page Cache Map

The *enclave page cache map* EPCM is a data structure accessible only to the hardware, which is used by the CPU to manage the EPC. It holds an entry for each page in the EPC stating its stats, type, access permissions, etc.

2.5.1.2 Memory Accesses

The memory access semantic was also changed to support enclave isolation. While running in enclave mode, the CPU can access the entire application's virtual address space. In any other mode the CPU is not allowed to access the EPC at all. Notice that, each enclave can access only its own pages and the pages pf its parent application, and that due to the page table (and the cryptographic protections) it cannot access memory of other enclaves. The flow of SGX memory accesses is outlined in Figure 2.8.

2.5.2 Enclave Creation

The enclave creation process is handled by the OS, using privileged CPU instructions. In order to create a new enclave, the OS first locates a free EPC page for the enclave meta-data, then it uses *ECREATE* privileged instruction to fill this page with the enclave meat-data (such as: base address, size, attributes), and to assign it to that enclave.

The new (empty) enclave is filled with content using the *EADD* privileged instruction which loads content to a free EPC page and associate it with an enclave. The *EEXTEND* privileged instruction can be used to measure the enclave content. Once the building process is completed, the *EINIT* privileged instruction is used to mark the enclave as ready for execution. The enclave content can no longer be changed afterwards.

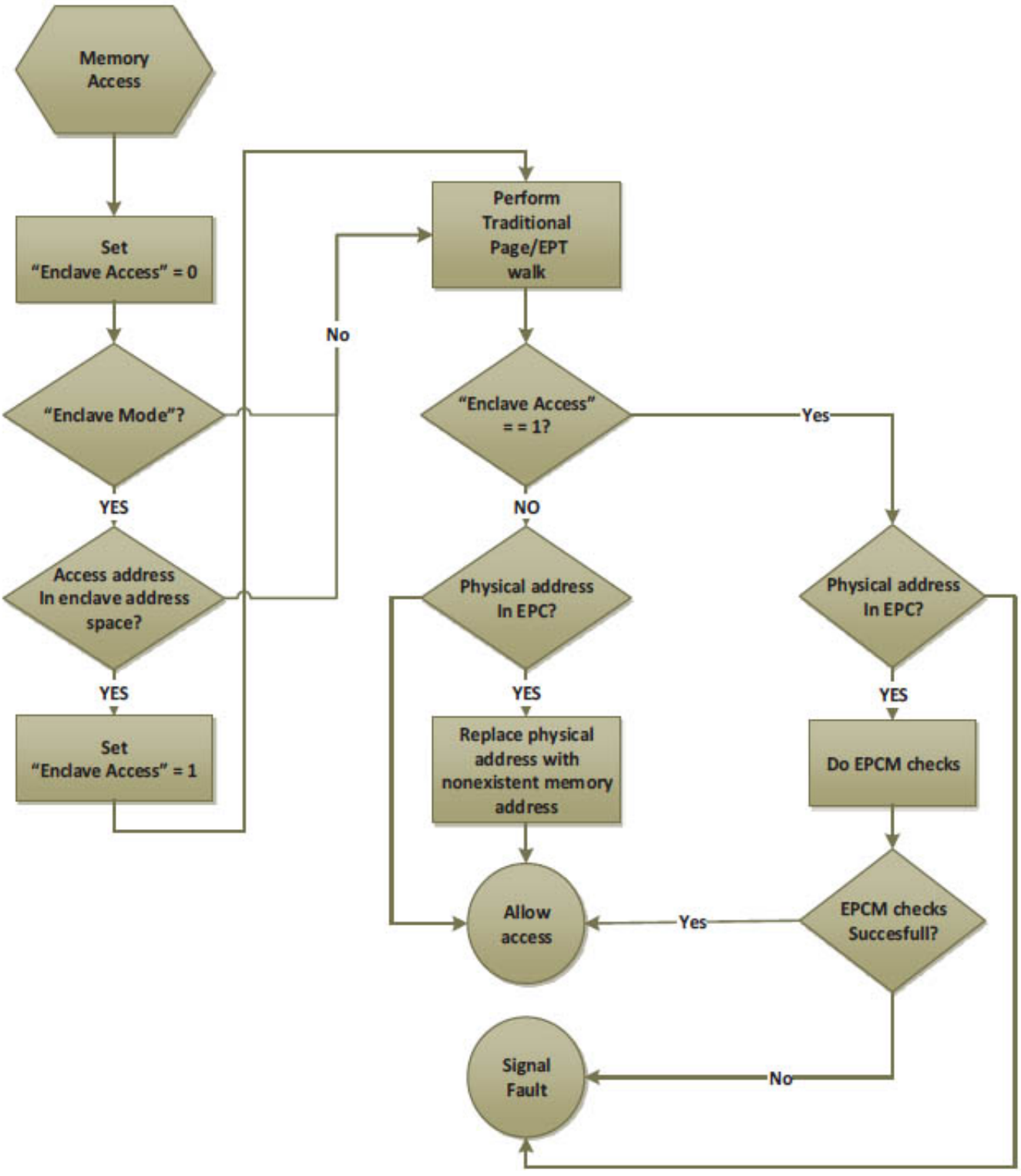


Figure 2.8: The Flow of SGX Memory Accesses [37]

2.5.3 Enclave Execution

The *EENTER* instruction is used to start enclave execution from one of as set of predefined entry points. The *EEXIT* instruction is used to exit enclave mode and return the control to the user. If an interrupt or exception occur while executing under enclave mode, the enclave execution is stopped, its state is saved and control is transferred to the OS. The *ERESUME* instruction is used to continue enclave execution after such asynchronous stops.

2.6 TEEs Memory Encryption

By its definition, the TEE should have its own isolated memory that cannot be accessed by the MEE. However, in most cases the TEE carves its memory from the device's main memory, and enforces isolation by denying the MEE from accessing this region of memory.

The TEE threat model assumes a highly sophisticated attacker, with high level privileges in the MEE, and physical access to the device. Since the TEE contains high value targets (such as DRM protection keys, biometric data and payments credentials), probing attacks against the system bus and memory are plausible scenarios.

Intel SGX offers a solution for this problem by encrypting any TEE information leaving the CPU package. The TEE information is decrypted only when it is re-loaded to the CPU package. An attacker that probes the system bus or memory can read only encrypted information, unusable to him.

On the other hand, TrustZone architecture by itself is not sufficient to prevent probing attacks. However, an equivalent (but very different) memory encryption protection scheme can be implemented for TrustZone-based TEEs by utilizing the SoC internal memory (if such memory exists). TrustZone can isolate this internal memory, which cannot be probed by attackers, from the MEE, thus providing probing protected memory for the TEE.

2.6.1 On Chip Memory

On Chip Memory (OCM) is a chunk of memory located within the SoC package. OCM typical size ranges from 64KB to 256KB. For example, Freescale's `imx53qsb`, which is a very chip and unsophisticated development board, has 128KB of internal fast access RAM. It should be noted that OCM is an additional RAM memory and is not part of the CPU cache. The OCM is located outside of the CPU package but within the boundary of the SoC, as outlined in Figure 2.9.

Using TrustZone we can configure the OCM as a secure memory, thus make it a part of the TEE memory. TrustZone enforces access control policy that denies any MEE component from accessing the OCM. Additionally, the OCM is physically located inside the SoC package hence it is not susceptible to external probing attacks.

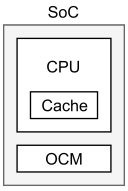
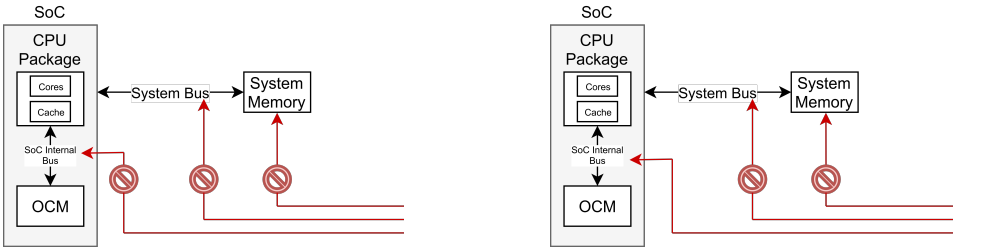


Figure 2.9: OCM



(a) Security Boundary of SGX (b) Security Boundary of OCM

Figure 2.10: Encrypted Memory Security Boundaries

2.6.2 Memory Encryption in TrustZone-Based TEEs

The OCM as an SoC internal memory, it is more harder to probe the OCM than the main memory or the system bus. The security perimeter of information stored in the SoC is merely the SoC package. This is similar to the protection offered by Intel's SGX. Figure 2.10 compares the security boundary of SGX to the security boundary of an OCM based solution. Both solutions offer protection against bus and memory probing attacks. However, the SGX security boundary is limited to the CPU package while the OCM based solution security boundary contains the entire SoC.

The OCM is typically too small to meet all of the TEE memory requirements, and thus it cannot be used as a sole memory resource. When the TEE requires more memory than the OCM offers, it can use the system memory as a swap area for the OCM. The TEE reads data and executes code only from the OCM pages, if the entire OCM is occupied the TEE selects a page to be swapped out to the main memory. Before swapping out the page, the TEE encrypts it and copies the encrypted page to the main memory. If the TEE needs to use a swapped out page, it first copies the encrypted page to the OCM and only then decrypts it. In that way all memory pages outside of the OCM are encrypted (and thus protected).

2.6.2.1 Memory Encryption in OP-TEE

According to its documentation [47], OP-TEE assumes that for security reasons some implementations may want to limit the usage of physical memory to the SRAM alone. Since the SRAM size is usually not sufficient to run the entire TEE, OP-TEE implements on demand paging mechanism, called the *pager*. The pager is used to swap out memory pages from the SRAM to the DRAM (secure or non-secure) in order to free memory,

and to swap them back to the SRAM on demand.

The swapped out memory pages are protected according to the page access permissions:

Read/Write AES-GCM encrypted. The pager encryption key is randomly generated at boot time. The encrypted data is stored in the DRAM, IV and GCM tag are stored by the pager in SRAM.

Read Only Integrity protected plain text. The pager holds SHA256 hashes of every RO swapped out page. The page content is verified before swap in. According to the documentation RO pages are not encrypted since the binary is not encrypted anyway.

Replay attacks are not possible, since the pager always holds the swapped out pages integrity protection tag (may it be SHA256 hash in the case of RO pages or GCM-TAG in the case of WR pages).

To the best of our knowledge, OP-TEE is the only TrustZone-based TEE that provides a memory encryption mechanism. In regards QSEE and Kinibi, according to many documentations and research paper, neither one provides memory encryption.

Chapter 3

Related Work

The first step towards the definition of TEE was introduced by Terra [18]. Terra uses hardware virtualization to create isolated execution environments that run side by side on the same hardware. Qubes [52] is an operating, system which uses a similar technique to increase its security. In Qubes, the application are running in isolated *qubes*, thus allowing to compartmentalize various parts of the system. These qubes are virtual machines, and the isolation is achieved by utilizing the hardware virtualization capabilities. Recently, Microsoft introduced Virtualization-based Security [38] (VBS), which is basically a virtualization based TEE.

The idea of an isolated execution environment that can withstand attacks even from the high privileged software was extended by OMTP in [46] and later by GlobalPlatform [24] to define TEE. In [67] McCune adds to GlobalPlatform's TEE definition a list of five requirements that a TEE must fulfill, which are now widely considered as a part of the TEE definition (these requirements were detailedly described in the introduction).

3.1 The GlobalPlatform Specification

GlobalPlatform [21] is a nonprofit organization that develops and publishes specifications for secure chip technology. It is the leading organization in standardization of trusted execution environments specification.

GlobalPlatform's TEE System Architecture [24] describes the hardware and software architectures required in order to implement a TEE. It provides a description of a TEE design and lists its requirements. Most of the TrustZone-based TEE architectures resembles the architecture suggested by GlobalPlatform, but do not fully comply with their specification.

The TEE Client API Specification [22] defines the external API, which is a set of protocols by which a TA may communicate with the MEE. It defines the concept of *sessions* (which are communication channels between a TA and an application running in the main execution environment), the messages structure, and a method to transfer data between the MEE and the TEE via a shared memory region.

Name	Compliance
QSEE	✗
Kinibi	✓
OP–TEE	✗

Table 3.1: Compliance to the GlobalPlatform API

The TEE Internal API Specification v1.0 [23] defines the internal API, which specifies how TEE components interact with each other. It defines basic OS functionalities (such as memory management, timer, and access to configuration properties), communication methods, trusted storage facilities, cryptographic facilities and time management facilities. The specification also provides some guidelines for the security features that a TEE must fulfill.

Despite GlobalPlatform efforts, not all TrustZone-based TEEs comply with their specifications. For example, the most prevalent TEE is Qualcomm Secure Execution Environment (QSEE) which dose not comply to GlobalPlatform specification. OP-TEE is another widely used TEE, which does not comply with the specification. On the other hand, Trustonic's Kinibi fully comply with GlobalPlatform specification. Table 3.1 summarize this discussion. The lack of standard TEE API takes a toll on TAs development. For example, it is very hard to port a TA from one TEE to another. Moreover, the MEE application also needs to be re-written in case the TEE is changed.

3.2 TrustZone-Based TEEs Related Work

There are various TrustZone-based TEE implementations, designed by both the academy and the industry. A TrustZone-based TEE includes the monitor mode code and the secure OS. In the industry, Qualcomm's QSEE [48] is the most common with a market share of around 60% [32]. Trustonic [65] (formerly known as Mobicore) is also a major player on commodity devices. There are a few open source TrustZone-based TEE projects, out of which the most complete and mature one is Linaro's OP-TEE [35].

The academia focuses mainly on special propose TEEs (e.g., specially designed for industrial control systems or real time). For example, Andix [17] is a TEE OS for ICS that was developed in Gratz university, and SafeG [55] is a TEE OS designed for embedded systems with real time restrictions, developed in Nagoya university.

3.2.1 Attacks on TrustZone-Based TEEs

A few vulnerabilities were successfully exploited on various TrustZone-based TEEs. Rosenberg [51] exploited a QSEE integer overflow vulnerability to achieve arbitrary code execution in the TEE. Affected devices include various poplar smartphones like LG Nexus 5, LG G2, HTC One and Samsung Galaxy S4. Another work by Rosenberg [50] shows how to unlock the Motorola Razr series bootloader by exploiting an arbitrary

memory write vulnerability. An exploit to a vulnerability on Mobicore (later called Trustonic) that allows to run code in the TEE was published by Se nsePost in [56]. More recently, Beniamini successfully exploited both QSEE and Kinibi to achieve arbitrary code execution inside the TEE [12].

To the best of our knowledge all the attacks on TrustZone-based TEEs are software attacks on the code running in the TEE (exploited via the TEE interface). We are not familiar with any attack exploiting a weakness in the TrustZone hardware implementation itself or its specification.

3.3 Side-Channel Attacks on TEEs

A side channel attack uses measurable side-effects gathered from the environment or the attacked system, rather than just the intended API of the attacked system. For example, by monitoring households electric usage the police is able to identify illegal marijuana farms, since these farms require constant electric consuming lighting, their electric usage spikes and can be identified.

By their definition TEEs are implemented alongside with the MEE, i.e., the TEE is sharing the system with untrusted code. For example, in TrustZone the TEE and the MEE share the same CPU. This fact makes TEEs susceptible to side channel attacks.

Cache timing attacks are a type of side-channel attack, that uses the time difference between accessing cached memory to accessing non-cached memory. The cache is typically a shared resource, which is used by various components. In a cache timing attack, the attacker and its victim share the same cache. By measuring the effects its victim has on the cache, the deduces the memory content of the victim, i.e., information leaks from the victim to the attacker. As described in the following subsection, both TrustZone and SGX are vulnerable to cache timing attacks.

3.3.1 Side-Channel Attacks on TrustZone

TrustZone does not use a separate cache for its memory access, but uses the same cache as the MEE. Instead, an attribute bit in the cache tags indicates which CPU state (secure or non-secure) is the owner of each cache line. While in non-secure state, the CPU cannot access any cache line marked as secure (and if it tries, a cache miss occurs).

However, cache eviction policies ignore the tag's NS bit Therefore, a non-secure memory access may cause the eviction of a secure cache line and vice-versa. This fact in used on [36] and more extensively in [72] to mount cache timing attacks on TrustZone from the MEE. In particular, it is demonstrated in [72] how encryption keys used by a TA within TrustZone can be leaked to unprivileged MEE code.

3.3.2 Side Channel Attacks on SGX

The SGX specification explicitly states that side-channel attacks are not part of its threat model, i.e., SGX is not promised to be protected from side-channel attacks. And indeed, many side channel attacks on SGX were published.

Discussing all side channel attacks on SGX is out of scope for this thesis. Instead, we mention the two most significant attacks. The first [25] shows that SGX is vulnerable to cache timing attacks, and that they can be used to leak AES encryption keys from within enclaves. The second introduces the Foreshadow [66] attack, which uses a speculative execution bug in the CPU to extract enclave secrets from the cache. This attack was then used to leak the SGX remote attestation private keys, thus completely breaking SGX ability to preform reliable remote attestation.

3.4 Shadow Stack Protection Mechanisms for Return Flow Hijacking

The program's return flow can be protected by a *shadow stack*, a second stack used only to store the return addresses. On every procedure call the return address is stored on both the program stack and the shadow stack. Before every return instruction the return address is compared to the address on the shadow stack. If the addresses match, the procedure returns. If they do not match, an attack is assumed and proper action is taken.

Shadow stacks promise complete protection against return flow hijacking. However, currently there is no shadow stack based solution available in the market. It turns out that it is not easy to properly implement a shadow stack based solution.

The problem seems to be with contradicting requirements: Unlike the call stack which is writable by nature, the shadow stack should be write-protected to prevent shadow stack overwrites, and at the same time it should support push and pop instructions (i.e., be writable). In addition, the shadow stack is accessed at every procedure call and at every procedure return, so these accesses must be fast enough to have minimal performance overhead. The combination of these requirements complicates any shadow stack based solution implementation.

In this section we describe two attempts to build a shadow stack based return flow protection solution. We also explain why none of these solutions can provide a complete remedy for the problem.

3.4.1 Intel Control Flow Enforcement Technology

Intel control flow enforcement technology [28] (CET) incorporates two hardware features for preventing control flow hijacking. The first is a shadow stack, whose goal is to stop return flow hijacking. The other is an indirect branch protector, whose goal is

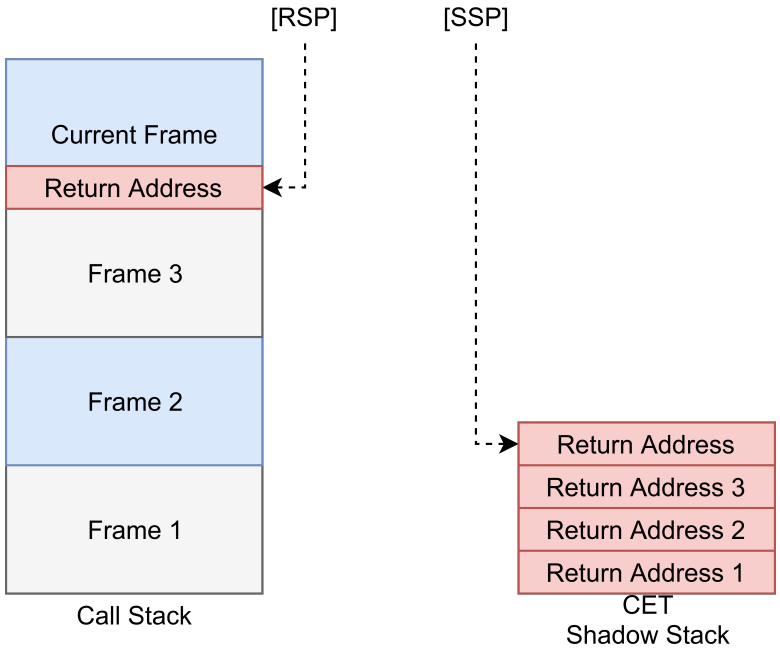


Figure 3.1: The CET Shadow Stack

prevent the program from preforming indirect branches to unintended locations. For the purpose of our work we are interested only in the former.

Intel's CET introduces a new register, the shadow stack pointer (SSP) was introduced. Recall that without CET, a call instruction pushes the return address to the regular stack, as pointed by the regular stack pointer (RSP). When CET is enabled, the call instruction also pushes the return address to the shadow stack (pointed by the SSP), as illustrated in Figure 3.1. . The ret instruction pops the shadow stack and the call stack, and compares the popped values. If the values are equal, the procedure returns. Otherwise, the CPU raises a security exception.

The CET shadow stack resides in the protected program (virtual) memory space. However, it is protected from overwrites since it is mapped into non-writable pages. CET does not add new control bits to mark that is memory pages are shadow stack related. Instead, this is done by a specific pattern of the R/W and dirty bits throughout the page tables hierarchy: in all non-leaf page table entires of the shadow stack, the R/W bit is set to 1, and in the leaf page table entries of the shadow stack the R/W bit is set to 0 with the dirty bit set to 1. Even though the shadow stack pages are non-writable, the call instruction is allowed to write to these pages.

If for some reason a certain memory page is mistakenly configured like a shadow stack page, an attacker can direct the SSP to this memory page and use the control flow instructions to write it. We now show that this is a plausible scenario.

Lets consider the code segment of a program. This memory region is first mapped as writable, then it is filled with the code and finally marked as non-writable. This means that it is likely that the leaf page table entry R/W bit is set to 0 and that the

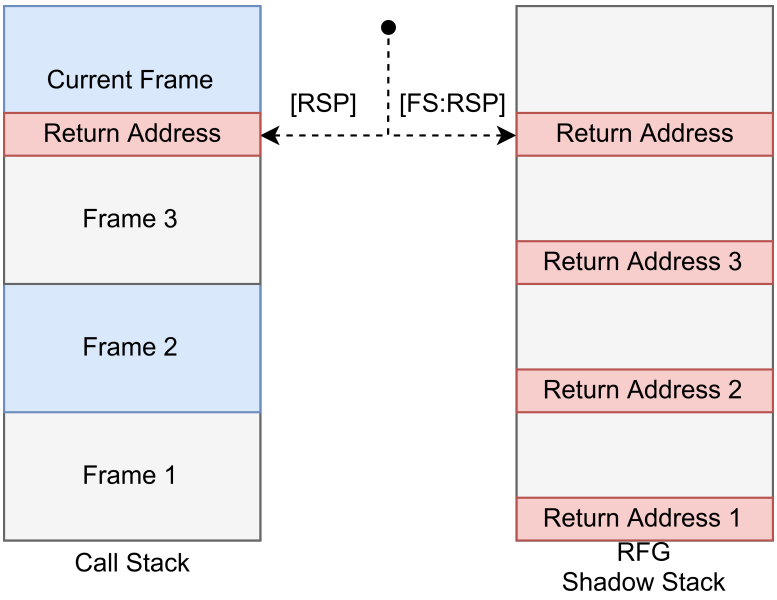


Figure 3.2: The RFG Shadow Stack

dirty bit is set to 1 (unless intentionally set to 0). The non-leaf page table entires R/W bits are likely to be set to 1, as the kernel needs to write to these pages. To summarize, we can see that unless carefully built the code segment can be configured as a shadow stack region and thus writable by the shadow stack push command.

CET enables to allocate s different shadow stack for each CPU privilege level (ring). The CPU automatically switches shadow stacks on privilege level changes. Additionally, instructions for shadow stack switching are defined by CET, which can be used by the operating system to switch shadow stacks on context switches.

Finally, it is interesting to note that although Intel has released a preview of the control flow enforcement technology, it has not presented a time line for its deployment.

3.4.2 Microsoft Return Flow Guard

Microsoft Return Flow Guard [31] (RFG) is an experimental software implementation of a shadow stack. In RFG the shadow stack resides in the process memory, and the shadow stack push and shadow stack pop commands are automatically added by the compiler. When configured to support RFG, the compiler adds a shadow stack push instruction to every procedure prologue, and adds a verification and a shadow stack pop before the ret instruction. Unlike in CET, the RFG shadow stack contains not only a copy of the return addresses, but also place holders to all other data that is stored on the stack, as illustrated in Figure 3.2. This stack structure enables to access the shadow stack using the stack pointer (RSP) and a dedicated segment selector (i.e., FS:RSP). By using RSP there is no need to manage a separated pointer for the shadow stack, which improve the performance.

The way RFG is implemented is inherently flawed. Any reasonable threat model

assumes that the attacker can write to the process memory. In many cases, the attacker can write to arbitrary memory locations. For example, a format string vulnerability typically allows an attacker to write to arbitrary memory locations. Therefore, it is very likely that the attacker is also able to overwrite the shadow stack, in which case he renders it useless.

Therefore, it is not surprising that Itkin [29] published an attack that can bypass RFG. In his attacks he used a write primitive that can simultaneously overwrite both the stack and the shadow stack, thus nullifying the shadow stack protection. We should mention that Itkin's attack also provides a technique to locate the shadow stack in the process memory.

In parallel, Microsoft announced that RFG is discontinued due to issues found by its red team.

Chapter 4

TROOS – Trusted Open Operating System

In this chapter we present *TROOS: TRusted Open Operating System*. TROOS is an OS for TrustZone-based TEEs, with a minimal attack surface. TROOS provides strong isolation for its trusted application. Moreover, trusted applications running on TROOS enjoy a very small TCB.

The rest of this chapter is organized as follows: Section 4.1 describes our threat model. Section 4.2 discusses the security level of various TrustZone-based TEEs. Section 4.3 presents the TROOS architecture and discuss its advantages, and finally, Section 4.4 describes our prototype implementation.

4.1 The Threat Model

The threat model of TROOS assumes that the attacker has physical access to the device, advanced reverse engineering abilities and sophisticated lab equipment. In particular, the model assumes that the attacker has unlimited access to the system's non-volatile storage devices, and that he is able to probe the system's main memory. The model further assumes that the attacker has complete control over the MEE, and that he is trying to mount an attack from within the MEE against any TEE asset.

However, the attacker is limited by the SoC boundaries, as he is unable to penetrate them. Hence, the attacker cannot access the internal SoC content. Storage devices that are physically placed within the SoC boundary (if such devices exist) are out of the attacker's reach.

On the tools and equipment level, the threat model assumes that the attacker has access to various TA and TEE kernel versions (the attacker can even get hold of versions that were not provided with the system). These kernel and TA versions are legitimate, i.e., they were produced by a certified entity, but may contain known unpatched vulnerabilities.

The model also assumes that the attacker may have established a partial foothold

within the TEE by exploiting another TA than the one he is targeting. In this scenario the attacker may be able to run arbitrary unprivileged code inside the TEE, and use it to interact with TEE components via the TEE internal API.

Recall that the main justification for using a TEE is because the MEE cannot be trusted. Therefore, it is reasonable to assume that the MEE is under the attacker's control. Moreover, in many cases the attacker is the owner of the device (like in the secure media player example in Section 2.2), hence he can easily access the non-volatile storage.

We claim that this threat model is realistic since it includes all the techniques and vulnerabilities that a sophisticated attacker is expected to use, and leaves out threats that give the attacker too much power, which in practice he is not expected to have.

4.2 The Security of TrustZone-Based TEEs

In this section we discuss the security level of some of the most widely deployed TrustZone-based TEEs:

- Qualcomm's QSEE.
- Trustonic's Kinibi.
- Linaro's OP-TEE.

We focus our discussion on weaknesses and vulnerabilities regarding the TEEs attack surface and TCB, since they might jeopardize the entire system. Some of these findings are ours and some were already published by other researchers.

A slightly different approach for building a TrustZone based TEE is presented in Komodo [16]: TrustZone is used to create *secure-enclaves*, and the API provides a very small set of command that are used to create these enclaves resulting in a very lean implementation. In particular, Komodo does not use a trusted OS running in TrustZone. The enclaves are managed by the MEE OS, and cannot use in secure OS request. Komodo's architecture is meant to imitate Intel's SGX architecture. A big advantage of Komodo is that its implementation is formally verified. However, the lack of secure OS presents limitation to the use cases Komodo can support, and it does not fully utilizes TrustZone features. In particular, the enclaves cannot use the services of secure devices, and no secure interrupts can be implemented.

For the purpose of this work we focus only on TEEs that use their own OS.

4.2.1 Deployment of Control Flow Hijacking Mitigations

Experience shows us that any computer system might contain vulnerabilities. Such vulnerabilities can be exploited by attackers, and put the system at risk. Mitigations, such as stack canaries and ASLR, can be utilized in order to prevent attackers from

Name		NX	ASLR	Canaries	Guard Pages
QSEE	Kernel	✓	✗	✓	✗
	TAs	✓	✓	✓	✗
Kinibi	Kernel	✓	✗	✗	✗
	TAs	✓	✗	✗	✗
OP-TEE	Kernel	✓	✗	✗	✗
	TAs	✓	✗	✗	✗

Table 4.1: TrustZone-Based TEEs Vulnerabilities Mitigations Summary

successfully exploiting vulnerabilities. Therefore, vulnerability mitigations increase the system's security, and decrease its attack surface.

A particularly important group of vulnerability mitigations is used to prevent control flow hijacking. The importance of these mitigations is due to the fact that such vulnerabilities are widespread and relatively easy to exploit. Additionally, the potential impact of control flow hijacking is devastating, as it enables an attacker to take over the system. We focus our discussion on the major control flow hijacking mitigations: NX, ASLR, stack canaries and guard pages.

Naturally, one might expect from a security oriented system, such as a TEE, to provide the highest level of security, and to deploy every possible vulnerability mitigations. This expectation is especially important in the case of control flow hijacking. Unfortunately common TEEs are not protected by all these mitigations.

The OP-TEE kernel does not support stack canaries, ASLR or guard pages. The only protection against control flow hijacking in the OP-TEE kernel is NX. Moreover, the OP-TEE TAs also suffer from the same problems as the kernel. Kinibi status is similar to the status of OP-TEE, i.e., neither Kinibi kernel nor it TAs utilize any mitigation other than NX.

QSEE does a better job than OP-TEE and Kinibi in utilizing mitigations. In addition to NX, QSEE TAs also enjoy the protection of stack canaries and ASLR, but guard pages are not used. Even though QSEE supports ASLR, it does not implement it in an effective manner: QSEE uses a flat memory mapping scheme (i.e., the virtual addresses are equal to the physical addresses). As the physical memory region assigned to QSEE is roughly 100MB (i.e., at most 15 bits of randomness), its ASLR does not have sufficient entropy to be effective.[1] On the other hand, The QSEE kernel supports an even smaller set of mitigations, as unlike its TAs, it does not support ASLR.

A summary of these findings is presented in Table 4.1.

[1]Even if we assume small 4KB pages, and assume that each page can be randomized, and assume that the memory size is 128MB, we only get $128MB/4KB = 2^{15}$ locations of pages. Therefore, we only have 15 bits of entropy.

4.2.2 Least Privileges Principle Usage

In the context of secure systems, a privilege is the ability to preform certain operations on specific objects, typically security sensitive operations. The privileges are determined by the system's security policy. For example, in Linux not every process can change the internal firewall settings, since this operation requires a special privilege.

The *least privileges principle* [53] states that a computer program should have the minimal set of privileges necessary to complete its task. When this principle is applied, a successful exploitation of a program has the minimal impact on the system.

In the context of an OS, the privileges are the permission to use services provided by the OS. Therefore, by the principle of least privileges the OS should provide a service to a program only if it is necessary for it to complete its task, i.e., not every program can access every OS service. This principle improves the OS security as it limits the interface exposed to certain programs. In other words, an OS that applies the principle of least privileges reduces its attack surface.

A TA can ask the trusted OS for services, e.g., access to securely stored keys, device operations or memory mapping manipulations. By the principle of least privileges, a TA should have access only to the services necessary for its task, i.e., not every TA can access every service. For example, a Bitcoin wallet TA may require cryptographic services, but may not directly access the screen buffer. On the other hand, a secure media player TA typically needs to access the screen, while certainly it is not allowed to access the Bitcoin keys.

Unfortunately, not even a single TrustZone-based TEE utilizes the least privileges principle. Both OP-TEE and QSEE offer more than 150 services to the TAs, but in both any running TA can access any service the OS provides (regardless of whether it is necessary for it to complete its task). The services themselves typically enforce some kind of restrictions. For example, any TA can call the file system's "read file" service, but access is granted according to the requested file permissions.

Kinibi takes a step towards complying with the least privileges principle, by providing the trusted OS services drivers [12]. Indeed, well designed drivers can restrict TAs from preforming certain operations. Nonetheless, it is preferred that the principles are enforced centrally by the trusted OS, rather than being patched in small pieces all over the system.

4.2.3 TAs Revocation in TEEs

The fact that TAs are designed with security in mind does not automatically ensure that they are immune to attacks. An attacker may locate a vulnerability in a TA, and use it to exploit the TA, or even the TEE itself. The impact of such an exploitation depends on the vulnerability and the TA implementation. It may lead to a minor information leak in some cases, but if we are not fortunate enough it may also lead to full blown arbitrary code execution within the TA context.

Once a TA is exploited, the damage is not necessarily limited to the exploited TA. The attacker can try to use his foothold in the TA to mount an attack on the TEE kernel, thus putting the entire TEE at risk. To make things worse, as shown above, security mitigations are not widely deployed in TEEs. Therefore, once a vulnerability in a TA is found, it can probably be exploited and used to launch further, more dangerous, attacks.

Typically, after a TA vulnerability is made public, the TA owner fixes it, and issues a patched version of the TA. Naturally, we expect the TEE to update the TA to its latest version, and even to deny the usage of any older TA versions that are already known to be vulnerable. Hence, the TEE should support a mechanism to revoke TAs.

We suspect that the reader is not surprised by the fact that neither QSEE, Kinibi nor OP-TEE provide proper support for TA revocation. This fact by itself is a crucial threat as it can be used to bypass all other security features and mitigations, and leave the system totally unprotected.

In [12] Beniamini shows how to exploit QSEE by using a legitimate but already known to be vulnerable TA. In his attack Beniamini used a two years old vulnerable version of the Widevine TA. This vulnerable version contains a vulnerability, that allows arbitrary code execution [41]. Although this version was already known to be vulnerable, and although a patched version of the the TA was already issued, the old vulnerable TA was still successfully loaded to QSEE.

In the same work, Beniamini used the same technique to investigate Kinibi TAs revocation status. He found that Kinibi versions before 400 suffered from the same problem. However, the same attack fails on the Kinibi 400 family. According to Beniamini, an eMMC feature called *replay protected memory block* [59] (RPMB) was utilized to implement the revocation mechanism.

After examining the OP-TEE code and documentation we conclude that it does not have any support for TA revocation. However, we found that as a part of its secure storage implementation OP-TEE defines an RPMB interface. This RPMB interface can be used (like in Kinibi 400) as a basis for a future TA revocation mechanism, and we hope that the implementors will indeed implement such a mechanism.

4.3 The TROOS Architecture

We designed TROOS in light of the security gaps we have just discussed. The leading design concern is to reduce the TEE's attack surface without losing any of its functionality.

It is not easy to design an OS, and even more so a security oriented OS. In particular, there is a constant need for compromises between the system's security requirements and other important requirements of the system. For example, one of the standard IPC mechanisms is a shared memory region. By defining a shared memory region, two processes can communicate with each other very efficiently and transfer large amounts

of data with low overhead. On the other hand, a shared memory region breaks the process isolation boundary, and hence it has a negative impact on the system security. The decision whether to allow two processes to share memory is a compromise between the system's security and its performance.

In order to keep security as our top priority, we need to define guidelines that can lead us to make security aware decisions. Therefore, we decided that whenever there is a trade-off between security and other requirements, TROOS follows the following principles:

1. **Least Privileges** Every component in the system should have the minimal set of rights it needs to complete its task.

2. **Strong Isolation** Component boundaries must be maintained. We do not allow solutions that violate component isolation (e.g., shared memory).

3. **Minimal TCB** We refrain from adding features that unnecessarily enlarge the TCB (even at the cost of forfeiting certain features).

4. **KISS** As long as it is secure, we use the simplest solution available (even if it is not optimal in terms of performance, memory capacity, etc.).

5. **Preformance** We are willing to sacrifice performance in order to achieve better security. However, we inspire to have minimal performance impact.

The first decision we had to make is whether to take the monolithic kernel or the micro-kernel approach. Needless to say that we are aware of the long lasting debate regarding the security of monolithic kernels vs micro-kernels [61, 62]. However, we feel that a micro-kernel better suits our guidelines. Therefore, we opted to build TROOS based on a micro-kernel. Further considerations follow with more details.

4.3.1 TROOS System Components

As a micro-kernel based OS, most of TROOS services are provided by user mode components. In TROOS, these components are implemented as user mode processes, and called the *system components*. The system components include the following components:

1. **Dispatcher** Manages the communication with the MEE.

2. **TAs Manager** Creates and destroys TAs.

3. **Crypto** Provides cryptographic services.

The system components also include a kernel mode component. Figure 4.1 is a block digram that outlines the interactions between the various system components of TROOS.

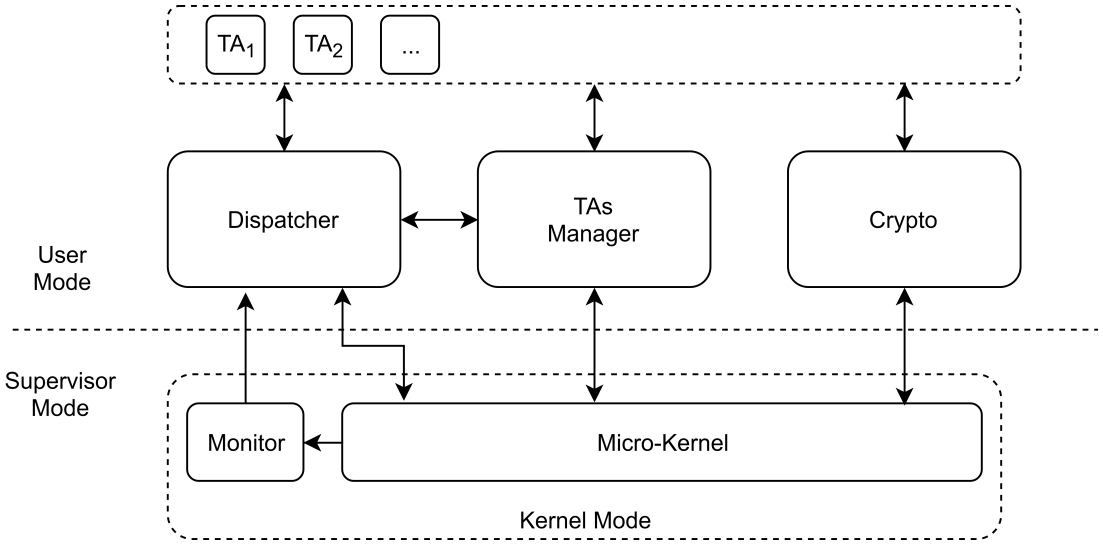


Figure 4.1: TROOS Block Diagram

Each of these components is running in a different process, therefore the system components are isolated from each other by the micro-kernel process isolation mechanisms. The system components use TROOS internal API to communicate with each other, and system calls to interact with the micro-kernel. The micro-kernel utilizes the least privileges principle, and restricts the services it offers to the system components. For example, only the TAs manager is allowed to assign new memory mappings.

If it possess the proper capability, a TA can use the TROOS internal API to interact with a system component. However, TAs are not allowed to directly interact with the micro-kernel, nor with other TAs. The micro-kernel interacts only with the system components, which provide the necessary services to the TAs.

4.3.1.1 The Kernel Mode Component

The only component with supervisor privileges is the kernel mode component. It consists of two modules: the micro-kernel and the monitor mode code (referred to as the monitor).

The micro-kernel provides the basic OS services, such as scheduling, memory management and inter-process communication. In order to be able to preform its task, the micro-kernel must run in an elevated privileged level.

The monitor intercepts the SMC instructions and handles the interaction between the MEE and the TEE. The monitor is responsible to save the state of the caller and to transfer the control to the callee. When TROOS is called, the monitor always transfers control to the dispatcher, which is an unprivileged system component that analyzes the request and then passes it to its target.

4.3.1.2 The Dispatcher Component

The dispatcher component receives the MEE requests from the monitor and transfers them to the relevant system component by using TROOS internal API, i.e., by invoking an RPC in the target component. For example, a request to destroy a TA is passed to the TAs manager by invoking the corresponding TAs manager RPC, while an I/O request to a TA is passed directly to the target TA by a write RPC in the target TA.

4.3.1.3 The TAs Manager Component

The TA manager is responsible for the creation and destruction of TAs. To create a new TA, the TA manager creates a new sub-process and loads the TA executable to that process, after properly verifying it. Like the system services, the TAs are isolated from each other by the OS isolation mechanisms. Upon TA creation, the TAs manager assigns to the newly created process capabilities according to the TA manifest. These capabilities remain fixed for to entire process lifetime.

In order to be able to complete its task the TAs manager is allowed to ask the micro-kernel to add new virtual address mappings for the TA it is creating (in order to map the shared I/O buffer) and for itself (in order to map the buffers used in the TA creation request). It is the only system component with that privilege.

4.3.1.4 The Crypto Component

The Crypto component provides the TAs with various cryptographic services, such as:

- Cryptographic keys.
- Remote attestation.
- Random number generator.
- Monotonic counter.

To meet the minimal interface approach, the Crypto component offers only the bear minimum of services required. In particular, it does not provide encryption or signature services (these can be implemented in a user library added to the TA).

4.3.1.4.1 The Cryptographic Keys Service Provides a set of unique cryptographic keys to each TA. Every TA has set of cryptographic keys accessible only by itself. The keys are only available to the TA while it is running in the TEE. The keys set is platform dependent, hence it is different on different devices.

These keys can be used by the TA for various cryptographic purposes. For example, a TA can use the keys provided by the Crypto component to create its own secure storage, by creating keys that can only be obtained only by it and only from within the TEE. These keys can be used to encrypt the stored data, and in that way only the TA can decrypt them.

4.3.1.4.2 The Remote Attestation Service Provides remote attestation services, which enable TAs to prove their state to a remote party. The attestation service provides a signed report that describes the state of system and the TA. This report includes (but not limited to) the following information: the measurement of the TA code (at the time of the TA creation) and the TROOS version number.

The attestation report is created and filled by the attestation service, and the TA has no control over its content. To prevent forgery, the attestation service uses a platform specific key, such as the TPM's endorsement key [63], to sign the report (after utilizing a proper privacy protection scheme).

Once the report is received by a remote party, the remote party can verify the report using the public part of key, in order to make sure that is was created by a genuine TA running under TROOS.[2]

4.3.2 Scheduling

In a TrustZone-based TEE, the TEE and the MEE share the same CPU, and must operate in a time sharing manner. This means that some scheduling policy between the two environments must be applied.

TROOS does not interrupt the MEE OS while it is running. When the MEE is controlling the CPU, it uses its OS scheduler to manage its processes. At any time it may choose to pass the control to TROOS (using the TROOS API) in order to use a trusted service. Once TROOS receives control over the CPU, it keeps it until it finishes to handle the service request. TROOS also operates with masked interrupts, i.e., any interrupt received while TROOS is running is ignored by the TEE, and is later served by the MEE. This means that from the point of view of the MEE OS, any trusted service request is blocking.

Recall that when using TrustZone, two operating systems are running in parallel. Each of these OSes has its own scheduler for its processes. TROOS has no specific requirements from the the MEE OS scheduler, and it may run alongside the MEE regardless of its OS processes scheduling policy, with the MEE scheduler controlling the switch to TROOS. In particular, the MEE may never pass the control to TROOS (i.e., DoS is possible).

For the purpose of this work, TROOS uses a non-preemptive scheduler. Any service request is handled without being stopped until it is completed, and only then the control is passed back to the MEE. This is merely an arbitrary choice for simplicity. TROOS could easily use a more complex scheduler without any significant design changes.

4.3.3 TROOS External API (MEE $\Leftrightarrow$ TEE)

In order to keep the TEE attack surface as small as possible, we wish to minimize the interface between the TEE and the MEE. At the same time, we want to keep our TEE

[2]The PKI necessary for remote attestation is out of scope for this work.

Command	Description	Return Value
TCREATE(*exec*, *mani*, *cert*, *io buffer*, *io buffer size*)		
	Creates a new TA.	TAID
TDESTROY(*taid*)	Destroys an existing TA	Status
TWRITE(*taid*, *n*, *cmd*)	Notifies the TA that data was written to its I/O buffer	#Bytes written
TREAD(*taid*, *n*, *cmd*)	Notifies the TA that data was read from its I/O buffer	#Bytes to read

Table 4.2: TROOS External API

useful and versatile, and have an external API that supports the most common use cases.

We chose not to follow the GlobalPlatform Client API, since it is too broad for our purposes, and contains interfaces which unnecessarily enlarge the TEE attack surface. For example, the GlobalPlatform API exposes a few commands to create sessions between a TA and its client application. While sessions may be useful in some cases, they are not necessary in others. Therefore, we think that only TAs that require sessions should expose such interfaces rather than make it a part of the TEE API.

Another example for common TEE interface that we chose not to include in TROOS API is firmware update. Many TEE offer a secure service which enables the MEE to update secure firmware (after properly authenticating it). TROOS does not provide such an interface. However, if firmware update service is needed it can be implemented by a dedicated system component, or preferably a dedicated TA.

In this subsection we propose an external API consists of only four commands, which meets our requirements. This API is presented in Table 4.2, it includes a command for the creation of TAs, a command for the destruction of TAs, and two commands for I/O operations. The following subsubsections provide a detailed description of the external API.

4.3.3.1 TCREATE

The TCREATE command creates a new TA. The MEE provides the TA code and initial data in a single file called the *TA executable*. The TCREATE command receives all of its arguments via memory buffers. These memory buffers are untrusted since they are shared with the MEE. Hence TROOS first operation is to copy the input buffers into its own internal buffers. Then, the TA executable is measured and authenticated using the provided manifest and certificate. Trust is then gained through these measurements authentication. These measurements are also used in the remote attestation reports and enable remote parties to know exactly who they are communicating with.

If the TA executable is successfully authenticated, the TA manager creates a new subprocess for the TA. TROOS maps the I/O buffer to the TA memory space and assigns a TAID to the subprocess. The new TA capabilities are set according to the manifest.

The TCREATE parameters are:

1. **TA Executable:** A pointer to a buffer containing the TA executable file.
2. **Manifest:** A pointer to a buffer containing the TA manifest.
3. **Cert:** A pointer to a buffer containing a certificate for the manifest signing key.
4. **I/O buffer:** The physical address of the shared I/O buffer. The TEE OS maps this address to the TA memory space.
5. **I/O buffer size:** The size of the I/O buffer to be mapped.

Return Value: Upon success returned is the new TAID. A negative return value indicates an error.

4.3.3.2 TDESTROY

The TDESTROY commands destroys an existing TA. The MEE can decide that it no longer requires the services of a certain TA, e.g., when the MEE process that used the TA services exits. In such a case, the OS should destroy the TA in order to free the resources of the TA. Another reason to destroy unnecessary TAs is that they enlarge the TEE attack surface, while they no longer provide any useful services.

Once invoked, the TDESTROY command stops the TA execution, kills the relevant TROOS process and frees its resources.

The TDESTROY parameters are:

1. **TAID:** The TA to destroy.

Return Value: Upon success a non-negative return value is returned. A negative return value indicates an error.

4.3.3.3 TWRITE

The TWRITE command notifies a TA that new data has been written to its I/O buffer by the MEE. It is used by the MEE by writing n bytes to the TA I/O buffer (which is stored in shared memory), and then calling the TWRITE API command to notify the TA that the new data is available. Upon receiving the command, the TA attempts to read n bytes from the I/O buffer.

The TWRITE parameters are:

1. **TAID:** The TA to be notified.
2. **N:** The number of bytes written to the buffer.
3. **Cmd:** An auxiliary argument. Can be used by the TA to determine how to handle the new data.

Return Value: Upon success the return value is the number of bytes (zero or positive) read from the I/O buffer by the TA. A negative return value indicates an error.

4.3.3.4 TREAD

The TREAD commands notifies a TA that its client application wishes to read from the I/O buffer. The TA writes data to the I/O buffer and passes the control back to the MEE.

The TREAD parameters are:

1. **TAID:** The TA to be notified.
2. **N:** The maximum number of bytes to read from the buffer.
3. **Cmd:** An auxiliary argument. Can be used by the TA to determine how to handle request.

Return Value: Upon success the return value is the number of bytes (zero or positive) written to the I/O buffer by the TA, i.e, the number of bytes ready to be read by the client application. A negative return value indicates an error.

4.3.3.5 TWRITE and TREAD Usage Example

Consider a TA that provides access to a secure RNG device. Such a TA typically offers various trusted services, such as random data generation and device configuration operations. For the purpose of this example we focus on one specific service provided by the TA, which we name `get_random_bytes(n)`. This service uses the RNG device to generate n bytes of random data, and returns them to the MEE.

In our example, a client application wishes to get n random bytes by calling `get_random_bytes(`n`)`. We assume that the TA already runs, and that it is waiting to serve its client applications.

A client application first writes n to the shared I/O buffer, and then uses the TWRITE interface with a `cmd` argument which indicates that the `get_random_bytes` service is called.

The TWRITE command passes control to the TA, which then parses the request and identifies that `get_random_bytes` is called. It then reads n from the I/O buffer, asks the RNG device to produce random data, and passes the control back to the MEE.

After the successful completion of TWRITE the client application uses TREAD, and passes control again to the TA, which writes the random data it received from the device to the shared buffer, and then passes control back to the MEE. Now the client application can read the random data from the shared buffer.

The `get_random_bytes` service could have been implemented using a single API command, which blocks the MEE until the random data is generated. By splitting the service to two API command, we allow the MEE to regain control while the RNG device is producing the random data.

Listing 4.1 and Listing 4.2 provide a pseudo-code description of the TA and client application respectively.

```
 0. Wait for service request
 3. Parse service request
 4. Read n from the shared buffer
 5. Ask the RNG device to generate
    n bytes
 6. Wait for service request
 9. Parse service request
10. Write random data to the
    shared buffer
11. Wait for service request
```

Listing 4.1: TWRITE and TREAD Usage Example: The TA

```
 1. Write n to the shared buffer
 2. r := TWRITE(taid, sizeof(n),
             #get_random_bytes)
 7. If r == error then: handle error
 8. r := TREAD(taid,n,
            #get_random_bytes)
12. If r == error then: handle error
13. Read n random bytes
    from the buffer
```

Listing 4.2: TWRITE and TREAD Usage Example: The Client Application

4.3.4 TROOS Internal API (TAs ⇔ System Components)

TROOS internal API follows a client-server model. A service request is submitted by triggering an RPC in the component that offers the desired service. The component that triggers the RPC request plays the role of the client, and the component that executes the procedure plays the role of the server. A component may offer and receive services, therefore it may act both as a server and as a client.

The client is eligible to receive a service from the server only if it possesses the relevant capability to access the particular server. The enforcement of the capabilities model is done by the micro-kernel component as a part of the IPC implementation.

For example, consider a Bitcoin wallet TA. A Bitcoin transaction is valid only if it is signed by the coin ownership key. The Bitcoin wallet TA is responsible to sign the transaction. The TA can trigger an RPC in the Crypto component, which allows it to access the coin ownership key. The TA then signs the transaction with this key. The TA should have the proper capability for calling the service that provides access to the signature keys.

TROOS internal API does not provide direct communication between different TAs. TAs that wish to interact with other TAs, must do so via the MME. Although the indirect communication may have a negative effect on performance, by not allowing direct communication we simplify TROOS, eliminate unnecessary APIs. In particular, we avoid sharing memory between TAs. However, the TAs may interact with each other via the MEE, thus the functionally of TROOS is not effected, and TROOS can support every TA supported by other TrustZone-based TEEs (with proper adjustments).

4.3.5 The Advantages of the TROOS Architecture

In this subsection we show how the TROOS architecture mitigates attacks on the TEE, by analyzing a real life attack. Our analysis shows how the TROOS API significantly limits the attacker ability to attack the TEE.

We chose to demonstrate TROOS's API advantages through the attack of Beniamini in [11], which we already discussed. Beniamini showed how a local unprivileged MEE user can exploit his way to full control over the QSEE kernel. This methodical attack advances step by step until achieving its final goal.

The first step of the attack is to exploit a vulnerability in the Linux kernel in order to freely interact with QSEE, then a vulnerable TA is exploited in order to gain initial foothold in QSEE. Finally, the exploited TA is used to exploit the QSEE kernel. For the purpose of this discussion we are only interested in the last two steps.

4.3.5.1 Exploiting the TA

In this step of the attack Beniamini [11] exploits a buffer overflow vulnerability in the Widevine DRM TA [69]. This vulnerability allows the attacker to copy an arbitrary size buffer to a global buffer in the TA memory, thus enabling him to overwrite any memory location after this global buffer. Once the attacker has established this write primitive he uses ROP to hijack the TA control flow and to gain full control over the TA. Although it is out of our scope to fully describe the exploiting techniques used in this attack, we would like to mention that they are extremely sophisticated.

To start the ROP chain execution, the attacker replaces a data structure, called *session*, used by the TA with a "poisoned" one by overwriting its pointer. In order to do so, the attacker needs to allocate the poisoned data structure in the TA memory.

The TA's writable memory areas are craved out of a special memory region called *secapp*. TrustZone isolates secapp from the MEE, however QSEE TAs have unlimited access to all allocated memory pages within that region, i.e., every TA can access every allocated page within secapp. This fact is used by the attacker to find a valid memory location for his poisoned session. This memory locations is located by using a global read primitive, which is able to read data from every allocated secapp page even outside of the TA context.

In TROOS, the buffer overflow vulnerability introduced by the Widevine DRM TA may still exist, and might be exploited by the attacker to overwrite the session pointer. However, the TROOS architecture does not allow TAs to share memory, and therefore the technique used by the attacker to create the poisoned session would have failed.

4.3.5.2 Exploiting the Kernel

In this step of Beniamini's attack an unverified write address vulnerability in the QSEE kernel is exploited. This vulnerability is found in one of the 69 (!) QSEE kernel

API commands exposed to the MEE. The exploit allows the attacker to write zero to arbitrary kernel memory locations (a.k.a write-zero primitive).

The write-zero primitive is then cleverly extended to an arbitrary write primitive, which is then used to inject code to the QSEE kernel (after bypassing its read-only memory protection). In order to do so, two more API commands are used.

To put it all together, the attacker is able to locate a vulnerable API command, and then by combing it with two more API commands, he mounts an attack on the QSEE kernel and takes over the TEE.

This attack is clearly caused by the high complexity of the QSEE API. In particular, the large number of API commands exposed to the MEE makes it much harder to verify that they do not contain vulnerabilities. Moreover, as demonstrated in this attack, the abundance of API commands provides the attacker more options to exploit the vulnerabilities he finds.

On the other hand, TROOS has a very simple API with only four commands. Therefore, it is easier to verify that it does not contain vulnerabilities. Moreover, even if there is a vulnerability in one of the API commands, it is harder to exploit it, since the attacker does not have a rich selection of API commands to utilize in his attack.

4.4 Implementation Details

Our goal was to design a real and usable OS for TEEs, and we did it by implementing a prototype of TROOS. Our prototype includes a top to bottom TEE implementation that complies with the TROOS architecture, a couple of example TAs, and an MEE driver that interacts with TROOS.

For obvious reasons we chose not to implement the kernel from scratch. Instead, we used an existing kernel that suits our needs. Since according to the TROOS architecture the kernel mode component is based on a micro-kernel, we needed to use a micro-kernel rather than a monolithic kernel.

In order to be able to comply with the TROOS architecture, the OS kernel should support the following properties:

- Process isolation.
- Inter-process communication with fine grained access control.
- Minimal application TCB.

When we checked the alternatives, we observed that the Genode OS framework meets all of these requirements, while it also has native TrustZone support.

The Genode security model is capability driven. It provides strong process isolation and enables us to easily create a set of capabilities for each TA. Moreover, TROOS requires RPCs for its internal API, and Genode supports RPC as an IPC mechanism. Finally, Genode enables its applications to define their TCB by the services they are

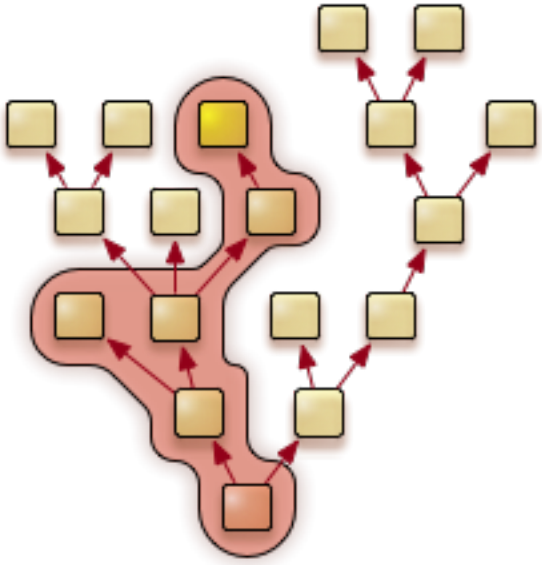


Figure 4.2: The TCB of a Genode Application. The TCB of the yellow sandbox is marked in red. The TCB includes all of the sandbox's ancestors, and an additional sandbox it interacts with. The rest of the system is not included in the TCB (figure taken from [40]).

using, i.e., a Genode application only needs to include in its TCB the system components it is actually interacting with.

For these reasons we opted to use Genode as a kernel for TROOS.

4.4.1 Genode OS Framework

The *Genode OS Framework* [20] is a toolkit for building highly secure special-purpose operating systems. The system is built in a recursive manner: each application is executed in its own sandbox with the bare minimum of access rights needed to complete its task. Any application can create and manage sub-sandboxes, which use a subset of its own resources. Genode provides mechanisms for communication between sandboxes, these mechanisms are controlled by the system's security policy which restricts the interactions between the sandboxes.

The Genode OS is micro-kernel based (a variation of L4 [34]). The framework provides many building blocks (such as device drivers, file systems and network stacks), which allow users to quickly build sophisticated complex system. However, due to its recursive structure, Genode is able to keep a small TCB for its applications, even when the system itself is very complex. In Genode, the TCB of an application includes all of its ancestors as well as the sandboxes it is interacting with (along with their ancestors as well). Other sandboxes are not included in the application's TCB. This concept is illustrated in Figure 4.2.

The Genode security policy is capability driven. In Genode, an application may be granted capabilities to interact with other applications in various manners. This

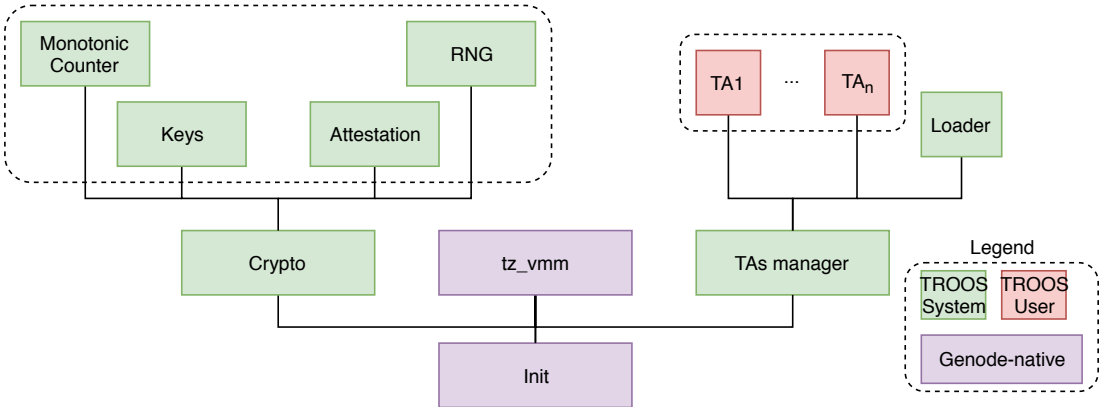


Figure 4.3: TROOS Components Under Genode

security model nicely suits the needs of a TEE, as it allows fine grained control over each trusted application ability to interact with other components.

4.4.2 Implementing TROOS System Components with Genode

The first thing we need to clarify in order to continue with our discussion regards to our terminology. We named the different parts of TROOS *components*. Genode terminology also uses the name components, but with a slightly different meaning. In Genode a component is basically a process, i.e., this is basic execution unit of the OS. For clarity, the term components in this subsection refers to TROOS system components. When addressing Genode components, we will explicitly say "Genode Components". [3]

In our prototype each system component and each TA is implemented in a separate Genode component (i.e., a different Genode process). These components interact with each other using Genode's native RPC API. The system structure is outlined in Figure 4.3. Genode provided us the basic OS services, and we built our system's components based on these services. To clarify which parts of the system are Genode-native and which are ours different colors are used in Figure 4.3.

4.4.2.1 The TAs Manager

The TAs manager is the parent of all the TAs. It delegates capabilities to each TA according to the TA manifest, and controls every service request made by the TAs.

In order to create the TAs the TAs manager uses an auxiliary component called the *loader*. The loader is a native Genode component, which creates new Genode components from executables. The TAs resources are taken from the TAs manager budget, and are disconnected from the loader after the creation process is finished.

[3] The Genode components are user mode processes.

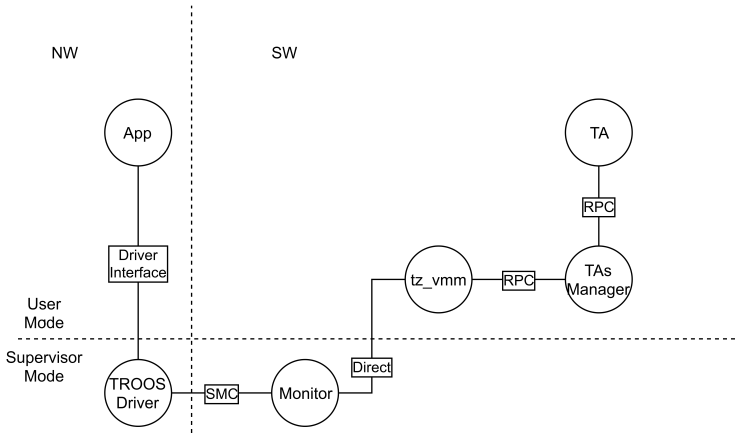


Figure 4.4: Handle SMC Flow

4.4.2.2 The TZ_VMM Component

The TZ_VMM component is the native Genode component that implements TrustZone support. It interacts with the monitor and receives the MEE requests. We use the TZ_VMM component to implement the dispatcher.

Recall that an SMC instruction causes an exception which is handled by the monitor. The monitor parses the SMC arguments and passes the control to the TZ_VMM component.

Although the TROOS architecture allows the dispatcher to directly interact with the TAs, this is not the case in our prototype. Instead the TZ_VMM component passes the request to the TAs manager which in turn passes the request to the relevant TA (if needed). This flow is outlined in Figure 4.4.

4.4.3 Utilizing the Genode Architecture

Various Genode features are utilized in order to realize the concepts specified in the TROOS architecture. The following subsubsections describe how the Genode architecture is leveraged to achieve compliance with TROOS architecture.

4.4.3.1 Components Isolation

In Genode each component operates in its own *protection domain*. These protection domains consist of a virtual address space and a capability space. Protection domains are isolated from each other, and can interact with each other only by sharing memory or by triggering RPCs. However, the TA manager does not allow its TAs to share memory. Moreover, TAs are not allowed to directly interact with Genode.

As a result, the desired strong isolation is achieved. Moreover, we are ensured that all components obey the interaction requirements of the TROOS architecture (as presented in Figure 4.1).

4.4.3.2 TA Capabilities

Recall that Genode has a native RPC mechanism, which allows a component to offer services to other components. Moreover, in order to trigger an RPC the client must have a capability allowing it to access the service.

In our prototype, the TAs manager poses capabilities to every service in the system, and delegates these capabilities to the TAs according to their manifest. Therefore, TAs may only access services allowed by their manifest.

4.4.3.3 Minimizing the TAs TCB

Recall that in Genode the TCB of an application includes all of its ancestors. Moreover, the recursive system structure allows each application to decide which additional parts of the system are included in its TCB.

Thanks to the system structure (as presented in Figure 4.3) a TA running under TROOS is not forced to include the entire system in its TCB. Instead, the TCB of a TA consists only the the kernel, the TA manager and the TA itself. However, if the TA wishes to use a service provided by a system component this component is also included in the TA TCB.

Notice that the dispatcher component (i.e., the TZ_VMM component) is not part of the TA's TCB. The requests from the MEE are considered untrusted, and the dispatcher only transfers them to the TAs. Therefore, the dispatcher only has the power to control untrusted data.

4.4.4 The TROOS MEE Driver

As a part of our TROOS prototype, we implemented a simple MEE driver for Linux that supports the interface to the TROOS trusted kernel. The driver, which is implemented as a char device, enables to interact with the external API of TROOS, and supports all of the four TROOS API commands.

The driver supports the API commands using the following technologies: the TREAD and TWRITE commands are implemented as the device's read and write operations. The TCREATE and TDESTROY commands are provided via the driver's `iotcl` interface.

Upon every TA creation request the driver allocates a new buffer to serve as the TA's shared I/O buffer, and links it to the file descriptor used to issue the request. When the TEE creates the new TA, it maps the buffer to the TA's address space. The state of the shared I/O buffer after a successful TA creation is outlined in Figure 4.5. Once the TA is created, the client application can use the driver's read and write interfaces to communicate with the TA.

The read and write interfaces allow only sequential transfer of data, which is not always the most convenient way to communicate. However, random memory access can be achieved by using the driver's `mmap` interface. This interface enables Linux char devices to map portions of their memory into the address space of their client. By

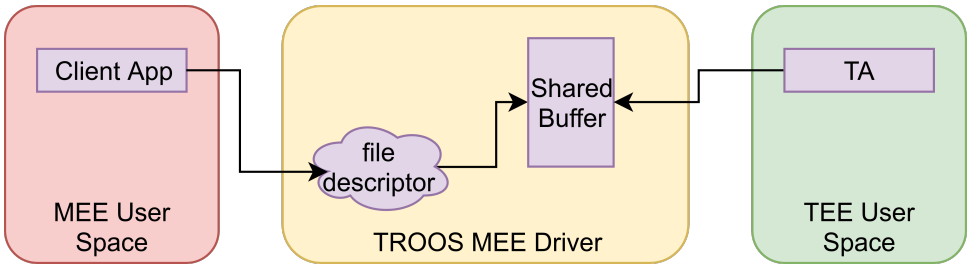


Figure 4.5: TROOS MEE Driver Shared I/O Buffer. The shared I/O buffer is allocated by the TROOS driver in kernel memory. The TA can directly access the buffer, since it is mapped to its memory during TCREATE. The client application uses the driver's read and write operation to access the buffer.

using mmap, the TROOS MEE driver can (theoretically) map the shared I/O buffer directly into the address space of the client application. However, this feature was not yet implemented in our driver.

4.5 Epilogue

In this chapter we presented the design of TROOS, an OS for TrustZone based TEEs. At the core of our design stands the principle that the design of a trusted system must take into account well known security concepts, and use them as the guidelines for the design. As banal as it may seem, we showed that is not the case in existing many TEEs.

We then demonstrated how the security concepts implemented in TROOS (such as least privileges, attack surface minimization and TCB reduction) improve its overall security relatively to QSEE (Qualcomm's widespread TrustZone based TEE). We did so by analyzing a real attack on a TEE and compared the results to a TEE that uses TROOS.

We believe that the theoretical tools presented in Chapters 1 and 2 combined with the approach we used to design TROOS should be implemented whenever designing a secure system.

Naturally, there is still much work to be done in order to bring TROOS to production level quality. However, we are certain that this effort, if invested, will not be in vain, and would lead to more secure systems.

Chapter 5

TKRFP – Trusted Kernel Return Flow Protection

In this chapter we present a TEE-based protection against return flow hijacking attacks, to which we call TKRFP: Trusted Kernel Return Flow Protection. This protection mechanism addresses the unfortunately forgotten problem of protecting the kernel against return flow hijacking attacks. Our solution uses the TEE to protect the return flow information in a secure way, such that even the MEE OS itself cannot manipulate it. Such a mechanism can also be extended to other kinds of security mitigations in order to reduce the attack surface of the MEE further, e.g., by using the TEE to implement an indirect branch protection mechanism.

Control flow hijacking is a family of attacks that takes over the program execution by modifying its control flow. These attacks usually utilize a write primitive (i.e., the ability to write to a memory location) to overwrite some data that the program uses to control its flow. For example, by overwriting a return address stored on the stack, an attacker can direct the program execution to an address of his choice, which may point to some code that the attacker wishes to run. This particular type of control flow hijacking is called return flow hijacking. Although initially the scope of return flow hijacking attacks seemed limited, by utilizing advanced techniques such as return oriented programming (ROP) an attacker may be able to run almost anything he desires. As described in the introduction, return flow hijacking attacks are frequently preformed by attackers. Thus, return flow hijacking attacks pose a serious security threat, and an effective protection mechanism for these attacks can significantly improve the system security.

There are several possible mechanisms to prevent control flow hijacking. Kernel address space layout randomization (KASLR) can somewhat reduce the impact of control flow modification. Unfortunately, most KASLR implementations offer only limited protection, while even a minor information leak suffices to bypass the KASLR protection of the most popular systems [44, 42, 43]. Stack canaries are also a very common and effective solution, but they can only protect against a limited set of

attacks. Intel recently announced a new planed technology to prevent control flow hijacking dubbed *Control-flow Enforcement Technology* (CET) [28], but no time line for its availability was published. Microsoft also implemented a software-based control flow hijacking mitigation which they call *Control-flow Guard* (CFG). CFG protects against various types of control flow hijacking, but return flow hijacking is not one of them. Microsoft also developed *Return-flow Guard* (RFG), which is an experimental security mechanism designed to provide protection against return flow hijacking. RFG was implemented in Windows 10 beta releases, but was then discontinued due to major flaws found in its design [29]. Both CFG and RFG were designed for the protection of unprivileged (user space) code. See Section 1.4 for a detailed discussion on these protection mechanisms.

In this chapter, we present *Trusted Kernel Return Flow Protection* (TKRFP), which is a TEE-based return flow hijacking protection specifically designed to protect the kernel. Our solution is based on a shadow stack: Whenever a return address is stored in the call stack, a copy of that return address is also stored on the shadow stack. Before the actual return is preformed the top address stored in the shadow stack is compared to the target of the return instruction (which is stored on the regular stack). If there is a match, the shadow stack is popped, and the return instruction is executed. Otherwise, the mismatch causes the kernel to stop running, thus enforcing protection from a modified return address on the regular stack. By utilizing TEEs (such as TrustZone) to protect the shadow stack, TKRFP prevents an attacker who gained kernel read and write privileges from successfully controlling the code execution by the means of return flow hijacking.

The rest of this chapter is organized as follows: In Section 5.1 we define the TKRFP threat model. The architecture of TKRFP is presented in Section 5.2, and Section 5.3 provides the details of our proof of concept implementation. We continue by evaluating our proof of concept implementation in Section 5.4. Finally, we discuss various related issues in Section 5.5.

5.1 TKRFP Threat Model

We assume a local unprivileged attacker, i.e., that the attacker has an initial foothold in the userland, and that this foothold cannot bypass the security policy enforced by the kernel. Under this threat model the attacker is able to run arbitrary userland code, and he can also interact with the kernel through various kernel APIs.

Moreover, we assume that the attacker has successfully exploited some kernel vulnerabilities, through which he gained arbitrary read and write primitives to the kernel memory. The attacker is then trying to leverage these primitives to gain arbitrary code execution within the kernel.

On the other hand, the kernel is assumed to be relatively well protected, e.g., it is assumed to use the most common vulnerability mitigations such as KASLR, NX and

stack canaries.

The attacker's read and write primitives enable him to bypass security mitigations deployed in the kernel, and hijack the kernel return flow. A write primitive can be used to hijack to program execution flow by overwriting a return address. Moreover, since we assume that the attacker can write to arbitrary memory locations, he can bypass the stack canaries protection by overwriting only the return addresses while leaving the stack canaries intact. Another way to bypass stack canaries is to use a read primitive to leak their content, thus nullifying their protection. A read primitive may also be used to bypass KASLR by leaking address information.

However, we assume that the attacker cannot change the kernel code segment. This is a reasonable assumption for two main reasons:

1. Previous works (such as [10, 19]) already showed how the kernel can protect its code from being changed by attackers with kernel read and write primitives.

2. The kernel code is mapped to non-writable memory pages. Although an attacker with read and write primitives can re-map the code pages such that they are writable, this kind of attack is much more complicated.

5.2 The TKRFP Architecture

We designed TKRFP to protect the kernel against a variety of attacks that manipulate return addresses stored on the stack. Since the stack is necessarily writable, the attacker typically has write access to it. Such attacks include ROP attacks which write to the stack directly (e.g., by overflowing a buffer with specific content chosen by the attacker), as well as more complex attacks that manipulate the stack indirectly. The latter kind may use the OS to perform writing: We present such an attack, in which the attacker manipulates a process to write to another process's stack, in Chapter 6.

Various solutions were proposed for protecting user space programs. Some of which protect the stack data by making it (or a copy of it) accessible only to the OS kernel, ensuring that user programs cannot manipulate or modify it. However, we are not aware of any existing solution targeting kernel protection, and which can protect the kernel from an attacker with kernel write privileges. Our solution is designed especially to protect the OS kernel. We use TEEs in order to protect the stack data from the OS kernel and also from user programs that succeed to gain access to the OS kernel memory. In this work a TrustZone-based TEE is used.

TKRFP adopts the idea of a shadow stack. In order to overcome weaknesses of other shadow stack implementations, our solution protects the shadow stack by disabling any direct read or write access by any entity of the MEE (both user space programs and the MEE operating system kernel). Our shadow stack is created and managed by a dedicated code in the TEE. The shadow stack is therefore accessible only to the TEE in a way that prevents the MEE OS and any of its entities from accessing it.

```
foo:
push {fp, lr}
add fp, sp, #8
sub sp, sp, #12
SHADOW_STACK_PUSH
```

Listing 5.1: ARM Procedure Prologue with Shadow Stack Protection

```
SHADOW_STACK_POP
sub sp, fp, #8
pop {fp, pc}
```

Listing 5.2: ARM Procedure Epilogue with Shadow Stack Protection

The MEE must still be able to instruct the TEE code to push or pop the return address to/from the shadow stack, so operations asking the TEE to push or pop must be provided to the MEE OS. In TKRFP, these push and pop operations are added to the MEE OS code during compilation: The shadow stack push operation is added to the procedure prologue, as demonstrated in Listing 5.1, and the shadow stack pop operation is added to the procedure epilogue, as demonstrated in Listing 5.2. These listings illustrate ARM assembly, as we implemented our prototype on an ARM CPU. The assembly instructions in these listings are the standard prologue and epilogue codes, and the capitalized markers show the locations of the added instructions.

An ideal implementation that requires an ISA modification would contact the TEE directly by the procedure call and procedure return instructions of the architecture, instead of the additional code in the prologue and epilogue.

On most systems the OS kernel does not have a single call stack. Instead, each process has its own kernel mode call stack. TKRFP can support both single and multi-stack systems, with as many stacks as needed. It maintains a shadow stack for each kernel mode stack. Whenever the kernel switches the stack (e.g., during a context switch), the shadow stack must also be switched. It is the OS kernel responsibility to ask the TEE to allocate new shadow stacks when needed and to ask the TEE to switch shadow stacks on a kernel stack switch. Our solution offers the OS kernel all the services required in order to do so, while still ensuring that the MEE kernel cannot directly access the shadow stacks.

5.2.1 TKRFP Security Properties

As stated above TKRFP's main goal is to catch all return flow hijacking attacks. In addition it also satisfies the following properties:

- The shadow stack is not accessible (neither for write nor for read) to any entity outside the TEE.
- Support for multi stack kernels.
- As transparent to the kernel developer as possible, i.e., a typical kernel developer does not need to interact with TKRFP (since most of the necessary code is added by the complier).

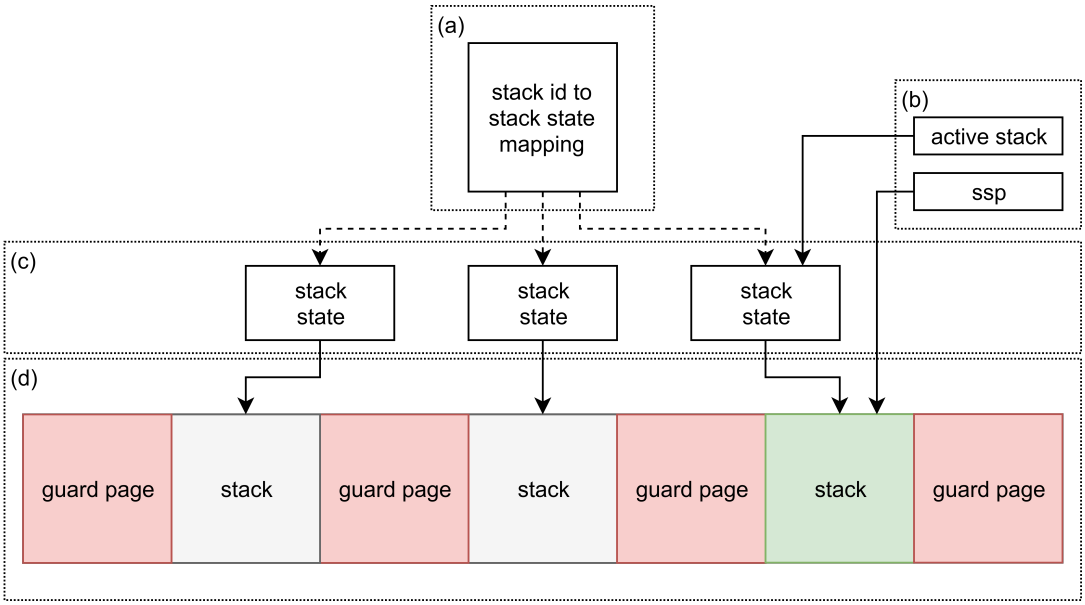


Figure 5.1: The TKRFP Architecture

- Backward compatibility. The ability to add shadow stack protection to existing kernels with minimal code changes.
- Forward compatibility. The interface between the the MEE and TKRFP is simple and fixed, so future OS updates will not require to wait for corresponding TEE updates.
- Reasonable performance impact.

5.2.2 The Shadow Stack Software Modules

The shadow stacks are manged by a dedicated TEE software module. The module uses several data structures:

(a) A mapping between shadow stack outer representation (stack id) to the shadow stack inner representation (stack state).

(b) Active shadow stack state pointer and SSP.

(c) Stack state data structures storing metadata for each shadow stack. In particular, the stack state stores the state of inactive stacks.

(d) The shadow stacks themselves. Each shadow stack is protected against downwards/upwards overflow by two adjacent guard pages.

An overview of the module components and their interactions is illustrated in Figure 5.1. All the module's components (code and data) resides in the TEE memory space.

Command	Parameters	Retun Value	Description
TKRFP_ALLOC	void	stack id	Allocate a new SS
TKRFP_SET_ACTIVE	stack id	void	Set as active SS
TKRFP_PUSH	return address	void	Push to the active SS
TKRFP_POP	return address	void	Pop from the active SS and verify
TKRFP_FREE	stack id	void	Free a shadow stack

Table 5.1: The TKRFP API

5.2.3 The Shadow Stack Operations

The shadow stack code and data reside in the TEE, and cannot be directly accessed by the MEE OS. However, the MEE OS can use the TEE interface to access the shadow stack API. The API is small and simple, as summarized in Table 5.1. Two operations, TKRFP_PUSH and TKRFP_POP are frequently called, while the other three are used to manage the stacks. The API operations are discussed below.

5.2.3.1 Initialization

During the system's boot process the TEE is typically booted before the MEE. At this stage TKRFP is initialized, and it is ready when the MEE starts to boot.

TKRFP initializes its data structures such that the first shadow stack activation operates in a similar manner to the subsequent calls. The active shadow stack pointer points to a dummy stack state, which is used to store the "previous" state, and each stack state stores the address of its stack head.

5.2.3.2 Creating a Shadow Stack (TKRFP_ALLOC)

It is the kernel responsibility to ask the TEE to allocate shadow stacks when needed (i.e., when allocating a new kernel stack). Upon receiving a request for shadow stack allocation (called TKRFP_ALLOC), the TEE allocates a new shadow stack (along with its guard pages), and assigns it an unused stack id. A stack state structure is also allocated to hold the new stack metadata, and a mapping between the stack id and stack state is created. Finally the stack id the returned to the MEE as the return value. The MEE kernel is expected to store this value in its process table.

5.2.3.3 Activating a Shadow Stack (TKRFP_SET_ACTIVE)

Among all the allocated shadow stack, only one is active at any time. The shadow stack push and shadow stack pop operations are always preformed on the active shadow stack. The MEE kernel is responsible to decide which of the allocated shadow stacks is the active one and ask the TEE to set it as active using the TKRFP_SET_ACTIVE API command. This command stores the current SSP in the stack state pointed by the active stack pointer, and then updates the TEE data structures with the new active

shadow stack. This update includes updating the SSP and the active shadow stack pointer.

For example, whenever the kernel switches stacks, it should also switch the active shadow stack by calling to the TKRFP_SET_ACTIVE API command with the shadow stack id of the stack to be activated.

5.2.3.4 Pushing to the Shadow Stack (TKRFP_PUSH)

The compiler adds to the procedure prologue a call to the TEE push command. The push command operates on the active shadow stack, and stores the procedure return address, which is received as a parameter, to the memory location pointed by the SSP. An alternative implementation can let the the TEE to read the return address directly from the MEE (e.g., read the banked LR). After storing the return address on the shadow stack, the SSP is incremented. If there is no active shadow stack while calling a push command an attack is assumed.

5.2.3.5 Verifying and Popping from the Shadow Stack (TKRFP_POP)

The compiler adds to the procedure epilogue (or any return sequence) a call to the TEE pop command. The pop command is called TKRFP_POP. It operates on the active shadow stack. After popping the top of the active shadow stack (pointed by the SSP), the result is compared to the address the MEE is about to use as the return address, which is given as a parameter to the pop command. If the addresses do not match, an attack is assumed, and the TEE preforms a security lockdown of the MEE and stops its execution.

There are various alternatives for lockdown. For example, once an attack is identified, the TEE may pass control to a dedicated code in the MEE that is assumed to be safe and unmodified by the malicious code. In this case, even though we know that the MEE was attacked, we still trust some specific MEE code to handle its protection. But, of course, in this case the TEE should take precaution measures (e.g., measure the relevant MEE code and verify its authenticity) before passing control back to the MEE.

5.2.3.6 Freeing a Shadow Stack (TKRFP_FREE)

When the MEE frees a kernel stack it should also ask the TEE to free the corresponding shadow stack, using the shadow stack free command, TKRFP_FREE. The TEE frees the stack control structures, and memory and deassigns the shadow stack id.

The active shadow stack cannot be freed (an error is returned if that happens). This fact allows the TKRFP_PUSH and TKRFP_POP to assume that there is an active shadow stack, and not verify the shadow stack before their operation.

```
struct ss_state {
  void *head;
  ss_status_t status;
};
```

Listing 5.3: The Stack State Record

5.3 Prototype Implementation

We implemented a prototype of TKRFP with TrustZone-based TEE. The prototype is composed of three parts:

- A TEE module with TKRFP API, running inside TrustZone.
- A Linux kernel with TKRFP support, with a dedicated code to allocate, free and switch TKRFP stacks using the TKRFP API.
- A GCC plugin to generate the TKRFP procedure prologues and epilogues during the kernel compilation.

5.3.1 The TKRFP TEE Module

The TKRFP TEE module creates and manages the shadow stacks. The module is loaded to TrustZone at system boot time, and exposes the TKRFP API to the MEE via the SMC mechanism.

In order to achieve better performance we implemented TKRFP in a way that never preforms a full world switch. The TKRFP commands are simple enough that the monitor mode is capable of handing them, without the extra overhead of switching to NS=0: When a TKRFP command is invoked, the monitor mode code handles the shadow stack operation from start to finish, and then returns control to the MEE, without the usual (costly) transfer of control to "non-monitor" secure code.

The TKRFP TEE module uses two arrays as its main data structures. The first is an array of shadow stacks, 4KB each. A size of 4KB is more than enough, since the shadow stack size is bounded by the kernel stack size which is very limited. Additionally, recall that our shadow stacks store only return addresses, therefore a stack of 4KB can store over 500 of return address, which is much more than the kernel typically needs. The second array is of the same length, but each entry in it is a stack state record (the record definition is given in Listing 5.3), which holds the metadata of a corresponding shadow stack. A shadow stack id is the array index of the corresponding stack state record. In addition to the two arrays, two auxiliary global variables are also used:

1. The SSP, which points to the head of the active shadow stack.
2. The active stack pointer, which points to stack state record corresponding to the active shadow stack.

The TKRFP_SET_ACTIVE, TKRFP_ALLOC, and TKRFP_FREE operations update the metadata and the global variables. The last two operations only change the stack status to mark it as allocated or free. But the TKRFP_SET_ACTIVE operation stores the SSP on the active stack state before restoring its value from the requested stack state.

The TEE module code needs to increment the SSP on every TKRFP_PUSH, and decrement it on every TKRFP_POP. These maintenance operations have considerable impact on the TKRFP overhead. We discuss a way to reduce this maintenance time in Subsection 5.5.2.

5.3.2 The TKRFP Customized Linux Kernel

In order to support TKRFP, the MEE OS kernel must be aware of the services provided by the TEE, and TKRFP push and TKRFP pop commands must be added to the procedure prologues and epilogues. Since out threat model assumes that the kernel code cannot be changed by the attacker, the TKRFP commands cannot be removed, and the protection of TKRFP is promised. The kernel code is only slightly modified compared to the original kernel in order to allocate the shadow stacks and switch them at context switches. Most of the modifications are those made by the compiler at compile time.

We customized a Linux Kernel 2.6.35 to support TKRFP by making the following changes to the kernel code:

- A new field, tkrfp_id, was added to the process descriptor record (struct task_struct). This field is used to store the id of the shadow stack that is allocated for the process.
- When a new process is created, the kernel allocates a kernel stack for that process. In order to support TKRFP, the kernel is required to also allocate a new shadow stack (using the TKRFP API) for the new process. The shadow stack id is stored in the tkrfp_id field in the process descriptor.
- The active TKRFP shadow stack should always be the one protecting the process which is currently running. When the kernel preforms a context switch, it must also switch the active TKRFP shadow stack.
- When the kernel frees a kernel stack, it should also free the corresponding TKRFP shadow stack.

All these changes are controlled by a new configuration flag, *CONFIG_TKRFP*, that we added to the kconfig security menu.

5.3.3 The TKRFP GCC Plugin and Automatic Kernel Modifications

GCC offers the possibility to extend its features by loading plugins [58], which are loadable modules that interact with the complier via a rich subset of GCC internal

```
foo:
push {r0,r1}
mov r0, #TKRFP_PUSH
mov r1, lr
smc #0
pop {r0,r1}
push {fp, lr}
...
```

Listing 5.4: Procedure Prologue with TKRFP Protection as Generated by The Plugin

```
...
push {r0,r1}
mov r0, #TKRFP_POP
mov r1, [fp]
smc #0
pop {r0,r1}
sub sp, fp, #8
pop {fp, pc} //the return instruction
```

Listing 5.5: Procedure Epilogue with TKRFP Protection as Generated by The Plugin

API. GCC compilation is composed of a series of passes. Each pass takes the current program representation a step forward towards an object file, which is the result of the compilation process. A plugin enables us to add our own pass before or after any of the existing passes in the series.

In order to deploy the TKRFP push and pop commands, we wrote a dedicated GCC plugin, and added it in the late stages of the compilation process, specifically after the *pro_and_epilogue* pass. This means that our plugin gets an intermediate representation of the program (called RTL), which already contains the generated procedure prologues and epilogues. Our plugin adds the required call to the TKRFP API command for every procedure prologue and epilogue. See Listing 5.4 and Listing 5.5 for details of the two cases.

As previously stated, on the ARM architecture, leaf procedures do not store their return address on the call stack, and are not vulnerable to return flow hijacking attacks. Therefore, our plugin identifies leaf procedures and skips them. Deployment of TKRFP protection is performed only to non-leaf procedures.

Since we intervene only in the late stages of the compilation process, most of the optimizations done by the compiler do not affect our plugin. However, two of these optimizations, the sibcalls optimization and the conditional return optimization, are directly linked to the procedure epilogue, and our plugin must handle them correctly.

5.3.3.1 Handling GCC's Sibcalls Optimization

Sibling procedures are procedures that share both return type size and arguments size. A *Tail call* is when a procedure ends with a call to a procedure and returns its return value to the original caller. When a tail call is done to a sibling procedure it is called a *Sibcall*.

One of GCC's optimization passes handles sibcalls. When GCC identifies that a certain procedure ends with a sibcall, the procedure does not return, but instead "calls" the sibling procedure in a special way. This "call" preforms some necessary stack adjustments, pops the return address to LR, and then branches to the sibling procedure. Listing 5.6 and Listing 5.7 demonstrate such a case.

```
void bar(void)
{
  foo();
  foo();
}
```

Listing 5.6: Sibling Procedures Example

```
<bar>:
push    {r4, lr}
bl      foo
pop     {r4, lr}
b       foo
```

Listing 5.7: The Generated Assembly with Sibcalls Optimization

```
<bar>:
push {r0,r1}
mov r0, #TKRFP_PUSH
mov r1, lr
smc #0
pop {r0,r1}
push {r4, lr}
bl foo
pop {r4, lr}
push {r0,r1}
mov r0, #TKRFP_POP
mov r1, [fp]
smc #0
pop {r0,r1}
b       foo
```

Listing 5.8: TKRFP GCC Plugin Sibcall Handling

Sibcalls present a problem for our plugin. Let us consider the case demonstrated in Listings 5.6 and 5.7. On one hand, the procedure `bar` is not a leaf procedure, hence a TKRFP_PUSH command is added to its prologue. On the other hand, `bar` ends with a call to `foo`, and not with an epilogue that can properly clean the shadow stack. Therefore the call to `bar` leads to a shadow stack mismatch.

To handle this issue, our plugin must identify sibcalls, and add a corresponding TKRFP_POP command to the stack adjustments code, in way that bypasses the described problem. See Listing 5.8 for an example.

5.3.3.2 Handling GCC's Conditional Return Optimization

ARM architecture supports adding a condition code to almost any instruction. When a condition code is added, the instruction is executed only if that condition holds. For example, if we add the *eq* condition code to the mov instruction, then it is executed only if the zero flag is set.

GCC uses conditional return instructions for both code size shrinking and performance improvement. By using conditional return GCC reduces the number of branches it is inserting to the object file, thus reducing the code size. The reduction of branches also reduces the number of instructions executed in run time. This optimization is particularly useful for procedures that verify their arguments and return on failure, as

```
void foo(void *p)
{
if ( p == NULL )
  return;
...
}
```

Listing 5.9: Conditional Return Example

```
<foo>:
mov ip, sp
push {fp, ip, lr, pc}
...
cmp r0, #0
ldmeq sp, {fp, sp, pc} //cond return inst
...
```

Listing 5.10: The Generated Assembly with Conditional Return Optimization

```
<for>:
mov ip, sp
push {fp, ip, lr, pc}
...
cmp r0, #0
stmfdeq sp!, {r0,r1} //cond push inst
moveq r0, #TKRFP_POP // cond mov inst (TKRFP arg1)
moveq r1, [fp] // cond mov inst (TKRFP arg2)
smceq #0 //cond SMC inst
ldmfdeq sp!, {r0,r1} //cond pop inst
ldmeq sp, {fp, sp, pc} //cond return inst
...
```

Listing 5.11: TKRFP GCC Plugin Conditional Return Handling

described in Listings 5.9 and 5.10.

Notice that a special stack frame handling is used to enable the conditional return: The initial stack pointer value is pushed to the stack (by pushing the ip register) in addition to the regular pushed registers in case that there is any conditional return in the procedure. In such a case, at time of procedure return (either conditional or non-conditional) the stack pointer is added to the list of popped registers, and thus the rest of the pops are preformed correctly.

To handle conditional returns we created a conditional TKRFP pop call. When our plugin identifies a conditional return, it simply inserts the conditional TKRFP pop command with the same condition. We made sure that this addition does not change the control flags, so both the return and TRKFP pop are either preformed together or skipped together. See Listing 5.11 for an example.

5.4 Performance Evaluation

We tested our modified kernel with TKRFP protection under the Re-aim framework [68] using various workloads. The Re-aim framework provides a set of micro-benchmarks which can be combined to create a simulation of a real workload, such as database, files server or heavy computing. Our tests show that the performance impact of TKRFP is small. In this section we provide an evaluation of the duration (in cycles) of various

	Linux	Linux With Canaries	Linux With TKRFP
Size	3116796B	3173792B	3168500B
Security Overhead		1.8%	1.6%

Table 5.2: Return Flow Protection Impact on the Kernel Image Size

CPU instructions including the SMC instruction (which switches the CPU to monitor mode and back), and then present the overall effect of TKRFP on the Re-aim workloads. We also compare the overhead of TKRFP to the overhead of stack canaries on the same workloads.

5.4.1 The Experimental Setup

We tested the TKRFP prototype on a Freescale's `iMX53qsb` board. The `iMX53qsb` is a low cost development board with an ARM Cortex-a8 (1Ghz) CPU, 1GB of RAM and full TrustZone support.

The MEE software used in our prototype is BusyBox 1.2.1 on top of Linux 2.6.35 complied with TKRFP support as described in Section 5.3.2. The TEE software used in our prototype is the TKRFP TEE module as described in Section 5.3.1, without an OS on the TEE side.

5.4.2 Kernel Image Size

The TKRFP complier adds instructions to the kernel image. The first step in evaluating the feasibility of TKRFP was to measure the impact of these extra instruction on the kernel image size. As shown in Table 5.2, we observed an increase of 1.6% in the kernel size, which is similar to the impact of compiling the kernel with canaries.

5.4.3 Timing the SMC Instruction

Besides the SMC instruction, all the TKRFP instructions are "regular" instructions. As a first step in evaluating the TKRFP performance overhead we measured the execution time of the SMC instruction.

Upon SMC invocation, the CPU mode is switched to monitor mode and an handler is called according to the registered monitor table (as illustrated in Figure 2.3). In our measurements we use a void handler, that is an handler that simply returns once called. Since in TKRFP an SMC is used to trigger the API commands, the execution overhead of an SMC needs to be measured from the call to the SMC till its return. We also measured simple instructions such as MOV, STR and LDR. The results are listed in Table 5.3.[1]

In the "not cached" scenario, TKRFP has quite a considerable impact on the procedure execution time where the code runs for the first time. This behavior is not

[1]NONE actually measures the timing measurement overhead. This value served us to tune the timing of all the other instructions.

	NONE	MOV	STR	LDR	SMC
Cached	1	1	1	1	58
Not cached	15	1	15	16	450

Table 5.3: Basic Instructions Cycles Consumption

	No protection	TKRFP	Canaries
Cached	30	199	39
Not cached	495	2666	591

Table 5.4: Return Flow Protection Performance Impact on a Single Procedure

excpected to repeat in the next time TKRFP protection is called, as we expect the TKRFP code and data to be cached after its first call (or few first calls).

5.4.4 Timing Protection Overhead of a Called Procedure

In order to measure the performance impact of the shadow stack, we timed the execution of an almost empty non-leaf C procedure (recall that in the ARM architecture leaf procedure are not vulnerable to return flow hijacking). Once complied, this C procedure consists mostly of its prologue and epilogue, thus it allows us to measure the impact of changing the prologue and epilogue. We measured three cases: (1) a procedure without return flow protection (2) return flow protection by shadow stack and (3) return flow protection by stack canaries. For each case we measured two scenarios. First, a single call to the procedure (which is highly affected by cache overheads). Second, an average of 10,000 successive calls (which forces all memory accesses to reside in the cache, thus eliminating the cache overhead). The results are listed in Table 5.4.

Since our procedure consists mostly of its prologue and epilogue, most of it operations are stack accesses. Once the relevant stack address and the procedure code are cached its execution time is considerably reduced. The canaries protections only adds a couple of memory accesses, therefore its impact in cached scenario is small. On the other hand, TKRFP is more complex, as it uses the costly SMC instruction two times for each procedure. Therefore, its impact on the procedure execution time is larger. However, the cached scenario shows us that TKRFP also enjoys the cache benefits. We continue to discuss the TKRFP performance impact in the following subsubsection.

5.4.4.1 Deconstructing the Performance Overhead

Let us take a look on the execution time of a TKRFP protected procedure. As seen in Table 5.4, the (cached) execution time is 199 cycles. It also shows that the procedure code itself takes 30 cycles, so the TKRFP protections costs $199 - 30 = 169$ cycles per procedure call and return. To support TKRFP two SMC instructions are added to the procedure (one for TKRFP push and one for TKRFP pop), so of these 169 cycles 116 are due to the two SMC invocations, which takes 58 cycles each according to Table 5.3.

The remaining 53 cycles preform the actual push and pop code in the TKRFP TEE module.

5.4.5 The Re-aim Framework Workloads

Now, that we know the overhead of TKRFP on a single procedure call, we wish to measure the impact of TKRFP on the whole system. For this purpose we used the Re-aim framework, and tested our prototype under various workloads. The Re-aim framework tests consist of "jobs", which are functions focused on a specific aspect of the system (math operation, disk write, etc.). Each test runs the same list of jobs, under several sub-processes.

The Re-aim framework measures its tests execution time, and reports the following metrics:

1. **Forks** Number of subprocesses in the test.
2. **JPM** Number of jobs finished per minute.
3. **JPM_C** Number of jobs finished per minute divided by the number subprocesses.
4. **P_tm** The total time of the test.
5. **C_utm** The total user space time scheduled for subprocesses.
6. **C_stm** The total kernel space time scheduled for subprocesses.

In our protection mechanisms analysis we add two more metrics, which measure the impact of the protection mechanism compared to the "original" kernel:

1. **S_oh** The system time overhead, i.e., $C\_stm_{new}/C\_stm_{original} - 1$.
2. **T_oh** The total time overhead, i.e., $P\_tm_{new}/P\_tm_{original} - 1$.

We applied two Re-aim workloads: The *compute* workload, which consists of multiple CPU-bound jobs, and the *disk* workload which consists of multiple I/O-bound jobs.

In the compute workload, the test processes does not call any OS service. Notice, that context switches are still preformed, so TKRFP commands are called by the MEE. Moreover, the Linux scheduling policy assigns a time slice to each process, causing a CPU-bound process to quickly consume its time slice, and to be switched out by the OS. Therefore, the compute workload creates a fair number of context switches.

Under this workload we observed an overhead of around 5% to the overall test time, while canaries caused an overhead of around 3%. The results are listed in Table 5.5.

The disk workload consists of multiple jobs that preform frequent disk I/O operations, such as reading from files or writing to files. Each job requires very small CPU time, and spends most of its time waiting for I/O operations to be completed. In such a workload the test processes use the OS services very frequently.

	JPM	JPM_C	P_tm	C_utm	C_stm	S_oh	T_oh
Linux	518	103	58	52	1.7	–	–
With TKRFP	495	99	61	52	1.9	11%	5%
With Canaries	505	101	60	52	1.7	0%	3%

Table 5.5: Return Flow Protection Performance Impact on a CPU-Heavy Workload

	JPM	JPM_C	P_tm	C_utm	C_stm	S_oh	T_oh
Linux	88	17	337	0.1	4.27	–	–
With TKRFP	89	17	335	0.29	5.13	20%	-1%
With Canaries	82	16	364	0.17	5.15	20%	8%

Table 5.6: Return Flow Protection Performance Impact on a Disk-Heavy Workload

Under this workload both TKRFP and canaries caused an overhead of around 20% to the system time. But since the total test elapsed time is much longer than the system time, the overall overhead on the whole system is of -1% and 8% respectively. The results are listed in Table 5.6.

On both workloads the impact of TKRFP on the whole system is relatively low. The impact on the system time is larger, but still remains reasonable. In the disk workload the TKRFP overhead is similar to the overhead of canaries. Although, at first glance it seems that TKRFP presets poor results in comparison to canaries under the compute workload, a deeper look shows that this is because canaries do not protect any procedure in the context switch execution path, while TKRFP does so.

Considering the extensive protection TKRFP provides, and its relatively low overhead we conclude that TKRFP is a very attractive return flow protection mechanism for the kernel.

5.5 Discussion

We conclude this chapter with a few possible improvements to TKRFP. Some of these improvements can reduce the TKRFP overall overhead, while others mitigate security flaws in the prototype implementation.

5.5.1 TKRFP With TrustZone-M

As shown in Subsubection 5.4.4.1, a significant part of the TKRFP overhead is due to the switches from the MEE to the TEE and vice versa (i.e., the SMC instruction). It is a price we must pay in order to use TKRFP. We cannot reduce the overhead caused by these transitions.

However, a TEE that provides faster transitions could further reduce the overhead of TKRFP. The ARM Cortex-M series with TrustZone-M provides such an opportunity. In fact, the minimization of this overhead is one of the main design goals in the TrustZone-M architecture [71]. Keeping that in mind, we programmed our prototype in a way that

supports quick porting to the ARMv8-M architecture.

5.5.2 Further Reducing the TKRFP Overhead

Another considerable part of the TKRFP overhead is due to the management of the SSP. The ARM architecture offers a separate state registers bank for each processor mode. In particular, the monitor mode has its own SP register. By using the monitor SP register for our SSP instead of its intended use, we can reduce the TKRFP overhead, since now the TKRFP TEE module does not need to preform SSP increments or decrements separately from the memory access. Instead, the pointer is managed by the architecture using ARM's push or pop instructions. A disadvantage is that we have no call stack to use for the monitor mode itself, which makes programming much more difficult. This improvement is left to later versions of TKRFP.

5.5.3 TOCTOU

We are aware of a possible TOCTOU vulnerability in our compiler TKRFP pop command implementation. Since we take the return address from the stack, there is a (small) gap from the time we finish our check to the time the return address is used by the return instruction. If an attacker overwrites the return address inside the gap, he can bypass our protection.

We can fix this by first popping the return address to LR, then calling TKRFP POP and then returning using LR. It is not a simple fix, and once fixed, performance is expected to reduce slightly.

5.5.4 TKRFP Push Cannot Be Used as a Write Primitive

The shadow stack present somewhat contradicting requirements: On one hand it require protected memory to store the shadow stack, and on the other hand the program must modify this memory on every procedure call and procedure return. This fact, can expose the system to unwanted threats.

For example, recall that in Intel's CET the shadow stack memory is a part of the virtual address space. In order to be protected from overwrites, the shadow stack memory pages are marked by a special configuration of the page tables that makes them non-writable. Only the call instructions can write to these non-writable memory pages. It is therefore that the shadow stack push operation is a write primitive to non-writable memory regions.

TKRFP does not create such an unwanted write primitive. First, TKRFP uses the TEE memory to protect the shadow stack, and does not write to any unintended non-writable pages. Second, in TKRFP we are ensured that the SSP always points to a shadow stack memory page. In TKRFP there are three ways to change to SSP value:

1. Switching shadow stacks: the TKRFP_SET_ACTIVE command changes the SSP value. The new value is selected from a stack state, thus points to a shadow stack.
2. Pushing to the shadow stack: the TKRFP_PUSH command increases the SSP. An upper guard page ensures that this command cannot overflow the shadow stack.
3. Popping from the shadow stack: the TKRFP_POP command decreases the SSP. A lower guard page ensures that this command cannot underflow the shadow stack.

By (1) we are ensured that on context switch the SSP points to a shadow stack memory, and by (2) and (3) we are ensured that neither the TKRFP_push nor the TKRFP_POP operations can cause the SSP to cross the shadow stack memory boundaries.

Chapter 6

Context-Switch Oriented Programming Attacks

In this chapter we present the context switch oriented programming (CSOP) attack, which is a new attack that can potentially bypass any shadow stack protection. The attack uses the fact that the kernel is responsible for the context switches, and in particular for the management of the process descriptors that contain the shadow stacks information.

For example, in TKRFP the kernel stores the stack id returned by TKRFP_ALLOC in the process descriptor table, and uses it as an argument to TKRFP_SET_ACTIVE, and in Intel CET the SSP is typically stored in the kernel stack when the process is switched out.

An attacker that has write capabilities to the kernel memory can overwrite this data, and control which shadow stack is chosen by the kernel at every context switch. The attacker can use this capability to partially control the shadow stacks content with the intention of overwriting the return addresses in a way that lets him choose his favorite return addresses. The result is similar to ROP [57] attack, where an attacker chooses code segments and calls them through a list of return addresses on the stack.

We should mention that it does not matter for security purposes if the selection of a shadow stack is done by explicit arguments passed by the kernel (e.g., a shadow stack id or SSP address) or by directly reading the kernel data structures (e.g., pid, kernel stack pointer, etc.). The attacker has the power to forge both kinds of data.

6.1 Kernel Control Paths

A *kernel control path* is the sequence of instructions executed by the kernel in response to a system call, an interrupt, or an exception. It starts when one of the three is invoked, and ends upon completing the service request. In the simplest situation, the CPU executes a kernel control path sequentially, i.e., beginning with the first instruction and ending with the last instruction. However, this is not always the case, and control paths

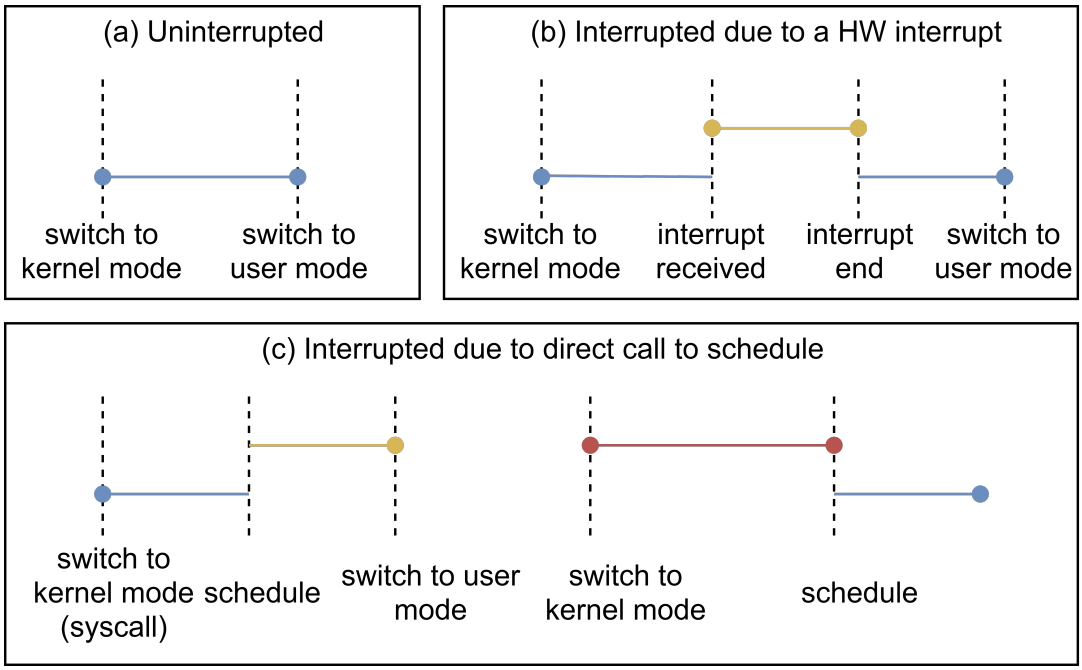


Figure 6.1: Kernel Control Paths

are often interrupted by other control paths. There are several situations where the CPU might interleave sequences originating from two different control paths:

1. An exception occurs while running a kernel control path (e.g., attempting to access data that is currently not in memory).

2. An hardware interrupt occurs while the CPU is running a kernel control path with the interrupts enabled.

3. When a kernel control path cannot immediately grant a request, that was issued by a user mode process by invoking a system call (e.g., when reading from disk). In this case, the OS puts the process in a wait queue, and calls the OS scheduler to switch context.

Typically in these situations the control paths are interrupted and the control is passed to another process via context switch. But notice that signals may allow an interrupted process to run user mode code which in turn may call other system calls.

Figure 6.1 exemplifies some control paths. In (a) an uninterrupted control path is illustrated. This control path is executed from start to finish without any interference. In (b) a control path is interrupted by an hardware interrupt. The control path execution is paused before it is finished, and a control path to handle an interrupt is invoked and executed to completion. Then, the original control path is switched back and completed. In (c) a control path is interrupted due to a direct call to the scheduler. The control path starts by a system call invocation, but calls the scheduler before its completion, and is switched out. Another control path is executed, after which the OS passes control

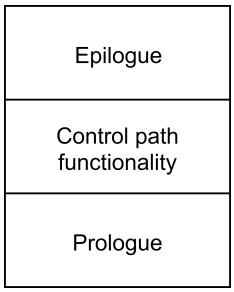


Figure 6.2: The Shadow Stack Structure

back to user space. Finally, the scheduler chooses the original process again and the control path is completed.

6.2 The Structure of Non-Active Shadow Stacks

Return flow hijacking attacks manipulate the stack in order to take over the program control flow. When a shadow stack is used to protect the regular stack, it must be manipulated as well in order to preform a successful attack. Therefore, it is important to understand the structure of the shadow stack. We do so by analyzing the structure of the kernel stacks of non-running processes.

The structure of the shadow stack follows the structure of the kernel stack, without the local variables and other data, as it contains the same return addresses as the kernel stack. We concentrate only on non-running processes, and thus all the processes we discuss have context switch frame at the top of the kernel stack.

Consider processes that were switched out during the execution of some kernel control path. The kernel stack contains a few frames, one for each procedure that was called (but did not yet return) during the current control path. As the shadow stack contains all (and only) the return addresses of the called procedures, it contains three sub-regions: A region that consists of the return addresses of the kernel entry, to which we call the *prologue*. A region that consists of return addresses of the functionality of the called control path, which might be empty in some cases, and a region consists of the return addresses of the scheduler and context switch code, to which we call the *epilogue*. Figure 6.2 outlines this structure. A more detailed illustration of the content of a shadow stack in the case of the Linux's `sleep`, `sched_yield` and `waitpid` system calls is provided in Figure 6.3.

6.3 Shadow-Stack Manipulation Techniques

Recall that under our threat model, the attacker can write to arbitrary locations in the kernel memory. Under this threat model an attacker has the power to overwrite the stack-id fields of processes, and completely control their content. But we assume that he can select only from a set of existing stack-ids (since the shadow stack implementation

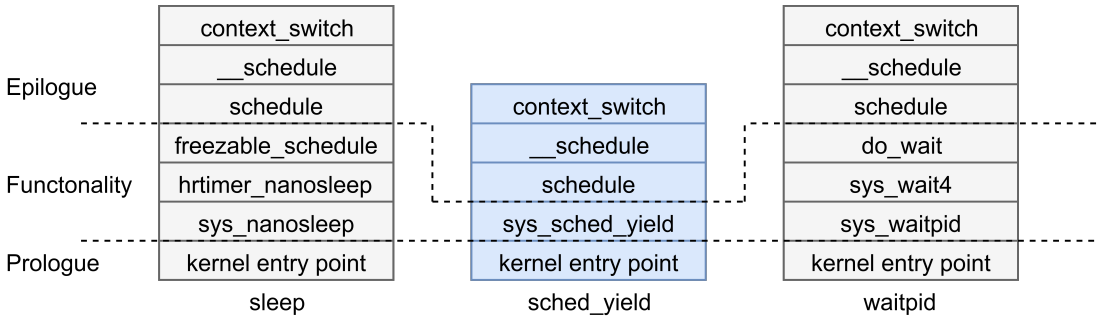


Figure 6.3: More Detailed Examples of the Content of the Shadow Stack

can easily detect and forbid use of unassigned stack-ids). Similarly, the attacker can select only from a set of valid SSP values. The next context switch to such an attacker-controlled stack-id, uses the modified stack-id as the argument to the shadow stack switch command.

The attacker may use this ability to bypass the shadow stack protection, and exploit the kernel. In such an attack, the attacker forces two or more processes to use the same shadow stack (either sequentially or simultaneously).

6.3.1 The Simple Case: Cascaded Frames

There are several ways to create cascaded frames. We describe here one that is most useful as a base for rest of this chapter.

If the attacker overwrites stack-ids of processes tha are paused during control path execution through a direct call to the scheduler, and forces them to use a single legitimate shadow stack id, then he can cascade frames of different processes on the same shadow stack. Although the attacker can choose specific frames that would be beneficial for him, the exact content of each frame cannot be fully controlled by the attacker. Each frame starts with a prologue (which then later causes to return to user space) and ends with an epilogue (which returns to the control path). In between these two there are other return addresses related to the control path functionality. This situation is illustrated in Figure 6.4.

Once the attacker finishes crafting the victim process shadow stack, he can overwrite the kernel stack of that process to fit the content of the shadow stack. Then, when the process is executed again, a sequence of control paths with context switches in between is executed. The algorithm for creating cascaded frames is presented in Algorithm 6.1. The resulting shadow stack structure is illustrated in Figure 6.5.

Notice that all the control paths executed in that manner were previously called by the involved processes. Therefore, this sequence of control paths could have been created by the OS scheduler anyway, but on different shadow stacks. Moreover, such cascades of control paths may be legally created by user space processes without any kernel write primitive. We elaborate on cascaded frames in this subsection mainly to

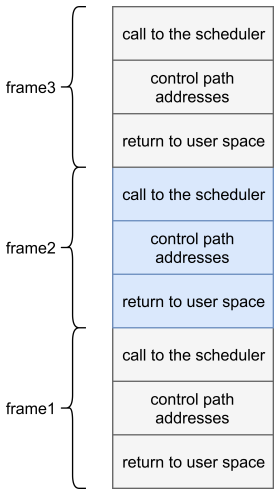


Figure 6.4: Cascaded Frames on a Shadow Stack

Algorithm 6.1 Creating Cascaded Frames

1: Create a new process P_0
2: **for** $i = 1$ to n **do**
3: Create a new process P_i in suspended mode
4: $SSP[P_i] \leftarrow SSP[P_{i-1}]$
5: Release P_i so it invokes control path i
6: Suspend P_i after it is switched out

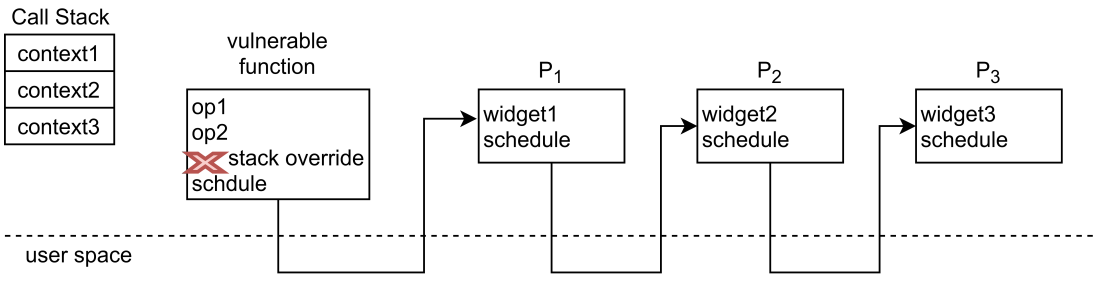


Figure 6.5: Cascaded Frames Execution

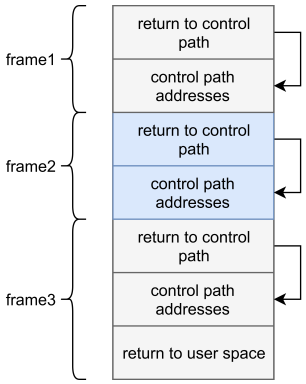


Figure 6.6: Trimming Cascaded Frames from the Bottom

serve as a basis for the next techniques.

6.3.2 Trimmed-Cascaded Frames

In the cascaded frames technique only complete frames can be pushed to the manipulated shadow stack. However, by creating an overlap between two different frames, the attacker can remove parts of the frames and achieve better control over the manipulated shadow stack content.

In such overlapping frames, one frame is written on top of part of the other frame, which may either be a consecutive part at the beginning or at the end of the other frame. Such a construction enables the attacker to construct shadow stack content that contain parts of frames, either on the side of the prologue or the side of the epilogue.

A possible way to create such an overlap between frames is to force different processes to use the same shadow stack (like in the cascaded frames technique), but to use SSP values that cause the frame to overlap. If the SSP is slightly incremented before the switch to another process, the frame is written on top on the previous one. The more interesting case is to start cascading the frames in the reverse order from the upper one in the Figure 6.6 towards the bottom, and use slightly decremented SSPs, which then force the bottom frames to be written on top of part of the ones above them.

6.3.3 SSP Injection

An attacker that can control the SSP value may direct it to any address on the shadow stack. By doing so, attacker can start the control path execution not only from its start, but from any point in the control path. SSP injection by itself may not be very useful. However, when combined with other shadow stack manipulation techniques, such as the trimmed-cascaded frames technique, it can add more power to the attacker.

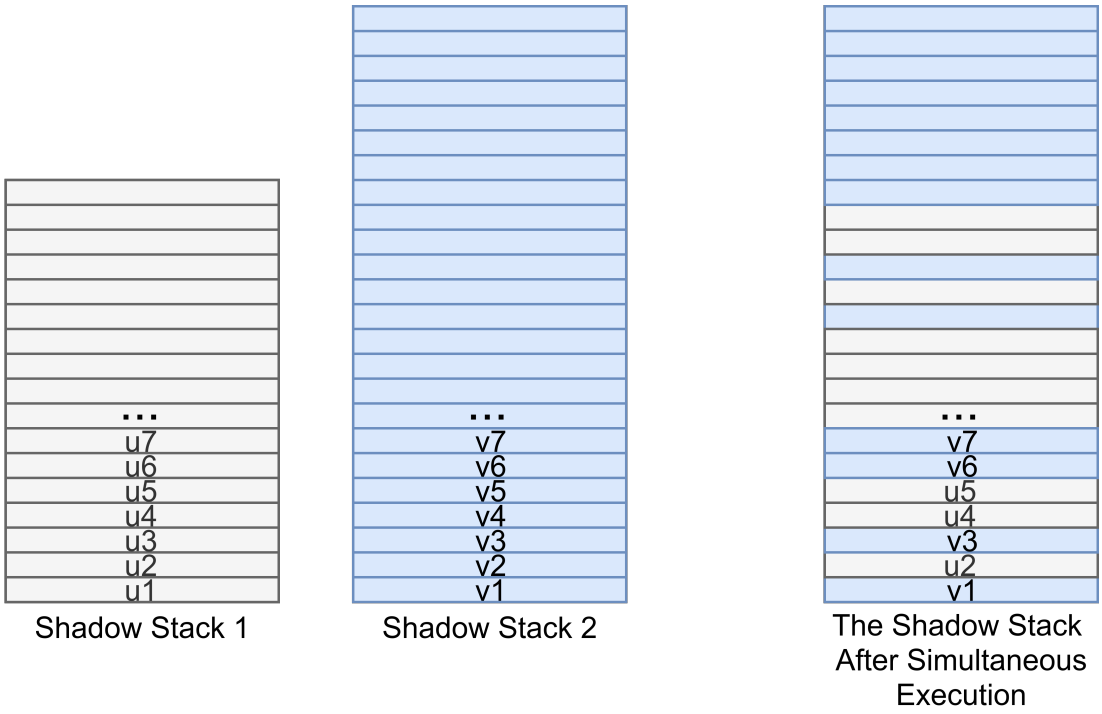


Figure 6.7: Shadow Stack After Simultaneous Execution

6.3.4 Simultaneous Use of a Single Shadow Stack

On multi-core systems several process are running at any time, each on a different core. Each running process may be executing a kernel control path. Therefore, multiple shadow stacks are used at the same time. Each core has its own SSP register, and typically each process has a different shadow stack.

An attacker may deliberately force two processes to use the same shadow stack at the same time. The attacker first overwrites the stack-ids of both processes with the same value. Then, he carefully times both processes such that they execute their control paths simultaneously. The result is that two processes are now running, using the same shadow stack, and writing on the same stack entires.

While the initial SSP values are equal on both processes, each process is running on different core and uses a different SSP register. Typically different control paths call different procedures in different timings, so the SSP registers soon start to drift from each other, while one SSP register advances faster than the other for a while, and later the other may catch up and pass.

The result is that instead of getting two shadow stacks, each with return addresses from a certain control path, we get a melted shadows stack consists of return addresses from both control paths, in some order (determined by the procedures call frequency). This situation is illustrated in Figure 6.7.

Notice that some return addresses are necessarily removed from the melted shadow stack. Since both processes write to the same stack and use the same addresses, each

address is used twice (in the case of two processes using the same stack). Only the return address that is written last appears on the melted shadow stack. This phenomena creates much flexibility to attackers that carefully time the control paths and succeed in predicting those "last" addresses.

6.4 The CSOP Attack

In this section we present the *Context Switch Oriented Programming* (CSOP) attack. This attack uses the above techniques to bypass the shadow stacks protection.

Recall that our threat model assumes that the attacker has gained an arbitrary write primitive to the kernel memory. In particular, the attacker can overwrite the kernel stack, and the shadow stack identifier (or the SSP) stored in the process descriptor table.

We present two variants of the attack.

6.4.1 The Basic Attack: Manipulating the SSP

For the purpose of this subsection we consider a shadow-stack based solution that stores the SSP address in the process descriptor table. In this simple solution the shadow stacks are stored in a dedicated (protected) memory region, and switched by storing and loading the SSP addresses to/from the corresponding process descriptor during a context switch. The only verification preformed checks whether the new address is actually contained in a valid shadow stack memory region.[1]

In this scenario the attacker can use his ability to control the SSP in order to create overlapping frames on the shadow stack, by using the trimmed cascaded technique of Subsection 6.3.2, where the frames are trimmed on their prologue side. Thus, the addresses that force a return to user space after the control path is completed. The result is a shadow stack with a series of control paths separated by a call to the scheduler.

Notice that the attacker builds the shadow stack frames in a reverse order (with respect to their execution). Frame 1 is the first to be created, but the last to be executed. In that manner, the attacker overwrites the prologue addresses that force the return to user mode.

Once the attacker finish to overwrite the return addresses, he can execute a series of control paths in a manner that cannot otherwise be created by the scheduler. These control paths act as the widgets in ROP, and can be used for the benefit of the attacker. Moreover, if the attacker uses a target process with a high priority, all the widgets are later executed by the same process at the last stage of the attack, as illustrated in Figure 6.8. Since no actual context switch is preformed, the state of the process is preserved between the different widgets. Thus, the attacker is able to achieve a "ROP like" capability.

[1] Indeed Intel CET keeps the SSP in the kernel memory, and stores/loads the SSP during context switch.

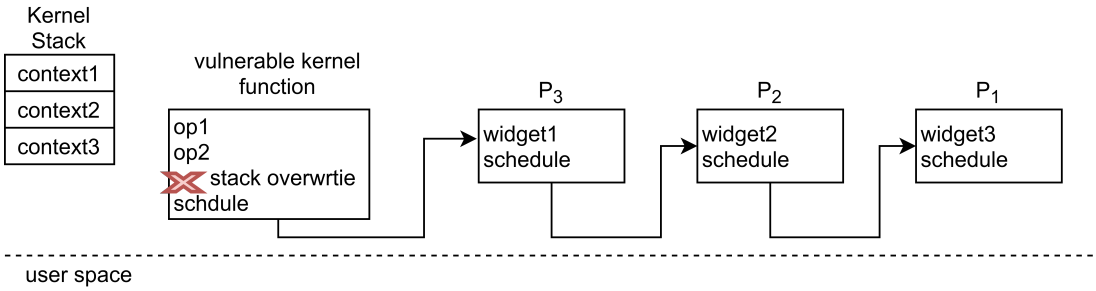


Figure 6.8: CSOP Attack Flow: Last Stage

A pseudo-code of the attacking algorithm is given in Algorithm 6.2. The content of

Algorithm 6.2 The Basic CSOP Attack

1: Create a new process P_0
2: $B_0 \leftarrow$ `sizeof`$(prologue\ of\ P_0)$
3: $\alpha_0 \leftarrow$ `top address of shadow stack of` P_0
4: **for** $i = 1$ to n **do**
5: Create a new process P_i in suspended mode
6: $S_i \leftarrow$ `sizeof`$(contorl\ path\ i)$
7: $B_i \leftarrow$ `sizeof`$(prologue\ of\ contorl\ path\ i)$
8: $\alpha_i \leftarrow \alpha_{i-1} + S_i$
9: $SSP[P_i] \leftarrow \alpha_i - B_{i-1}$
10: Release P_i so it invokes control path i
11: Suspend P_i after it is switched out
12: $SSP[P_n] \leftarrow$ `top address`
13: Overwrite P_n's kernel stack (and stack pointer) to fit the shadow stack return addresses with a content favorite for the attacker
14: Release P_n so it executes the CSOP chain

the shadow stack in some locations of this attack algorithm are outlined in Figure 6.9.

6.4.2 A More Powerful Attack on Multi-Core Systems

For the purpose of this attack we assume a multi-core system. On such a system several processes are running at any time, each using a shadow stack. Now, consider an attacker that forces two processes that are running simultaneously (on different cores) to use the same shadow stack.

If these two processes call kernel control paths roughly at the same time, they both write on the same shadow stack in some arbitrary order each using it own SSP register. So each of them may write its return address on top of another return address of the other, like presented in Subsection 6.3.4. The result is that return addresses of two different control paths are mixed on the same shadow stack without being separated by returns to user space.

By carefully selecting the control paths and their timing, while considering the computation time that it takes to the control paths between one procedure call to the

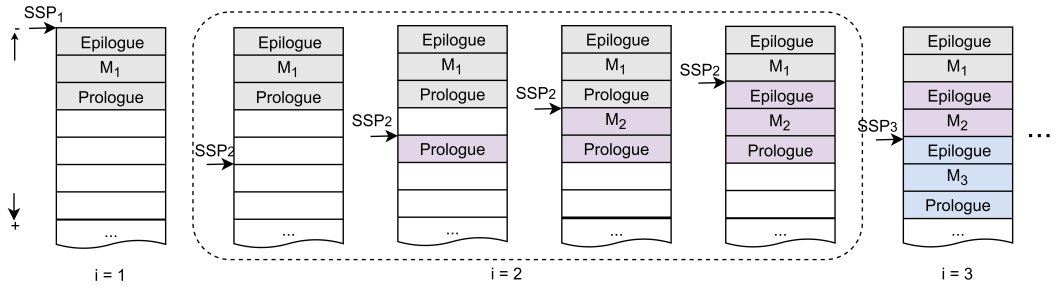


Figure 6.9: The Content of the Shadow Stack During the CSOP Attack

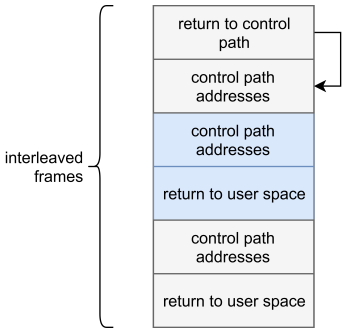


Figure 6.10: A Well-Manipulated Shadow Stack After Simultaneous Execution

next, the attacker can gain control over the creation of the melted shadow stack and manipulate it so it suits his needs. The attacker may preform several attempts until achieving a specific desired structure on the melted shadow stack. An examples of such shadow stack structure is outlined in Figure 6.10.

In such a manner, the attacker can create CSOP chains that could not have been generated from any existing control path. This capability significantly advances the attacker's ability to create useful CSOP chains. The rest of the attack follows the basic attack.

6.5 Discussion

TKRFP was designed with CSOP attacks in mind, in order to be protected against them. To protect itself from attacks that are based on SSP manipulation techniques TKRFP uses handles as stack-ids. These handles allow the OS kernel to choose which stack is used, but it cannot directly set the SSP value. TKRFP also forbids the simultaneous use of the same shadow stack by more than one process by verifying in the TKRFP_SET_ACTIVE command that an a shadow stack cannot be activated if it is already active.

Intel CET protects the SSP from being manipulated by writing a special token at the top of non-active shadow stacks. This token is used to verify that the SSP may only be loaded with addresses that were pointed by the SSP before the process was switched out. Even though the token is invalidated once used, we suspect that due to

synchronization issues two processes can be forced to use the same shadow stack. Hence, the attack described in Subsection 6.4.2 is likely possible on Intel CET.

The mitigations for CSOP include two types: Mitigations intended to prevent SSP manipulation, and mitigations intended to prevent simultaneous use of a shadow stack. In TKRFP we chose to protect the SSP by using handles, which point to a protected data structure that keeps the actual SSPs, while in Intel CET tokens, that are stored on top of the stack and uniquely identify it, are used. In order to prevent the simultaneous use TKRFP takes a direct approach by forbidding an active stack activation while it still active. On the other hand, Intel CET uses token invalidation, which is a riskier approach that relies on the fact that the to-be-invalidated token cannot be used before its invalidation. As said above, we expect that this is not a foolproof solution.

Designers of shadow-stack-protected solutions must be aware of the CSOP attacks and include the necessary mitigations in their design, since naive implementations of shadow stack solutions are not protected against them.

Chapter 7

Summary

In this thesis we discussed various aspects of trusted execution environments. We started by evaluating the security level of existing TrustZone-based TEE implementations, and offered ways to improve them. Then we showed how a TEE can effectively be used to improve the security of the main system by protecting it from return flow hijacking attacks.

7.1 TROOS: TRusted Open OS

At the beginning of Chapter 4 we described design flaws in current TrustZone-based TEE. In particular, we showed that the API (both external and internal) of these TEEs is too broad. This broadness is especially important, as the API commands are the main attack vector on the TEE, and as attackers frequently use vulnerabilities found in the API to mount successful attacks on these TEEs.

We then presented TROOS: a trusted OS for TrustZone-based TEEs designed to mitigate these flaws. TROOS is designed as a fully functional TEE while keeping a very limited attack surface. The attack surface reduction is achieved due to a carefully designed compact external API (which consists of only four commands). TROOS also limits its internal API by enforcing a capabilities based security policy on its TAs. Another important security feature of TROOS its ability to reduce it's TA TCB.

We believe that by implementing these concepts we can improve the security level of trusted execution environments and build more secure systems.

7.2 TKRFP: Trusted Kernel Return Flow Protection

In Chapter 5 we presented TKRFP: A TEE-based return flow protection. Our solution is capable of catching all return flow hijacking attacks (including ROP) against the MEE OS kernel. TKRFP offers effective protection even when considering a wide threat model, in which an attacker has arbitrary read and write primitives to the kernel memory.

We implemented a prototype for TKRFP, consisting of a TrustZone-based TEE implementation, and a compatible MEE. In particular, we created a customized Linux kernel and a GCC plugin that support TKRFP. We analyzed the performance of our prototype and showed that a TEE-based return flow protection is a feasible and efficient solution.

We showed that TKRFP can be incorporated on various systems and architectures and improve their security. Considering the reasonable performance overhead and the comprehensive protection that it provides, TKRFP offers a significant advancement in kernel protection.

7.3 CSOP: Context Switch Oriented Programming Attack

In Chapter 6 we presented CSOP: A new attacking technique against shadow stacks. This attack enables an attacker with arbitrary write privileges to bypass the protection offered by the shadow stacks, and thus to achieve arbitrary code execution. CSOP is a real threat to shadow-stack-based solutions, which must be considered when designing such solutions.

We also offer effective defenses to protect shadow stacks from CSOP, that we incorporated in TKRFP. We hope that other shadow-stack-based solutions, such as Intel's CET, will also consider CSOP in their threat model, and protect against it, in order to provide the best protection possible.

Bibliography

[1] ARM. Armv7-a architecture reference manual.

[2] ARM. Armv7-m architecture reference manual.

[3] ARM. Armv8-a architecture reference manual.

[4] ARM. Armv8-m architecture reference manual.

[5] ARM. Cortex-a. `https://developer.arm.com/products/processors/cortex-a`.

[6] ARM. Cortex-m. `https://developer.arm.com/products/processors/cortex-m`.

[7] ARM. Primecell infrastructure amba 3 trustzone protection controller. 2004.

[8] ARM. Security technology building a secure system using trustzone technology (white paper). 2009.

[9] ARM. Arm cortex-a series programmer's guide for armv8-a. 2015.

[10] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[11] Gal Beniamini. Getting arbitrary code execution in trustzone's kernel from any context. `http://bits-please.blogspot.com/2015/03/getting-arbitrary-code-execution-in.html`.

[12] Gal Beniamini. Trust issues: Exploiting trustzone tees. `https://googleprojectzero.blogspot.co.il/2017/07/trust-issues-exploiting-trustzone-tees.html`.

[13] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[14] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[15] The Linux Kernel documentation. Kernel self-protection. `https://www.kernel.org/doc/html/latest/security/self-protection.html`.

[16] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305. ACM, 2017.

[17] Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. The andix research os–arm trustzone meets industrial control systems security. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 88–93. IEEE, 2015.

[18] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003.

[19] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[20] GenodeLabs. Genode os framework. `https://genode.org/`.

[21] GlobalPlatform. `https://globalplatform.org/`.

[22] GlobalPlatform. Tee client api specification v1. Technical report, GlobalPlatform Inc, 2011.

[23] GlobalPlatform. Tee internal api specification v1.0. Technical report, GlobalPlatform Inc, 2011.

[24] GlobalPlatform. Tee system architecture. Technical report, GlobalPlatform Inc, 2011.

[25] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, page 2. ACM, 2017.

[26] Intel. 64 and ia-32 architectures software developer manual. *Volume 3B: System programming Guide*, 2, 2011.

[27] Intel. Software guard extensions programming reference, revision 2, 2014.

[28] Intel. Control-flow enforcement technology preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2017.

[29] Eyal Itkin. Bypassing return flow guard (RFG). https://eyalitkin.wordpress.com/2017/08/18/bypassing-return-flow-guard-rfg/, 2017.

[30] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: Epid provisioning and attestation services. *White Paper*, 1:1–10, 2016.

[31] Tencent Security Xuanwu Lab. Return flow guard. https://xlab.tencent.com/en/2016/11/02/return-flow-guard/, 2016.

[32] K Lady. Sixty percent of enterprise android phones affected by critical qsee vulnerability. https://duo.com/blog/sixty-percent-of-enterprise-android-phones-affected-by-critical-qsee-vulnerability.

[33] Donald C Latham. Department of defense trusted computer system evaluation criteria. *Department of Defense*, 1986.

[34] Jochen Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.

[35] Linaro. Op-tee. https://www.op-tee.org/.

[36] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium*, pages 549–564, 2016.

[37] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP@ ISCA*, 2013.

[38] Microsoft. Virtualization-based security (vbs). https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs, 2017.

[39] National Vulnerability Database (NVD). https://nvd.nist.gov/.

[40] Feske Norman. *Genode Foundations*.

[41] NVD. CVE-2015-6639. https://nvd.nist.gov/vuln/detail/CVE-2015-6639.

[42] NVD. CVE-2017-1000410. https://nvd.nist.gov/vuln/detail/CVE-2017-1000410.

[43] NVD. CVE-2017-14954. https://nvd.nist.gov/vuln/detail/CVE-2017-14954.

[44] NVD. CVE-2018-7755. https://nvd.nist.gov/vuln/detail/CVE-2018-7755.

[45] Offensive Security. Msfrop. https://www.offensive-security.com/metasploit-unleashed/msfrop/.

[46] OMTP. Advanced trusted environment. Technical report, 2009.

[47] OP-TEE Design. The pager. https://github.com/OP-TEE/optee_os/blob/master/documentation/optee_design.md#8-pager.

[48] Qualcomm. Qsee. https://www.qualcomm.com/solutions/mobile-computing/features/security.

[49] Nguyen Anh Quynh. Optirop: the art of hunting rop gadgets. In *Black Hat conference*, 2013.

[50] Dan Rosenberg. Unlocking the motorola bootloader. *Azimuth Security Blog*, 2013.

[51] Dan Rosenberg. Qsee trustzone kernel integer over flow vulnerability. In *Black Hat conference*, 2014.

[52] Joanna Rutkowska and Rafal Wojtczuk. Qubes os architecture. *Invisible Things Lab Tech Rep*, 54, 2010.

[53] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[54] Jonathan Salwan and Allan Wirth. Ropgadget. https://github.com/JonathanSalwan/ROPgadget.

[55] Daniel Sangorrin, Shinya Honda, and Hiroaki Takada. Dual operating system architecture for real-time embedded systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium*, pages 6–15, 2010.

[56] SensePost. A software level analysis of trustzone os and trustlets in samsung galaxy phone. https://www.sensepost.com/blog/2013/a-software-level-analysis-of-trustzone-os-and-trustlets-in-samsung-galaxy-phone.

[57] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[58] Richard M Stallman. Gnu compiler collection internals. *Free Software Foundation*, 2002.

[59] JEDEC Standard. Embedded multimedia card (emmc), electrical standard 4.51. *JEDEC Solid State Technology Association, JESD84-B451 (revision ofv JESD84-B45*, 264:73–83, 2011.

[60] Gary Stoneburner, Alice Y Goguen, and Alexis Feringa. Sp 800-30. risk management guide for information technology systems. 2002.

[61] Andrew S Tanenbaum. The tanenbaum-torvalds debate, 1999.

[62] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, (5):44–51, 2006.

[63] TCG. Trusted platform module summary. `http://www.trustedcomputinggroup.org/resources/trusted_platform_module_tpm_summary`, 2009.

[64] PaX Team. Pax address space layout randomization (aslr). 2003.

[65] Trustonic. Secured Platforms. `https://www.trustonic.com/solutions/trustonic-secured-platforms-tsp`.

[66] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium. USENIX Association*, 2018.

[67] Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan M McCune. Trustworthy execution on mobile devices: What security properties can my mobile platform give me? In *International Conference on Trust and Trustworthy Computing*, pages 159–178. Springer, 2012.

[68] Cliff White. Performance testing the linux kernel. In *Linux Symposium*, page 457, 2003.

[69] Widevine. Widevine DRM.

[70] John Williams. Inspecting data from the safety of your trusted execution environment. *BlackHat USA*, 2015.

[71] Joseph Yiu. Armv8-m architecture technical overview. *ARM WHITE PAPER*, 2015.

[72] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices.

העבודה שלנו מתמקדת בדרכים בהן סביבת מחשוב אמינה יכולה לשפר את האבטחה של מערכות מחשוב. ביצענו ניתוח של מימושי סביבות מחשוב אמינות וזיהינו פגמים מסויימים בתכנון האבטחה שלהם. על בסיס הניתוח הזה פיתחנו את TROOS, מערכת הפעלה לסביבות מחשוב אמינות אשר מתמודדת עם הפגמים שמצאנו. בנוסף, פיתחנו את TKRFP: מנגנון הגנה אשר משתמש בסביבת מחשוב אמינה כדי להגן על מערכת ההפעלה מפני התקפות מסוג return flow hijacking. TKRFP מוסיף שכבת הגנה למערכת ההפעלה ובכך משפר את רמת האבטחה הכוללת של המכשיר. לבסוף פיתחנו סוג חדש של התקפה: תיכנות מונחה החלפות הקשר אשר יכולה בעיקרון לעקוף את ההגנות שמספקת מחסנית צללים למניעת התקפות מסוג return flow hijacking. אנו דנים גם באמצעי ההגנה הנדרשים כנגד התקפה חדשה זו.

תקציר

התלות במערכות מחשוב גדלה באופן מתמיד. אנחנו נסמכים על מערכות אלה כמעט בכל תחום בחיי היומיום שלנו ומפקידים בידיהן את המידע הרגיש ביותר שלנו, ואת תהליכי קבלת ההחלטות הקריטיים ביותר. לדוגמה, טלפון נייד סטנדרטי עשוי להכיל מידע ביומטרי של בעל המכשיר, מספר כרטיס אשראי ופרטי הזדהות של חשבון הבנק שלו. בנוסף, עשויים להישמר בו גם היסטורית הגלישה באינטרנט, קבצים, היסטוריית מיקומים, רשימת חברים וסוגים נוספים של מידע פרטי ורגיש.

מנקודת המבט של תוקף, הרווח הפוטנציאלי של התקפה מוצלחת על מכשירים אלה הוא מאוד משמעותי. לפיכך, מוכנים התוקפים להשקיע מאמצים הולכים וגדלים כדי בתיכנון ומימוש התקפות בעלות רמת תחכום גבוהה. אותם תוקפים משתמשים במגוון טכנולוגיות כדי לתקוף בעקביות טווח רחב של מטרות כמו ממשלות, מוסדות ואנשים פרטיים.

סוג מעניין במיוחד של התקפות, הן התקפות המכוונות כנגד מערכת ההפעלה של הקורבן. מערכת הפעלה מודרנית היא תוכנה בעלת רמת מורכבות גבוהה ביותר המכילה מליוני שורות קוד, בשל כך היא מועדת לטעויות ועל כן פגיעה להתקפות. התקפה על מערכת ההפעלה היא בעלת משמעות הרסנית. תוקף המצליח להשתלט על מערכת ההפעלה יכול לעקוף את מדיניות האבטחה של המכשיר, אשר נאכפת בדרך כלל על ידי מערכת ההפעלה. יתרה מכך, תוקף אשר משתלט בהצלחה על מערכת ההפעלה מקבל שליטה מוחלטת על המערכת, הכוללת שליטה על כל תוכניות המשתמש, שליטה על קבצים ועל התקני קלט/פלט מסוגים שונים. התוקף יכול אפילו לעקוף מנגנוני הגנה, כגון אנטי־וירוס, מכיוון שהזמינות והשלמות של מנגנונים אלה תלויה בשירותים המסופקים על ידי מערכת ההפעלה. לאור עובדות אלה ניתן להסיק כי מערכת ההפעלה היא מטרה בעלת ערכיות גבוהה במיוחד לתוקפים.

האיום הנ"ל אינו תיאורטי בלבד. לדוגמה SubVirt הינה נוזקה בעלת יכולת לשבש ולעקוף מנגוני אבטחה של מערכת ההפעלה. SubVirt משתמשת ביכולות וירטואליזציה של החומרה כדי להשתלט על מערכת ההפעלה ולגרום לה לרוץ בסביבה וירטואלית אשר נשלטת על ידה. נוזקה כזו היא קשה במיוחד לגילוי מכיוון שהיא יכולה להשפיע בעורמה על כל פעולה שעשויה לגלות אותה. BluePill היא נוזקה דומה בעלת יכולות התחמקות מזיהוי טובות אף יותר, מכיוון שהיא תוכננה כך שלא תשאיר כל זכר אשר עשוי לחשוף אותה.

כדי להיות מסוגלים לשמור על אבטחת המערכת גם במקרה שתוקף מצליח להשתלט על מערכת ההפעלה, נדרש להוסיף רכיב אמין מבודד למערכת. דוגמה אחת לסוג כזה של רכיב היא סביבת מחשוב אמינה. סביבת מחשוב אמינה יכולה לקחת חלק בהגנה על מערכת ההפעלה מפני התקפות או לשמור מידע רגיש מחוץ להישג היד של מערכת ההפעלה ובכך למנוע את דליפתו במידה שתוקף השתלט עליה. סביבת מחשוב אמינה יכולה גם לבצע פעולות רגישות עבור המערכת הראשית.

המחקר בוצע בהנחייתו של פרופסור אלי ביהם ודוקטור שרה ביתן, בפקולטה למדעי המחשב.

תודות

בראש ובראשונה, אני רוצה להודות למנחיי, פרופ' אלי ביהם וד"ר שרה ביתן, על התמיכה הרבה לאורך העבודה על התזה הזאת. תודה לכם על שנתתם לי את החופש למצוא את דרכי, תוך כדי הנחייה והכוונה. למדתי מכם רבות.

אני רוצה גם להודות להוריי, אחיי ומחותנייי על תמיכתם לאורך התקופה העמוסה הזאת. תודה שהייתם שם בשבילי בכל פעם שנזדקקתי לכך. תודה מיוחדת מוקדשת לשלושת ילדי המקסימים, אתם ממלאים את חיי באושר.

התודה הגדולה מכולן היא לאישתי, ליאור. תודה לך על התמיכה האינסופית, על שעות רבות בהן לקחת על עצמך יותר כדי לתת לי את הזמן הדרוש לעבוד ועל כך שאת תמיד דוחפת אותי קדימה. את האור של חיי ואני אוהב אותך מאוד.

אני מודה לטכניון, למרכז המחקר לאבטחת סייבר ע"ש הירושי פוג'יווארה ולמערך הסייבר הלאומי של ישראל על התמיכה הכספית הנדיבה בהשתלמותי.

סביבות מחשוב אמינות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

אסף רוזנבאום

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
כסלו התשע"ט חיפה נובמבר 2018

סביבות מחשוב אמינות

אסף רוזנבאום