Overseer: Enforcing fine-grained memory access control across execution environments

Anonymous Authors

ABSTRACT

TrustZone enables Rich Execution Environments (REEs) and Trusted Execution Environments (TEEs) to share physical memory by building virtual address spaces without knowing each other's mapping. While TEE hardware protection guarantees that the REE ("Normal World") is restricted to accessing non-secure memory exclusively, the Secure OS and Trusted Applications (TAs) operating within the TEE ("Secure World") can map physical memory belonging to REE OS and applications. As a result, vulnerabilities within TEE code, in conjunction with the inherent semantic gap between the REE and TEE can be leveraged by malicious actors to completely bypass REE security policies and leak or compromise REE-sensitive data or code. Our comprehensive analysis of TAs from commercial TrustZone devices and associated CVEs in the past decade has revealed numerous REE and TEE physical memory access vulnerabilities across multiple vendors. To mitigate the security impact of this critical semantic gap, we introduce Overseer, a secure monitor mechanism that prevents unauthorized physical memory mappings by TEEs, one of the important enablers of compromise. Overseer further enables the Secure OS and the TAs running inside the TEE to regulate the invocation of Secure Monitor Calls (SMCs) and System Calls. Overseer's performance overhead is under 2% on average. A detailed CVE analysis shows that Overseer can mitigate the attack surface of over 53 TA vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; **Access control**.

KEYWORDS

Hardware Security; Access Control; ARM TrustZone

ACM Reference Format:

Anonymous Authors. 2026. Overseer: Enforcing fine-grained memory access control across execution environments. In *Proceedings of ACM Asia Conference on Computer and Communications Security (ASIA CCS'26).* ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

TEEs are designed to isolate security sensitive applications from REEs which include rich operating systems and associated untrusted, potentially compromised applications. ARM TrustZone [1] implements TEE which has gained widespread use mainly due to ARM's mobile device marketshare. In TrustZone, the *"Secure World"* operates at a higher privilege level isolated in hardware from the *"Normal World"* REE. Inside the TEE, "Trusted Applications" (TAs) are run by the Secure OS and handle sensitive operations such as storing encryption keys or authenticating biometric data on behalf of REE processes.

Upon start, the TrustZone bootloader isolates TEE memory (a subset of physical memory) by tagging it as "secure", thus ensuring that the *Normal World* REE cannot access it, whereas the Secure World can access both secure and non-secure portions. This enables the two environments to communicate through the use of non-secure memory. TAs and Normal World applications can communicate by setting up shared non-secure physical memory in their virtual address spaces.

However, this isolation asymmetry also further enables an inherent critical inter-world semantic gap between the two environments. The Secure World can directly manipulate Normal World physical memory without the Normal World's knowledge, and without consideration of its use inside the Normal World virtual address space. For example, the Secure World might maliciously or unknowingly overwrite memory used for Normal World OS code and data pages. While the TEE-assigned memory is protected by hardware against REE access, the asymmetric protection renders the REE-imposed memory access policies incapable of preventing unauthorized access from within the TEE (i.e. compromised TAs and Secure World OS).

Previous work [2, 3, 4] has shown the importance of limiting the access of potentially compromised TAs in terms of both TEE and REE access. As a result, commercial TrustZone-based TEEs have incrementally introduced software-based protection (e.g., pointer sanitizing, hard-coded constraints inside TA and kernel binaries). However, these techniques only aim to prevent very specific attack vectors (e.g., shared memory, privilege escalation into TEE kernel). An unknown number of vulnerabilities remain hidden within TAs that could be leveraged into accessing or modifying target TEE or REE physical memory. More importantly, the REE applications and OS remain incapable of controlling TEE access into their assigned physical memory and are rendered powerless against TEE compromise.

Prior work has also shown how the inter-world semantic gap in conjunction with vulnerable TA code enables malicious Normal World applications to trick TAs into compromising the Normal World through Boomerang [5] or Horizontal Privilege Escalation (HPE) [6] attacks. In a Boomerang attack, a TA is tricked into operating on memory belonging to a victim Normal World process or OS by unsanitized pointers provided from a malicious Normal World process. Similarly, in an HPE attack, a TA is tricked into providing sensitive information stored inside the Secure World by the use of un-validated parameters originating in the Normal World. Both Boomerang and HPE attacks stem from the same inter-world

ASIA CCS'26, June 1-5, 2026, Bangalore, India

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

semantic gap that prevents the Secure World OS from validating information originating in the Normal World.

In HPE and Boomerang attacks, TAs are considered not fully compromised but rather "confused deputies" with little to no agency or directed purpose. There are defenses in this setting. For example, Cooperative Semantic Reconstruction (CSR) [5] assists the TA in verifying that a target memory address belongs to the requesting Normal World application before access. However, the defense efficacy relies on TA's adherence to the CSR's protocol. If the TA is fully controlled by the attacker or malicious, defense will be bypassed as the TA can deviate from the protocol.

In this work, we explore such threats where TAs are fully compromised. This is a practical reality. For example, a Normal World application can obtain arbitrary code execution within a TA by exploiting vulnerabilities in the TA implementation [7, 8]. Once compromised, a TA can be leveraged by a malicious CA to access Normal World memory, including potentially arbitrary regions.

[Bin: Rewrite of the three paragraphs above] Prior work also [5, 6] demonstrates how the inter-world semantic gap, combined with vulnerable TA code, enables malicious Normal World applications to leverage TAs for compromising the Normal World. For example, in the Boomerang attack [5], a malicious Normal World process tricks a TA into operating on memory belonging to a victim Normal World process or OS by providing unsanitized pointers. In this scenario, TAs act not as fully compromised entities but as "confused deputies", blinded by the semantic gap that prevents the Secure World OS from validating information originating in the Normal World. To bridge the gap, Cooperative Semantic Reconstruction (CSR) [5] was proposed to assist the TA in verifying memory ownership against the requesting application in the Normal World. However, CSR's efficacy relies on TA's adherence to the CSR's protocol. If a TA is malicious or fully compromised, the attacker can bypass CSR by deviating from the protocol.

In this work, we explore the threat of fully compromised TAs. This is a practical reality, as Normal World applications can obtain arbitrary code execution within a TA by exploiting its implementation vulnerabilities [7, 8]. Once compromised, a TA can be leveraged by a malicious CA to access arbitrary regions of Normal World memory. There are potential defenses. For example, ReZone [9] partitions Secure World into sandboxed *zones* that provide separate execution environments for the Secure OS and TAs. Compromising a zone will not allow the attacker to access arbitrary memory regions. However, limitations of ReZone include (i) its reliance on additional hardware that may not be available on commercially off-the-shelf platforms or requires a hardware-specific implementation, and (ii) hardware granularity constraints that can force co-located cluster cores to halt during a single core's zone entry-exit period, introducing non-trivial performance overhead.

To address the fundamental risks posed by compromised Secure World components, we propose Overseer, a software-based mechanism that enforces fine-grained access control across TrustZone execution environments. Operating at the Secure Monitor, Overseer ensures that access to Normal and Secure World physical memory is explicitly authorized by policies defined by the memory owner and further reduces the attack surface by policy-enforcing Secure Monitor Calls (SMCs) and Secure OS system calls. Overseer is TrustZone-implementation agnostic, requires minimal code changes, and incurs insignificant overhead, with TEE benchmarks showing an average performance impact of less than 2%.

To further understand how TEE related CVEs and other attack vectors allow a vulnerable TA to manipulate Normal World memory, we examined 125 TA binaries from mobile devices running three popular TEEs: Trustonic's Kinibi, Qualcomm's QSEE, and Samsung's Teegris. Among these binaries, using a combination of manual analysis and symbolic binary execution, we identified numerous TAs that can easily map and subsequently compromise Normal World memory, thus presenting a critical threat to Normal World security.

Complex attack vectors are introduced when a TA is provided either direct or indirect access to Normal World physical memory and security-critical TEE resources. Figure 1 illustrates how a malicious Normal World application can target vulnerable TAs, compromise them and leverage their access to further escalate their privileges either in Normal or Secure World. We also analyzed around 77 TA-related CVEs and found that in more than 68%, the attacker could copy or modify arbitrary Normal or Secure World memory locations.

Thus, careful management of physical memory mappings is crucial for ensuring the integrity and security of both the worlds. A fundamentally different system is required to give back applications and OSes the ability to determine who can access their memory in terms of both REE and TEE processes. This system needs to monitor the mapping of memory in both worlds and introduce system-wide fine-grained control based on the principle of least privilege [10].

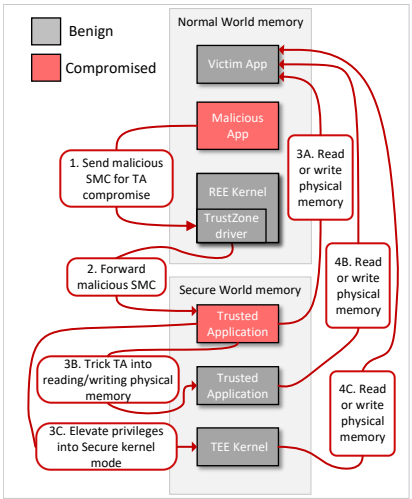


Figure 1: An illustration of how a vulnerable TA can be exploited to compromise Normal World memory

2 BACKGROUND

Trusted Execution Environments (TEEs) establish secure enclaves within processors, offering a protected realm for code and data execution that is isolated from other software on the same device. The code executed within TEEs benefits from a heightened level of trust compared to typical applications and operating systems, which reside in the Rich Execution Environment (REE) with lower privilege levels. To prevent interference, a hardware-based isolation

mechanism ensures that REE code cannot impact TEE execution or access resources exclusively allocated to the TEE environment.

2.1 TrustZone architecture

ARM Cortex processors incorporate the hardware aspect of a TEE by implementing the ARM TrustZone security extensions, or *TrustZone*. TrustZone divides the hardware and software resources so that they either exist in the *Secure World (SW)* for the security subsystem or the *Normal World (NW)* for the rest of the software, based on the special *Non-Secure (NS)* bit. Each physical core is split into two logical cores that either runs the *Secure World* TEE software when NS=0 or the *Normal World* REE software if NS=1. Each memory region and I/O peripherals also maintains the NS bit giving a completely separate physical address space for NW and SW.

TrustZone guarantees that software running in the Normal World can only make Non-Secure memory accesses, while software running in the Secure World has the ability to make both secure and non-secure memory accesses. Hence, secure operations, such as accessing secure keys or fingerprint data, require an explicit transfer of control from the Normal World to the Secure World, commonly known as a *world switch*.

The management of core execution (e.g., toggling between the Normal and Secure World using the NS bit) is done by software running at a higher privilege level above user and supervisor, named *Monitor* mode. Monitor code execution is triggered using interrupts or special CPU instructions named *Secure Monitor Calls (SMCs)*.

Like a conventional operating system, a TrustZone Secure OS (TZOS) operates within the Secure World, running at a higher privilege level. It takes charge of resources, device drivers, and the management of a collection of Trusted Applications (TAs) which provide task-specific functionality running at user privilege level. Typical TA use cases include managing keys, performing cryptographic operations, secure storage etc, for applications such as Digital Rights Management, Biometric authentication, and mobile payments [11]. The TAs run in their own address space and make requests to the TZOS for any I/O or IPC, using system calls. We use the terms Secure OS and TZOS interchangeably throughout the paper.

ARM processors incorporate different exception levels both in NW and SW which represent distinct privilege levels to establish a hierarchical structure for managing interrupts, exceptions, and system calls. EL0 (Exception level 0) is the least privilege level where applications run. EL1 runs the OS kernel and its associated functions like system calls, interrupts and exceptions. EL2 is the Hypervisor mode that provides virtualization support. EL3 runs the Secure Monitor. Transitions between Exception levels occur exclusively when an exception is triggered or when returning from an exception. An Exception level, denoted as ELn, is considered higher than another Exception level when its assigned value of n is greater. For instance, EL3 is regarded as a higher Exception level than EL1.

2.2 TrustZone communication

Requests to TA can originate from the Normal World Client Applications (CA) or from other TAs. Communication between CA and TA typically operates under a Client-Server model. First, a CA process establishes a connection to a TA process. Then, the CA issues requests to the TA process, which responds back with messages containing either results or error messages. However, TrustZone prevents the two processes from exchanging messages directly, as SMCs can only be issued in supervisor mode. Therefore, Normal and Secure World kernels act as intermediaries in their communication. For example, when a CA requires a digital signature, it copies the inputs in a shared buffer and informs the Secure Monitor (running at EL3) which TA it wants to forward the request, via an SMC call. The Monitor interrupts the TZOS, which then invokes the TA to process the request and return the results in the shared buffer. The shared buffer format depends on the underlying TZOS. For example, within Kinibi, a common buffer is utilized for both input and output purposes. Conversely, QSEE and Teegris offer distinct input and output buffers.

3 INVESTIGATING TA SECURITY AND CAPABILITIES

Most TrustZone-enabled commercial devices run under either a QSEE, Kinibi or Teegris TZOS [12]. In this section we present a study of the TAs operating under these three TZOSes and show how adversaries can leverage vulnerable TAs to escalate their privileges into security-critical processes inside Normal and Secure World.

To investigate, we extracted TA binaries from the newest TrustZone-enabled mobile devices running each TZOS. A total of 125 TA binaries were extracted from three commercial firmware images, each designed to run under either the QSEE, Kinibi or Teegris TZOS.

Coarse-grained TA access Under QSEE all TA binaries execute under the same privileges and the concept of TA Drivers does not exist. In contrast, Kinibi and Teegris have introduced the notion of TA Drivers to only provide a subset of TAs to access the pool of security-critical system calls (e.g., physical memory mapping). Regular TAs under Kinibi and Teegris have limited functionalities and instead typically have to rely on TA drivers for operations such as reading biometrics or copying data into physical memory belonging to Secure World I/O devices. However, access to TA drivers is largely unrestricted. Any TA can access driver-exposed APIs. Note, both TA and TA drivers execute at the same EL0 exception level. The main difference between them lies in the set of Secure OS functions (syscalls) they can utilize through their embedded libraries.

3.1 Approach

Upon reverse engineering the extracted binaries, we have categorized them into 50 unique QSEE TAs, 34 Kinibi TAs among which 8 are TA drivers and 41 Teegris TAs that include 10 TA drivers. With the exception of 3 Teegris TAs binaries that were encrypted, a combination of manual and automatic analysis was used to determine the reach of all other TAs in terms of both Normal and Secure World. Using manual investigation of TA and kernel binaries we have identified the set of security-critical operations utilized by TAs to read and write physical memory belonging to Normal World and I/O devices. Under QSEE all 50 TA binaries examined include an identical set of APIs that correspond to kernel system calls. These

APIs enable all QSEE TAs to read and write physical memory. Thus, escalating privileges across QSEE TAs does not provide adversaries with additional access to physical memory. In contrast, under Kinibi and Teegris only TA Drivers are provided with access to physical memory mapping APIs. To accurately determine the limitations imposed on Kinibi and Teegris TAs, we further examined whether TA drivers provide TAs with indirect access to physical memory mapping using automatic binary analysis as below.

Automated TA Driver binary analysis TA Driver execution is simulated by leveraging angr's [13] symbolic execution framework and our reverse-engineering findings. Any constraints imposed on physical memory mapping and input parameters within TA Driver binary code are automatically extracted and manually examined to determine possible indirect physical memory access.

angr is setup to simulate the execution starting from the TA entry point and searching for all execution paths containing physical memory mapping APIs. To verify all execution paths are explored, the memory containing input and output parameters is made symbolic. The input parameters correspond to a command ID issued by either a Normal World client application or another TA and buffers containing the respective input/output data.

The parameters made symbolic are preserved during the symbolic execution of TA binary code, propagating automatically through arithmetic operations. Further, external functions (e.g., kernel APIs, IPCs) and libraries are simulated by leveraging angr's simulated procedures (SimProcedures). To ensure the symbolic parameters are preserved, SimProcedures are carefully constructed to simulate the behavior of their corresponding external function. Note, these SimProcedures are necessary as our angr binary analysis framework cannot emulate the entire Secure OS required to directly execute these external functions. Thus, we have instead manually analyzed their operations and simulated all their potential side-effects inside the angr symbolic execution.

Once TA Driver execution reaches a physical memory mapping API, all constraints imposed on the symbolic input parameters are collected and examined. An indirect physical memory access is reported if symbolic input parameters are directly used by the TA driver for mapping physical memory. For example, writing physical memory according to symbolic input represents indirect physical memory access, controlled within the TA Driver only by the constraints imposed on the respective input symbols. Of course, further examination of symbolic input parameters, memory mapping APIs within the kernel and Secure Monitor binaries are used to determine the full reach of TA Drivers with regards to physical memory.

Overall, the automatic binary analysis process has revealed a set of dangerous execution paths that reach sensitive operations such as writing to arbitrary physical memory. Manual analysis was employed to analyze these execution paths and identify all other potential ones that could reach the same sensitive operations.

3.2 Investigation findings

The security of TEE provided APIs is not uniform across all devices, even after GlobalPlatform's *TEE Internal Core API Specification* [14] has been introduced. Each TEE has introduced various constraints at different execution levels to mitigate the security impact of vulnerabilities within TEEs.

Physical memory access	Kinibi	QSEE	Teegris
TAs with direct access	8 / 34	50 / 50	10 / 41
TAs with indirect access	15 / 34	-	15 / 41
Total	23 / 34	50 / 50	25 / 41

Table 1: TAs with capabilities to access Normal World physical memory by either directly mapping it inside their address space or through communication with other TAs. In total, 98 out of 125 TAs (78.4%) have such capabilities.

Mitigations against TA compromise To protect TAs against malicious requests, multiple techniques which are standard in Normal World have been introduced inside commercial TEEs. First, all investigated TEEs have introduced at the time of writing TA-level ASLR and Stack Cookies as mitigations against TA vulnerability exploitation. Further, they mark Normal World memory as Execute-Never inside the Secure World MMU. Several TA binaries are even stored encrypted, making the identification of vulnerabilities difficult for both adversaries and independent researchers. However, previous work [3, 2] has shown that vulnerable TAs remain prone to Normal World compromise, and how they can be used to obtain control over both TEE and REE.

Shared memory mapping Communication between CAs and TAs happens over shared memory. However, all memory references have to be converted to a common entity before being shared with the other world. Previous work [5] has shown that unverified pointers contained within shared physical memory can trick TAs into unknowingly changing their own private data. However, under Kinibi and QSEE the TA and CA shared memory is setup using virtual addresses in the TAs address space, which are also forwarded to the CA. Similarly, Teegris constructs shared memory buffers at random virtual addresses, which cannot be directly controlled by either the CA or TA. Thus, a compromised TA cannot control where the shared memory is mapped both in Normal and Secure World. Additional vulnerabilities are required to trick the TEE or Normal World kernel into malicious mapping of shared memory.

Physical memory mapping We observed that several TAs also require mapping specific physical memory addresses into their address space. This mapping is facilitated under all three TEEs through kernel provided APIs. Under QSEE any TA can request mapping of targeted physical memory, while Kinibi and Teegris restrict access to memory mapping APIs to TA drivers. However, analysing driver binaries, we have learnt that introducing memory mapping API restrictions does not necessarily prevent TAs from indirectly reading and writing physical memory data.

Table 1 presents the breakdown of the TA binaries analyzed for each vendor, indicating how many TAs are capable of obtaining direct or indirect access to physical memory regions. Direct access is obtained whenever a TA (or TA driver) can request the TEE kernel to map a target physical memory inside their address space. Indirect access corresponds to access obtained to physical memory through communication with TA drivers. Specifically, under both Kinibi and Teegris a TA driver enables 15 TAs to read or write data located in arbitrary Normal World physical memory. We consider a TA with direct or indirect access to physical memory to be dangerous and

potentially malicious (assuming the TA has been compromised) if the respective TA access lacks constraints that prevent it from mapping Normal or Secure World memory. For example, if the input constraints only prevent a TA from accessing Secure World kernel physical memory, we consider it a vulnerability as it could freely read or alter Normal World data.

Of course, TEEs do not provide TA with arbitrary access to physical memory. Restrictions are imposed on the memory mapping APIs provided, to prevent direct TA access into TEE kernel and Secure Monitor memory. However, no further limitations are introduced regarding TA access into physical memory of Normal World, as some TAs cannot operate without it. The TEE kernel and Secure Monitor lacks the knowledge required to enforce fine-grained access into Normal World memory. As a result, Normal World is left helpless against compromised TAs capable of accessing physical memory.

4 THREAT MODEL

When discussing attack vectors pertaining to memory accesses present in TrustZone devices, there are numerous ways (e.g., privilege escalation, side-channels, etc.) in which an adversary might obtain direct or indirect access to data inside memory pages belonging to a target process. As mentioned in Section 2, there are five elements involved in a typical TrustZone system: 1) CAs 2) Normal World OS 3) Secure Monitor 4) Secure OS 5) TAs. In a mobile environment, a CA usually refers to a mobile application operating on an Android, Windows, Linux, or similar operating system (referred to as the Normal World OS). Secure Monitor, Secure OS and TAs are developed by the TEE vendors (e.g., Trustonic, Qualcomm, Samsung etc). Overseer is designed to run within the Secure Monitor at EL3.

When vulnerabilities exist inside Secure World, an adversary-controlled CA can induce TA compromise by sending a malicious SMC request. As illustrated in Figure 1, such an attack can allow the adversary to escalate its privileges in both Normal and Secure World due to the inter-world semantic gap. Historically, such vulnerabilities have been prevalent in commercial TrustZone implementations. Thus, in this paper we construct our threat model assuming the adversary initially has control over a TA and can also communicate with other benign TAs to trick them into accessing memory through the capabilities presented in Section 3.2. The assumed goal of such an adversary is to gain unauthorized access to a set of target memory pages belonging to processes or OSes running in Normal and Secure World that it cannot directly compromise.

In addition to the assumption that attacks originate from a compromised or malicious TA, we also consider CAs to be untrusted. Thus, CAs, TAs, Normal and Secure OSes must all define policies that restrict any unauthorized access.

We assume the adversary can obtain control over the TA code, access its entire address space, and use Secure OS provided system calls to map additional memory inside its address space or communicate with other applications running inside Normal or Secure World. The Normal and Secure OS supply essential information needed for policy evaluation, including extracting policies, forwarding them, and triggering enforcement by passing the relevant policy data. We therefore assume that the adversary does not compromise only the policy-enforcement components in either OS or in the Secure Monitor (for example, using techniques in [15, 16]). All other components of the Normal and Secure OS are assumed to be vulnerable to exploitation. We also assume the standard TEE isolation is in place, preventing Normal World from directly accessing Secure World resources. Attacks in the Normal World that maliciously access memory pertaining to other processes and OS within the Normal World are outside the scope of the current work.

In this paper we only consider attack vectors consisting of either direct access through mapping of physical memory inside compromised TAs or indirect access to the respective memory through SMCs, system calls or TA provided APIs. But as detailed in Section 7.3, any indirect attacks that trick benign trusted applications that have access to target memory, SMC or system calls, cannot be prevented. Attack vectors and access policies pertaining to storage (e.g., file), caches or other hardware used to store sensitive information are not considered. Our goal in this paper is not to completely prevent the effects of adversaries compromising TAs or escalating their privileges into other Normal or Secure World components. Instead, we aim to limit the impact of such attacks by (i) ensuring a set of predefined memory access policies are always enforced and (ii) introducing fine-grained access policies in terms of TA and kernel provided APIs.

5 OVERSEER

5.1 Fine-Grained Access Control

The analysis in Section 3 has shown that TEE-enforced security does not provide system-wide protection against compromised TAs. As a result, the Normal World is rendered powerless against malicious TA physical memory access, regardless of the fact that most TAs do not require the ability to map physical memory at all. To minimize the impact of TA compromise on system security, we propose the following properties as necessary for enforcing fine-grained access control:

(1) Access to physical memory, SMCs, and system calls must be allowed only through policies.
(2) REE policies should control which REE memory TEEs can access.
(3) Memory access control should not prevent the usage of free (unmapped) memory.
(4) Access policy updates should not require code modification.
(5) Access must be expressly permitted through policies and denied otherwise.

To satisfy the first property, access control enforcement must first restrict all access in terms of TEE system calls, physical memory mapping, and SMC communication. Access to these resources must be provided through predefined signed policies. For SMC-based communication, the recipient determines access. Each TA determines which CAs or TAs can access its API through SMCs. System call access is of course decided by the kernel. In case of physical memory access, each Normal and Secure World process determines who can access the memory mapped inside its virtual address space. Under the second property, the Secure World is prevented from accessing Normal World memory data or code without explicit REE kernel or application approval. As a result, adversaries cannot leverage the semantic gap between worlds to bypass Normal World

or Secure World memory access restrictions. The third property ensures that physical memory access policies do not impact the use of free physical memory. All processes should have free access to physical memory that does not already contain data or code belonging to the kernel or other processes. The fourth property guarantees that policies can be introduced and updated without requiring additional application or kernel development. This property is crucial since introducing TAs likely requires updating multiple access policies. Note that this process also ensures that vulnerable TAs do not automatically provide access to security-sensitive resources. Finally, the fifth property ensures that the omission of an access specification (e.g., due to developer mistakes, kernel tricked into not forwarding policies, etc.) does not provide adversaries with access to the corresponding resource. For example, when a TEE introduces a new system call without updating the policies, the respective system call ends up being unusable instead of providing arbitrary access to all TAs. Further, in an Overseer-protected system the vast majority of REE software does not require interaction with TAs or the TZOS. The fifth property ensures that access to their physical memory is denied by default, without requiring any changes or provision of policies.

The main challenges in introducing an access control system based on the described properties include: (i) monitoring and enforcing access control over security-sensitive operations and (ii) collecting the information required to determine whether each operation should be allowed or denied.

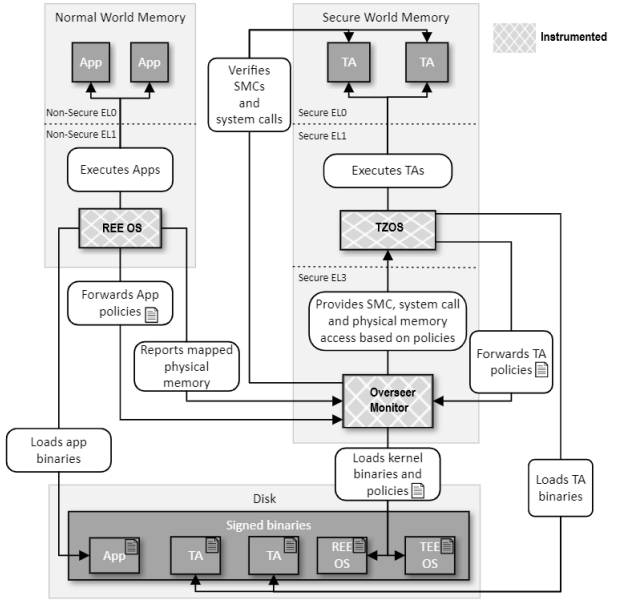


Figure 2: Overview of Overseer components

5.2 Architecture

In this section, we introduce Overseer, our proposed EL3-based access control system. Overseer introduces a unified fine-grained access control over physical memory, system call APIs, and TA communication. Based on the properties listed in Section 5.1, Overseer control is enforced in the presence of compromised TA software and leverages access control policies to bridge the semantic gap between the two environments.

Figure 2 depicts how Overseer leverages its EL3 isolation and control over Normal and Secure World resources to monitor and enforce access policies upon all executing processes. Overseer is built around five security-critical operations:(i) on-demand loading of access policies, (ii) monitoring SMCs, system calls and physical memory mapped inside running processes, (iii) enforcement of fine-grained physical memory access policies, (iv) minimizing kernel and TA access and (v) prevention of arbitrary Secure World access to Normal World memory. The five operations enable the enforcement of OS and application-defined access policies across both worlds. Each operation is detailed in the following subsections. In order to enable policy enforcement, changes have been made to REE and TEE OS to read the policies from the application binaries and forward them to Overseer along with their mapped physical memory regions. The Secure OS has also been instrumented to notify Overseer of any syscall invocations from TAs.

5.3 Defining access policies

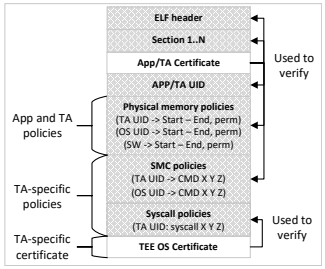


Figure 3: Providing access policies using signed binaries under Overseer

Under Overseer, each software running on the device (applications, TAs and OSes) provides a set of signed access policies that define who has access to their memory. TAs and TZOS are also required to define policies that describe who can communicate with them and access their functionality through either APIs or SMCs.

Figure 3 depicts the format of each policy and how it is provided by an application or TA through the use of a signed ELF binary. Each signed binary provides a unique identifier (UID), that is used both to track its corresponding policies inside EL3 and to identify corresponding entries in other loaded policies. The UIDs are based on Universally Unique Identifiers [17] that support backward compatibility across different versions and implementations. All policies are enforced based on UIDs extracted from signed binaries as illustrated. Each set of memory, SMC and system call policies are defined in terms of who is provided access (e.g. APP UID) and what it can access. The policies detail whether a process can issue SMCs, system calls, and map physical memory belonging to other software with permissions to read, write or execute. CA developers know precisely which TAs may access their memory. Note, only TA binary policies contain SMC and system call permissions because regular processes are not restricted from accessing Normal World system calls and cannot receive SMCs.

To ensure that policies are protected against tampering prior to being loaded and stored inside EL3, all binary provided information has to be signed using the application/TA developer private key.

Importantly, in the case of system call specific policies (that control access into the Secure World OS), the TZOS vendor, who is the provider of the resources, actually defines and signs the syscall policies included inside TA binaries, and its certificate is included by the TA developer inside the binary.

Importantly, to enable the Normal World OS to provide general access into specific areas of Normal World physical memory, the corresponding access policies can also be defined for general Secure World access by specifying a special "SW" identifier instead of a specific UID. This identifier is required to support features such as setting up shared memory regions between applications and TAs inside Normal World memory. Similarly, for general Normal World access to a TA's SMCs, use the "NW" access identifier.

5.4 Loading access policies

It is impossible to know prior to starting a system how access should be restricted in terms of allowed SMCs, system calls and physical memory accesses. Each time the system runs, there is a constant change in executing processes, each requiring varying degrees of system access. Under Overseer, the OS is responsible for including the initial set of memory access policies inside its signed binary image. At boot time these policies are sent to Overseer. For example, the Normal World OS is expected to define any physical memory that needs to be made accessible to the Secure World. Additional policies are loaded at runtime as needed from signed application and TA binaries as depicted in Figure 3. These policies are also removed once the corresponding application stops executing. Note, loading policies only when necessary is crucial for minimizing the impact of policy verification on system performance.

As mentioned in Section 5.3, Overseer developers define their access policies in terms of syscall numbers, SMC commands, address space regions, and persistent unique identifiers (UIDs). The former two are straightforward to enforce, as they are not dependent on the system state. syscall numbers and SMC commands are fixed and known before the system starts running. The latter two require additional processing before they can be enforced.

The Normal and Secure World OSes validate the signatures on the policies contained within the binaries by utilizing the corresponding application certificates, which are also embedded within the application binary. Signatures on the syscall policies is verified using the Secure OS certificate. Policies are loaded from within the binary upon application startup and cannot be altered during runtime. The memory regions specified inside signed binaries are automatically translated by the instrumented OS binary loader into the corresponding physical memory address ranges prior to being forwarded to Overseer running inside EL3. This enables fast verification of policies by directly comparing physical memory ranges inside EL3, without requiring additional information from the OSes, page walks, or context switches. Further, the process ID of an executing application is associated with its corresponding UID loaded inside EL3 from the signed binary.

5.5 Monitoring mapped physical memory

To introduce the necessary monitoring of Secure World memory mapping, Overseer requires both the Normal World OS and TZOS to report all newly mapped physical memory through SMCs. Reporting of mapped physical memory is enforced due to Overseer's

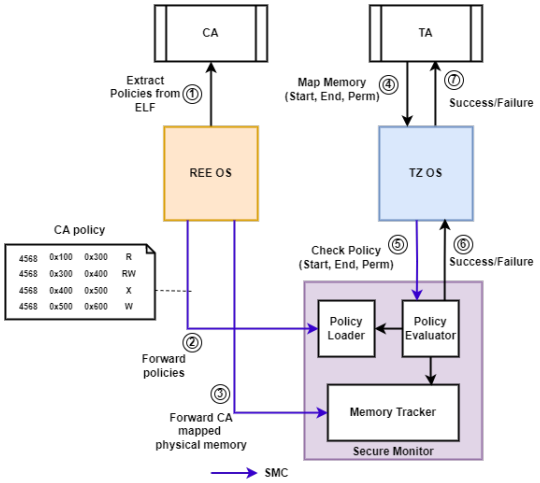


Figure 4: Overseer policy evaluation flow. (1) To run a CA, REE OS loads its signed ELF binary and extracts the policies. In (2) and (3), the policies and the mapped physical memory are notified to Secure Monitor via SMC. When a TA requests the TZOS to map a physical memory (4), TZOS issues an SMC to SM for validating the request (5). Based on the requested memory, SM identifies the owner and evaluates its policies. The result (6) and (7) is notified back to TZOS and TA.

control over page table updates. An Overseer memory tracking component ("Memory Tracker" in Figure 4) running at EL3 records all the physical memory (at page size granularity) that is mapped across both worlds. For each mapping, the corresponding process, memory range, and access permissions are extracted, verified, and stored inside EL3 until the respective memory is unmapped.

Tracking the mapped physical memory of all running processes in both Normal and Secure World is crucial for enforcing memory access policies. Without it, adversaries could try to preemptively map unused physical memory that would later be used unknowingly inside the victim's process address space. Such an attack would bypass the introduced policies because they do not prevent malicious processes from mapping unused physical memory. To prevent these attacks, Overseer always verifies the recorded memory mappings on each mapping request to ensure there is no overlap between the mapping of physical memory between different processes without explicit permissions through loaded memory access policies. The shared memory between the Normal World OS and TAs is a chunk of memory owned by the Normal World and should be explicitly allowed to be mapped by the Secure World through policies (using the SW identifier defined in Section 5.3).

5.6 Enforcing memory access policies

By controlling Secure World MMU and page table management, Overseer governs mapping of physical memory within Secure World, both at Secure EL0 and Secure EL1 execution levels. This enables Overseer to enforce the loaded policies and only allow TAs and the TZOS to access Normal World memory when explicitly allowed by either Normal World processes or the REE OS. Additionally, TAs access to Secure World physical memory that is already mapped is also provided based on TA and TZOS policies.

The verification of loaded policies is triggered on every update to the TTBR0 or TTBR1 page tables from the TZOS. Typically, under ARMv8 the TTBR1 register holds the kernel space page tables while TTBR0 is used for managing the user space page tables. Thus, in this work, we assume the separation between user and kernel space is done through the use of TTBR registers. For a different distribution between user and kernel space, Overseer would simply have to receive the respective information from the kernel or directly determine their separation based on the page table entry access permissions.

For Secure World TTBR1 page table updates, Overseer determines whether the memory ranges being mapped, correspond to Secure World or Normal World physical memory. The former is always allowed as Overseer does not restrict TZOS kernel access into Secure World. For the latter, the mapping is only allowed provided there is at least one policy that allows the TZOS kernel to map the entire respective memory ranges with their underlying permissions. Based on the principle of least privilege, TZOS can only access Normal World physical memory when policies are loaded either from a Normal World process or the REE OS. These policies either (i) provide general Secure World access into the respective memory or (ii) provide access to specific TZOSes based on TZOS UIDs that would be included in the signed policies.

In case of Secure World user space, the TTBR0 page tables are reconstructed upon every process context switch, and are also updated during the regular execution of processes. In both the cases, we modified the TZOS code that updates the page tables to also trigger an SMC to Overseer, sending it the table entry's starting physical address, the size and the requested access permissions. Upon receiving the SMC, Overseer first searches for application policies (for the matching memory range) that provide access to all Secure World processes, regardless of UIDs (policies with the "SW" identifier). Then, TA-process-specific access is determined by comparing the requested mapping against the memory ranges that contain UID-specific policies. UID of the originator is computed from the Context ID register. If a match is found, the permissions in the request are verified against the permission from the policy. If not, the request is finally verified to ensure that the underlying memory range does not overlap with any other mapped memory. This process guarantees that TAs can only access physical memory mapped inside other Normal and Secure World processes based on provided policies. Note, TAs retain the ability to freely map any unused Secure World physical memory under Overseer without the risk of policy violation due to the verification of all mapped memory overlap. Refer to Figure 4 for policy evaluation flow.

Importantly, the policy loading process detailed in Section 5.4 verifies that Normal World and TAs can only provide TA or TZOS access into their own address space. When a process starts executing, the instrumented OS binary loaders validate that the physical memory ranges within the policies forwarded to Overseer always correspond to already mapped physical memory. It is impossible under Overseer for a process to grant permissions to physical memory that itself does not have access to. At the kernel level, REE OS-defined policies can cover the entire Normal World memory, while TZOS policies are restricted to physical memory marked as Secure.

Dynamically allocated memory protection. Access to memory not defined within any of the application's policies is denied by default in Overseer. This includes dynamically allocated memory, where the physical address is known only at run-time. To facilitate TAs to access dynamic memory of CAs and TAs, we introduce a new custom memory allocator within the Normal and Secure OS. The allocator accepts one additional argument that specifies the access policy on the newly allocated memory, in terms of who (TA UID) can access it and with what permissions. CAs and TAs should use this new allocator API instead of the default system calls (e.g., malloc or calloc). The custom allocator performs two tasks. First, it calls the default memory allocator, ensuring that the new memory mappings are reported to the "Memory Tracker". Second, it triggers an SMC to Overseer, forwarding the policies associated with the newly allocated memory. Overseer then incorporates these policies into its existing set for future evaluations. A customized free call reverses the operation of the allocator by sending an SMC to Overseer to remove the dynamic policies and its associated memory mappings.

Placement and deployment. There are two placement choices for the Overseer policy enforcement framework: it can be implemented either as a component of the TZOS or as a dedicated system operating at the Secure Monitor privilege level. Section 7.3 details the advantages of each choice. To facilitate deployment, Appendix B provides steps on how to build Overseer's memory policies for a Normal World application.

5.7 Fine-grained SMC and system call access

Overseer control over memory mapping only limits the direct access of TAs to physical memory. However, as depicted in Figure 1, adversaries might still obtain access to the respective memory by elevating their privileges inside Secure World or tricking TAs into performing the malicious memory accesses on their behalf, leading to indirect physical memory access. To mitigate the impact of such attacks and minimize the overall attack surface exposed to compromised TAs inside secure World, Overseer also controls access to SMCs and TZOS system calls based on access policies.

Secure Monitor Calls. Each SMC issued by either a TA or the Normal World kernel first arrives inside the EL3 Secure Monitor due to the TrustZone interrupt forwarding hardware. This enables Overseer to automatically intercept all SMCs and either forward or drop them based on their origin (UID), SMC Function ID and loaded policies. The origin is calculated by extracting the issuing process context id from the "Context ID" register and then finding the corresponding UID inside the Overseer managed tables. Similarly, the SMC function ID is directly read from the R0 (for ARM32) or W0 (for ARM64) registers. The corresponding policies are automatically loaded by the instrumented TZOS when the intended receiver starts executing. Note, when SMCs arrive at the EL3 Secure Monitor, no context switch is yet performed. Thus, the Context ID register always contains the context id of the issuing process when Overseer intercepts the SMC handling.

By specifying access policies, TAs have control over who can access their exposed SMC-based APIs. For example, a KeyMaster TA can leverage policies to only permit the KeyStore Normal World

process to import/export encryption keys and prevent other malicious or compromised processes from accessing its API. Moreover, the SMC Function ID can be used to limit what functionality within the TA is made accessible. For instance, a TA used for attestation can allow any Normal World process to issue SMCs for requesting signing of data, while only allowing one or more trusted ones to maintain its remote attestation certificates.

System calls. To control access to TZOS-provided functionality, an intercept placed at the beginning of the TZOS Supervisor Call (SVC) exception handler forwards each TA issued system call, to Overseer for inspection. Policy evaluation verifies syscall permitted access based on their number and not its arguments or state. Overseer does not mitigate vulnerabilities within the syscall themselves. Similar to SMC handling, the originating UID is computed from the Context ID register while the system call ID is placed by the instrumented kernel inside the X0/W0 register. System call-specific policies are forwarded by the TZOS from TA binaries. Note, these policies must be signed by the TZOS vendor to ensure the TZOS retains control over its own system calls.

It is important to mention that the TZOS itself could enforce fine-grained system call access without introducing the additional context switches to EL3, as shown by the introduction of TA drivers in the case of Kinibi and Teegris. However, as detailed in Section 7.3, the existing TZOS system call restriction is not based on the least privilege access principle and does not prevent indirect system call access. In contrast, under Overseer the TEE vendor must explicitly provide TAs with access to system calls as required. Additionally, the management of policies through TA binaries provides a much more scalable solution to the system call access management, as updating policies does not require any kernel code changes.

6 OVERSEER IMPLEMENTATION AND EVALUATION

Implementation. Overseer integration only requires the introduction of 580 LOC into the Secure Monitor code running at EL3. That includes 240 LOC for handling SMCs that load and manage policies, 140 LOC for recording mapped physical memory and 200 LOC for policy evaluation. The main integration challenge represents introducing the code instruments detailed in Section 5.5 inside OP-TEE and Linux kernel code.

120 LOC have been altered inside the OP-TEE kernel to trigger SMC requests on page table updates. Additionally, 180 LOC have been introduced to verify the signatures and forward the policies located within TA binaries for access verification. Note, current OP-TEE logic lacks support for complex inter-TA communication, which under Overseer should also be constrained under fine-grained access control policies.

In the case of the Linux kernel loaded inside Normal World, 150 LOC had to be added to forward the page table entry changes to Overseer and 160 LOC for verifying and forwarding the policies located within the kernel and application binaries. For SMCs handling, no change is required inside Normal World as they always first arrive inside their EL3's corresponding handler.

Overseer Integrator. To streamline the integration of Overseer policies into binaries during development, we introduce an *Integrator* tool that automatically produces signed binaries from annotated source code. Given the annotated code that defines the policy and the developer key, the tool generates a signed binary that specifies the policy's protected virtual memory regions and their access permissions for designated TAs. See Appendix B.

Evaluation. Evaluating the impact of introducing Overseer directly into the commercial TEEs investigated in Section 3.2 is not possible. Access to the source code of Kinibi, Teegris, and QSEE is tightly guarded. Devices running these TEEs do not provide access to their Secure Monitor due to obvious security risks. Secure Monitor code for Teegris devices is even deployed encrypted on commercial devices. Instead, the effort to introduce Overseer is evaluated by integrating it into the Trusted Firmware-A (TF-A) open-source reference implementation.

Our Overseer prototype is built into the first stage bootloader of a Nitrogen8M development board, containing Arm Cortex-A53 processor and 2GB LPDDR4 RAM. The bootloader loads and verifies inside Secure World a modified OP-TEE[18] kernel, as detailed in Section 5.5 and inside a Normal World Linux 5.1 binary containing the instrumented code described in Section 5.5.

To evaluate Overseer's impact on the system performance, a series of multilevel benchmarks have been performed. First, the impact of forcing Secure World page table changes to undergo verification inside EL3 is analyzed in Section 6.1. Then, the introduced overhead on Secure World and Normal World kernel operations and applications is presented in Section 6.2. Overall, the evaluation results indicate Overseer has minimal impact on both OS and application performance while introducing the crucial system-wide fine-grained access control.

6.1 TEE memory management overhead

The Secure World memory handling is the most impacted under Overseer. Specifically, the verification of physical memory policies and introduced context switches mainly affect the TA and TZOS memory mapping operations. To calculate the degree of Overseer-introduced memory mapping overhead, we set up evaluation code inside the OP-TEE kernel. This code uses the Counter-timer Physical Count register (CNTPCT) to time each request to map one or more 4KB physical memory pages. The average execution time after running each request a thousand times is presented in Figure 5.

The overhead over a vanilla OP-TEE setup is calculated for three Overseer configurations, where either 10, 50, or 100 physical memory policies are pre-loaded. Figure 5 illustrates the additional time required under Overseer to map between 1 and 1024 physical contiguous memory pages under each configuration. As expected, the additional policy verification and context switches can slow down memory mapping significantly when only a couple of memory pages are processed at a time (up to 25% when 100 policies require verification). In contrast, the overhead is negligible in the case of large chunks of contiguous memory pages, where the bulk of computation is performed inside the OP-TEE kernel.

Importantly, the impact Overseer has upon memory mapping is heavily dependent on the number of "active" policies. This impact is depicted in Figure 5 where the overhead of mapping a single memory page jumps from 6.4% to 24.8% when an additional 90 memory policies have to be verified. However, with the exception

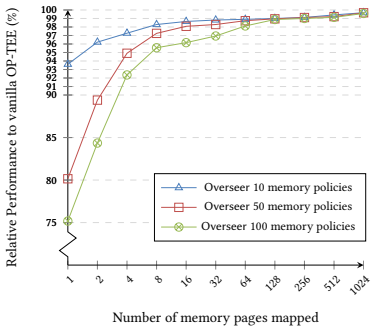


Figure 5: Overseer memory mapping overhead. As the number of policies grows, the validation time and subsequently the mapping time also increase. Additionally, larger contiguous memory chunks incur less mapping overhead.

of OS-provided policies, each other policy is only loaded when the process corresponding to its application or TA starts executing and is removed once the respective process exits. Thus, it is unlikely that hundreds or even dozens of such policies to be active at the same time in practice.

Overseer's Memory Tracker (Section 5.5) uses 32 bytes for storing each mapped physical memory range. That includes the Context ID, starting physical address, size and attributes. Similarly, an Overseer policy would also require 32 bytes. On a typical system, if we assume 30 active applications with 10 policies each and around 100 memory mappings per process, the memory tracker would consume around 96KB static memory within the Secure Monitor. Storing policies would take an additional 9.6KB. Note, CAs are not required to define an Overseer policy. They only need to define access policies for regions they wish to share with the TA, as access to unspecified regions is denied by default.

6.2 Benchmarks

Secure OS benchmarks. Overseer enforcement of access policies mainly impacts the operations inside Secure World. We use IMX OPTEE xtest [19] benchmarks to capture the overhead of policy enforcement. The benchmarks were run with 300 memory access policies and around 4000 memory mappings in the memory tracker. Figure 6 (b) and (e) summarizes the benchmark results compared against unmodified OP-TEE and EL3 monitor. Overseer has minimal impact (< 12%) on TA and secure kernel operations. We observed that the "secure storage" write benchmarks resulted in 320 additional SMC calls from the Secure OS to the Secure Monitor for policy evaluation. Note that xtest spends most of its running time invoking Secure World calls, while typical client applications, such as bio-metric authentication, secure messaging, and mobile payment apps, spend less than 10% of their total runtime in the Secure World [20]. As a result, the overall impact of Overseer on these apps is less than 2%.

REE OS benchmarks. Figure 6 (a) and (d) presents the latency overhead incurred on key OS operations, such as system calls, forking processes, handling page fault/signals, and context switches. These results are collected using standard LMBench 3.0 [21] micro-benchmarks that contrast the operation latency under a vanilla Linux setup (with OP-TEE running inside Secure World and an unmodified EL3 monitor) and one that also has Overseer running inside EL3, set up with 100 policies related to physical memory access. In the case of context switching, the latency is measured for the switching between 2 processes that perform no additional computation.

The results indicate that Overseer minimally impacts most Normal World kernel operations, incurring mostly an under 10% overhead. The most impacted operations correspond to forking a process, opening and closing a file, and context switching. The underlying cause of this overhead is likely partly due to the code introduced for policy loading inside the kernel and indirect overhead from operations inside EL3 and Secure World. Overall, Overseer introduced overheads are likely to only delay Normal World kernel operations by a few microseconds, a more than acceptable cost for introducing the required protection against malicious Secure World access.

Application benchmarks. To understand the overall impact of Overseer on application performance, we leveraged standard Phoronix test suites to evaluate how different types of Normal World applications are affected by Overseer-introduced overhead. Figure 6 (c) and (f) contrasts the overhead incurred when Overseer is set up to run inside EL3 100 policies related to physical memory access against the standard Linux setup with an unmodified EL3 monitor. Overseer can protect applications such as SQL databases, OpenSSL software, and Java programs against Secure World access with a minimal average impact (under 1%) on their performance.

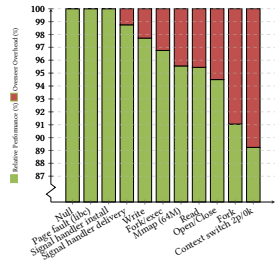
7 SECURITY ANALYSIS

7.1 Security Properties

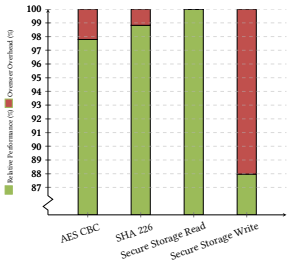
Under Overseer, policies that control access to physical memory, SMCs, and system calls are based on the following two security guarantees: (i) the policy enforcement mechanism inside EL3 is always enforced (Security Property 1) and (ii) introduced policies do not expand the attack surface (Security Property 2). However, these policies only restrict possible attack vectors inside Normal and Secure World and do not eliminate any potential vulnerabilities inside application, TA, or OS code.

Security Property 1. *Secure World software cannot bypass Overseer's integrity and its control over policy enforcement.*

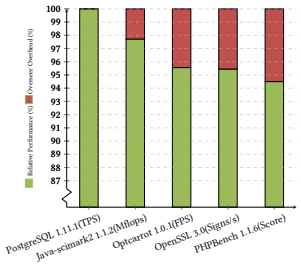
Discussion: Overseer's policy enforcement mechanism is rooted in its control over MMU and page table management in the case of memory access policies, and intercepts inserted into the SMC and system call handlers. Overseer ensures that only verified REE OS, TZOS and EL3 code is loaded. This verification is performed through Secure Boot, a process that verifies that the signed binary images are not tampered with. In addition, techniques such as TZ-RKP [15] and Sprobes [16] can further ensure the protection of OS code from untrusted apps. TZ-RKP for example, modifies the kernel to remove certain system control instructions from executable memory thereby completely preventing unauthorized kernel code modifications, code injection, or execution of user space code in privileged mode. Moreover, Overseer denies access to any memory beyond the defined policies. The trusted EL3, TZOS and REE OS ensure that it is impossible for an adversary to maliciously bypass Overseer's policy enforcement.



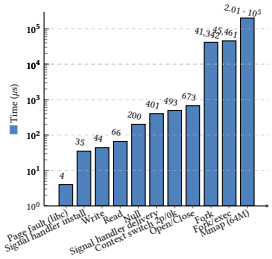
(a) LMBench Micro Benchmark for Normal World operations. Relative performance to native execution.



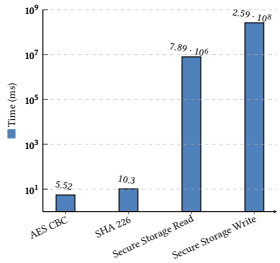
(b) OP-TEE benchmarks using xtest. Relative performance to native execution.



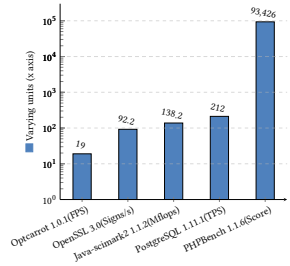
(c) Normal World application benchmarks. Relative performance to native execution.



(d) LMBench Micro Benchmark for Normal World operations. Absolute values with Overseer.



(e) OP-TEE benchmarks using xtest. Absolute values with Overseer.



(f) Normal World application benchmarks. Absolute values with Overseer

Figure 6: Overseer Performance Benchmarks of Normal and Secure Worlds. Figures (a) and (d) show the LMBench micro-benchmark results for 100 Normal World operations. Overseer incurs < 10% overhead for most operations. Figures (b) and (e) show the OP-TEE benchmarks using xtest. SHA226 and AES_CBC results show the time taken to compute SHA226 hash and perfrom AES encryption/decryption in CBC mode on a 4096 and 1024 byte buffer respectively. Secure Storage tests used a 16KB data buffer. All tests were run with 300 memory access policies. Overseer introduces < 3% overhead for crypto tests and < 12% for secure storage. Figures (c) and (f) show the Normal World application benchmarks. Overseer incurs < 6% overhead for most operations.

Security Property 2. *Adversaries cannot leverage maliciously designed access policies to bypass existing orthogonal access control mechanisms.*

Discussion: Under Overseer, each application, TA, and OS can only define access policies in terms of resources (memory, SMC, system calls) that it already has control over. For memory access, the physical memory regions translated by the instrumented OS binary loaders from policies always have to be already mapped inside the process address space. Similarly, in the case of SMCs and system calls, the API owner (either TA or TZOS) provides the corresponding policies. Thus, it is impossible for adversaries to introduce policies that grant them access to previously inaccessible memory, SMCs, or system calls.

7.2 CVE Mitigation

We performed qualitative analysis on 77 existing TA-related CVEs from [22] and bug reports from the CVE database [23] and in 53 of those, Overseer can significantly reduce the attack surface. For example, TA vulnerability CVE-2022-22271 [24] allows an attacker to copy data from any arbitrary memory. With Overseer, the copy is only allowed if the entity owning that memory explicitly grants read permission to the TA, through policies. Otherwise, access is denied. Likewise, Overseer helps in buffer overflow attacks (e.g., CVE-2019-20607 [25]) which could result in arbitrary code execution, allowing unauthorized access to read and write arbitrary memory locations. Note, Overseer does not prevent the buffer overflow attack, but instead helps in mitigating the attack surface of the exploit. The rest of 24 CVEs are not in scope of the threat model described in Section 4. For example, CVE-2020-12752 [26] is about brute-force attack against a TA to determine user credentials. This cannot be mitigated with Overseer. A detailed breakdown of the vulnerabilities is presented in Appendix A. A case study of an attack that exploits CVE-2021-1961 [27] is presented in Appendix C.

7.3 Additional Considerations

Policy Enforcement Point (PEP), TZOS Vs Secure Monitor: There are a couple of design choices as far as the placement of Overseer is concerned. 1) Integrate Overseer into the Secure Monitor operating at EL3. 2) Run it as part of TZOS at EL1. We followed the former approach in this paper. Instead, if Overseer is part of TZOS and policy enforcement happened within TZOS, it would avoid an additional SMC call to the Secure Monitor on every update to the TZOS page tables (Section 5.6). Similarly, SMC requests that are triggered to validate syscalls (Section 5.7) can also be avoided. But the downside of the approach is that the Normal World OS would then have to forward all CA/OS policies and memory mappings to the TZOS via the Secure Monitor. This would result in the costly *world switch* and an additional call to the TZOS from Secure Monitor on every update to the page tables within the Normal World.

Secondly, all SMC requests from Normal World initially reach the Secure Monitor, which then performs the *world switch* and interrupts the TZOS to invoke the TA process. Once the *world switch* happens, all the Normal World contextual data (e.g., PIDs, registers etc) is no longer valid and is instead overwritten with Secure World parameters. In the current design, Overseer evaluates the SMC policies before the *world switch* happens and so the originating process details are still intact. If policy enforcement happens within TZOS after the *world switch*, all the contextual data of the originating process has to be additionally saved and forwarded to the TZOS for policy evaluation.

Although both the design choices are feasible, we intend to keep the design simple by following the existing framework where the Secure Monitor centrally controls all communication between both the worlds and therefore would be the best choice for policy enforcement. Performance measurements (Section 6) also indicate that Overseer adds a minimal overhead to Secure World operations.

Indirect memory, SMC, and system call access. The introduced policies only prevent direct access and do not protect against indirect attacks where adversaries trick or compromise applications with access to target memory, SMC, or system call. For example, an OS that is granted access to a target physical memory through policies could be tricked by a malicious TA into writing malicious data into the respective memory due to a lack of pointer sanitization. However, the impact of such attacks is severely limited as policies only provide access when required. In the previous example, the OS is limited in regards to what physical memory it can map inside Normal World and TA access to OS-provided system calls is minimized.

Policy enforcement inside Normal World. The current Overseer design only tackles the problem of restricting Secure World access through proposed policies. Of course, the control can also be expanded inside Normal World by instrumenting the REE OS kernel code and introducing the required changes for intercepting system calls and page table writes. However, restricting access inside Normal World requires much effort to integrate into existing systems and imposes a significantly larger overhead upon system performance. For example, restricting access to system calls based on the principle of least privilege would break the functionality of applications that do not provide the required access policies inside their signed binaries (thousands of existing applications). Thus, for practicality, Overseer focuses only on restricting Secure World access, where security importance is paramount.

Sharing memory in Normal World. Normal World mechanisms such as shared memory, shared libraries, and fork or exec allow applications to share code and data pages by mapping the same physical memory into multiple address spaces. As described in Section 5.5, Overseer tracks Normal World physical memory mappings via the instrumented OS but does not enforce policies on them. Therefore, shared Normal World mappings do not affect application behavior, even when physical addresses overlap across processes. However, Overseer raises an error if a Normal World mapping overlaps with memory already mapped to a TA, indicating that the TA preemptively mapped an unused region later assigned to a CA. In contrast, when mappings originate from the Secure World, Overseer enforces the owning application's policies and rejects any unauthorized memory overlap.

Overseer versus existing access control. As presented in Section 3.2, there are some TAs that restrict access by hardcoding access rules. Similarly, some TZOSes also categorize TAs (e.g., TA drivers) and restrict access to system calls based on each category. However, these techniques are hard to maintain (require code modification) and do not provide access based on the principle of least privilege, and typically lead to indirect access. For example, a single TA driver under Kinibi and Teegris provides 15 other TAs with the same permissions to read and write physical memory. In contrast, Overseer eliminates the need for TA drivers and restricts access using policies that can be easily maintained. In the former example, policies would restrict the 15 TAs to only map the physical memory they require. Importantly, under Overseer, all policies are provided by the resource owner and enforced from within EL3, while existing hardcoded access rules are vulnerable to Secure World code compromise.

8 RELATED WORK

Several works identified vulnerabilities in TAs that result in leaking secrets and altering TA memory [28, 29, 30, 31, 12, 22]. Efforts have been made to analyze the TEE isolation provided by TrustZone and identify any weaknesses within, both manually and using fuzzing. The different TrutZone TZOS variants are presented and compared in [32]. A case study of the KNOX containers is also presented in [33]. These studies analyze the attack vectors that can be used to escalate privileges into the Secure World.

Boomerang [5] and HPE [6] attack vectors are manifestations of the TA confused deputy problem. Boomerang and HPE attacks use the semantic gap present in the memory sharing between the two worlds to trick the Secure World into overwriting arbitrary Normal World memory. Specifically, Boomerang only targets vulnerabilities in SMC requests made by the CA that involve memory it does not own. In contrast, in this work, we present vulnerabilities arising from the ability of TAs to arbitrarily access and alter the contents of Normal World physical memory, both directly and indirectly, and not just those stemming from SMC requests.

Sanctuary [34] and SHELTER [35] enable running security sensitive apps inside strongly isolated compartments within the Normal World. However each executing Sanctuary/SHELTER App requires a dedicated CPU core during its lifetime. Also, porting existing TAs to such apps involves significant development cycles. Overshadow [36], OSP [37], Terra [38], vTZ [39], pKVM [40] and PrivateZone [41] use virtualization techniques to isolate Normal World execution environments to constraint vulnerabilities. Conversely, Overseer does not rely on virtualization support and has better performance gains. More importantly, the approaches mentioned above protect Security Critical Code (SCC) (e.g., a TA) by executing it within a virtual environment, thereby shielding it from susceptible Normal World applications and OS. However, if the SCC code has bugs and is compromised, it will indirectly result in compromising the Normal World applications and OS. Overseer addresses this issue and restricts access from TAs according to the policies.

TZ-RKP [15], Sprobes [16] and AppBastion [42] provide OS code integrity by preventing kernel code injection and return-to-user attacks. This is achieved by taking over the MMU and TTBR registers through OS code instrumentation. While these protect the

Normal World OS code from untrusted apps, Overseer protects the Normal World from compromised TAs.

SGXJail [43] protects host applications from mis-behaving trusted code running inside Intel Software Guard Extensions (SGX) *enclaves*. The *enclaves* are run within a sandbox process and all communication between the host and the enclave happens over shared memory, thereby preventing any arbitrary read/write access from the enclave. But *enclaves* that need direct access to the host memory break when SGXJail is active. Conversely, Overseer permits a TA to access the CA memory directly, contingent upon adherence to the CA policies. ReZone [9] relies on additional hardware resources (for e.g., Resource Domain Controller and Cortex M4 hardware on i.MX8 SOCs) available on COTS SoCs to partition the TEE into multiple zones each having access to private memory regions. The untrusted OS and TAs run within the zone thereby preventing it from arbitrarily accessing Normal World and other zones memory. Overseer, on the other hand does not require extra hardware; instead, it utilizes software techniques to assume exclusive control over page tables, compelling the validation of all memory mappings through policies. Overseer also restricts indirect access to Normal World memory through SMCs and Syscalls.

9 CONCLUSION

The existence of an inherent inter-world semantic gap creates an opportunity for compromised code to maliciously manipulate REE and TEE memory. This is further exacerbated by unrestricted SMC and system call invocations. In this paper we investigated how vendors manage their Normal and Secure World physical memory and discovered dozens of associated vulnerabilities. To mitigate these and other vulnerabilities we then introduced Overseer, a policy enforcement mechanism for cross-world fine-grained memory access control across physical memory, SMCs and System Calls. Overseer overheads are minimal.

REFERENCES

[1] ARM. 2009. Bulding a secure system using trustzone technology. *ARM Technical White Paper*.

[2] Federico Menarini. [n. d.] Breaking tee security part 2: exploiting trusted applications (tas). https://www.riscure.com/blog/tee-security-samsung-teegris-part-2. ().

[3] David Berard. [n. d.] Kinibi tee: trusted application exploitation. https://www.synacktiv.com/en/publications/kinibi-tee-trusted-application-exploitation.html. ().

[4] laginimaineb. [n. d.] Qsee privilege escalation vulnerability and exploit. http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html. ().

[5] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: exploiting the semantic gap in trusted execution environments. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society. DOI: 10.14722/ndss.2017.23227. https://doi.org/10.14722/ndss.2017.23227.

[6] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. 2020. Horizontal privilege escalation in trusted applications. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[7] Laginimaineb. [n. d.] Qsee privilege escalation vulnerability and exploit. https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html. ().

[8] joffrey guilbon. [n. d.] Attacking the arm's trustzone. https://blog.quarkslab.com/attacking-the-arms-trustzone.html. ().

[9] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. 2022. {Rezone}: disarming {trustzone} with {tee} privilege reduction. In *31st USENIX Security Symposium (USENIX Security 22)*, 2261–2279.

[10] Jerome H. Saltzer. 1974. Protection and the control of information sharing in multics. *Commun. ACM*, 17, 7, 388–402. ISSN: 0001-0782. DOI: 10.1145/361011.361067. https://doi.org/10.1145/361011.361067.

[11] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. Trustzone explained: architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 445–451.

[12] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. 2020. PARTEMU: enabling dynamic analysis of Real-World TrustZone software using emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, (August 2020), 789–806. ISBN: 978-1-939133-17-5. https://www.usenix.org/conference/usenixsecurity20/presentation/harrison.

[13] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. Sok: (state of) the art of war: offensive techniques in binary analysis.

[14] GlobalPlatform. [n. d.] Tee client api specification v1.0. https://globalplatform.org/specslibrary/tee-client-api-specification/. ().

[15] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. 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* (CCS '14). ACM, Scottsdale, Arizona, USA, 90–102. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660350. http://doi.acm.org/10.1145/2660267.2660350.

[16] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: enforcing kernel code integrity on the trustzone architecture. *CoRR*, abs/1410.7747. arXiv: 1410.7747. http://arxiv.org/abs/1410.7747.

[17] Paul Leach, Michael Mealling, and Rich Salz. 2005. A universally unique identifier (uuid) urn namespace. (2005). https://datatracker.ietf.org/doc/html/rfc4122.

[18] NXP-IMX. [n. d.] Op-tee trusted os. https://github.com/nxp-imx/imx-optee-os. ().

[19] NXP-IMX. [n. d.] Imx op-tee test. https://github.com/nxp-imx/imx-optee-test. ().

[20] Julien Amacher and Valerio Schiavoni. 2019. On the performance of arm trustzone. In *Distributed Applications and Interoperable Systems*. José Pereira and Laura Ricci, editors. Springer International Publishing, Cham, 133–151. ISBN: 978-3-030-22496-7.

[21] Larry W McVoy, Carl Staelin, et al. 1996. Lmbench: portable tools for performance analysis. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.

[22] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. Sok: understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. *2020 IEEE Symposium on Security and Privacy (SP)*, 1416–1432.

[23] MITRE. [n. d.] Common vulnerabilities enumeration. https://cve.mitre.org/. ().

[24] MITRE. [n. d.] Cve-2022-22271. https://www.cvedetails.com/cve/CVE-2022-22271. ().

[25] MITRE. [n. d.] Cve-2019-20607. https://www.cvedetails.com/cve/CVE-2019-20607. ().

[26] MITRE. [n. d.] Cve-2020-12752. https://www.cvedetails.com/cve/CVE-2020-12752. ().

[27] MITRE. [n. d.] Cve-2021-1961. https://www.cvedetails.com/cve/CVE-2021-1961. ().

[28] Gal Beniamini. [n. d.] TrustZone Kernel Privilege Escalation. http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html. ().

[29] David Berard. [n. d.] Kinibi TEE: Trusted Application exploitation. https://www.synacktiv.com/posts/exploit/kinibi-tee-trusted-application-exploitation.html. ().

[30] Daniel Komaromy. 2008. Unbox Your Phone. https://medium.com/taszksec/unbox-your-phone-part-iii-7436ffaff7c7. (2008).

[31] Dan Rosenberg. 2014. Reflections on Trusting TrustZone. *BlackHat USA*.

[32] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: a comprehensive survey. *ACM Comput. Surv.*, 51, 6, (January 2019), 130:1–130:36. ISSN: 0360-0300. DOI: 10.1145/3291047. http://doi.acm.org/10.1145/3291047.

[33] Uri Kanonov and Avishai Wool. 2016. Secure containers in android: the samsung knox case study. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices* (SPSM '16). ACM, Vienna, Austria, 3–12. ISBN: 978-1-4503-4564-4. DOI: 10.1145/2994459.2994470. http://doi.acm.org/10.1145/2994459.2994470.

[34] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. Sanctuary: arming trustzone with user-space enclaves. *Proceedings 2019 Network and Distributed System Security Symposium*. https://api.semanticscholar.org/CorpusID:86835387.

[35] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. 2023. SHELTER: extending arm CCA with isolation in user space. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, (August 2023), 6257–6274.

ISBN: 978-1-939133-37-3. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-yiming.

[36] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not.*, 43, 3, (March 2008), 2–13. ISSN: 0362-1340. DOI: 10.1145/1353536.1346284. http://doi.acm.org/10.1145/1353536.1346284.

[37] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. 2016. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (USENIX ATC '16). USENIX Association, Denver, CO, USA, 565–578. ISBN: 9781931971300.

[38] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. 2003. Terra: a virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev.*, 37, 5, 193–206. ISSN: 0163-5980. DOI: 10.1145/1165389.945464. https://doi.org/10.1145/1165389.945464.

[39] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Vtz: virtualizing ARM trustzone. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 541–556. ISBN: 978-1-931971-40-9. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua.

[40] Jake Edge. 2020. Kvm for android. https://lwn.net/Articles/836693/. (2020).

[41] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang. 2018. PrivateZone: providing a private execution environment using ARM TrustZone. *IEEE Transactions on Dependable and Secure Computing*, 15, 5, (September 2018), 797–810. DOI: 10.1109/tdsc.2016.2622261. https://doi.org/10.1109/tdsc.2016.2622261.

[42] Darius Suciu, Radu Sion, and Michael Ferdman. 2022. Appbastion: protection from untrusted apps and oses on arm. In *Computer Security – ESORICS 2022*. Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors. Springer Nature Switzerland, Cham, 692–715. ISBN: 978-3-031-17146-8.

[43] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. 2019. SGXJail: defeating enclave malware via confinement. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, (September 2019), 353–366. ISBN: 978-1-939133-07-6. https://www.usenix.org/conference/raid2019/presentation/weiser.

[44] Tamir Zahavi-Brunner. [n. d.] Attacking the android kernel using the qualcomm trustzone. https://tamirzb.com/attacking-android-kernel-using-qualcomm-trustzone. ().

[45] Thomas M. Zeng. 2012. The android ion memory allocator. https://lwn.net/Articles/480055/. (2012).

[46] codeaurora.org. [n. d.] Qualcomm patch for cve-2021-1961. https://git.codelinaro.org/clo/le/kernel/msm-4.19/-/commit/edc0db63de9a0a7375bf05fafda3d47b558fd9ff. ().

A CVE ANALYSIS

CVE	C	CVE	C	CVE	C	CVE	C
2015-9183	TA ✓	2017-6293	TA ✓	2015-4422	TA ✓	2015-6639	TA ✓
2019-10615	TA –	2019-14009	TA ✓	2017-18310	TA –	2019-20537	TA ✓
2019-20583	TA ✓	2020-0075	TA ✓	2019-20585	TA ✓	2019-20586	TA ✓
2019-20610	TA ✓	2020-13832	TA ✓	2020-0076	TA ✓	2020-0077	TA –
2020-12752	TA –	2022-33225	TA –	2020-13835	TA –	2019-20545	TA ✓
2021-34389	TA ✓	2023-21420	TA ✓	2015-9000	TA –	2019-20587	TA ✓
2015-9162	TA –	2015-9002	TA ✓	2018-5210	TA ✓	2021-1889	TA ✓
2019-20563	TA ✓	2018-5885	TA ✓	2019-20560	TA ✓	2019-20571	TA ✓
2019-20607	TA ✓	2019-20589	TA ✓	2020-10837	TA ✓	2021-1932	TA ✓
2020-11603	TA ✓	2020-11123	TA –	2021-1890	TA –	2023-21501	TA ✓
2018-21063	TA –	2022-22271	TA ✓	2023-30733	TA ✓	2019-20581	TA ✓
2023-21500	TA ✓	2021-25468	TA ✓	2020-11604	TA –	2019-20584	TA ✓
2023-21499	TA ✓	2021-25469	TA ✓	2020-10836	TA ✓	2019-20588	TA ✓
2023-21498	TA ✓	2020-11600	TA ✓	2020-10849	TA –	2019-20590	TA –
2023-21497	TA ✓	2021-1909	TA ✓	2019-20544	TA ✓	2019-20596	TA –
2019-20602	TA –	2019-20603	TA –	2019-20562	TA ✓	2018-11938	TA –
2021-1923	TA ✓	2021-1888	TA –	2019-14078	TA ✓	2018-5918	TA ✓
2021-1930	TA –	2019-14130	TA –	2019-2329	TA –	2017-18300	TA ✓
2017-18297	TA –	2016-0825	TA ✓	2017-18280	TA –	2021-1961	TA ✓
2022-35858	TA ✓						

Table 2: Mitigation analysis of CVEs: In Scope(✓), Out of Scope (-)

B OVERSEER INTEGRATION

In the following, we first outline the procedures for building a Normal World application that communicates with a TA through SMC. The application initiates a request for data encryption by exchanging both the data and encryption key through shared memory. We then introduce the Overseer Integrator, an automation tool that modifies the application to enforce Overseer policies. This allows the TA direct access to the application's memory without relying on the shared memory. To keep it concise, we highlight only the essential steps in the process.

```
1  /* CA code */
2
3  char inbuf [1024 * 1024] = {0};
4  char outbuf[1024 * 1024] = {0};
5
6  read_data_from_file(char *filename)
7  {
8      int fd = open(filaname,..);
9      read(fd, inbuf,..);
10 }
11 main()
12 {
13     char key[16];
14     TEEC_UUID uuid = {0xABCDEF};
15
16     /* fill inbuf */
17     read_data_from_file("file.txt");
18
19     /* Initialize a context connecting to
20     the TEE */
21     res = TEEC_InitializeContext(...);
22
23     /* Open a session with the TA */
24     res = TEEC_OpenSession(...,&uuid,...);
25
26     /* Set the key. Load some dummy value */
27     memset(key, 0xa5, sizeof(key));
28     op.params[0].tmpref.buffer = key;
29     res = TEEC_InvokeCommand(..,TA_AES_CMD_SET_KEY, &op
       ,..);
30
31     /* Send for encryption */
32     op.params[0].tmpref.buffer = inbuf;
33     op.params[1].tmpref.buffer = outbuf;
34     res = TEEC_InvokeCommand(..,TA_AES_CMD_CIPHER,..);
35 }
```

```
1  /* TA code */
2
3  TA_InvokeCommandEntryPoint(...,cmd,...,params)
4  {
5      switch (cmd) {
6      case TA_AES_CMD_SET_KEY:
7          /* copy key */
8          key = params[0].memref.buffer;
9
10     case TA_AES_CMD_CIPHER:
11         in  = params[0].memref.buffer;
12         out = params[1].memref.buffer;
13         /* decrypt 'in' to 'out' */
14         encrypt_buffer(key, in, out);
15 }
```

B.1 Shared memory communication

The CA starts by first initializing the input buffer with the contents of the file (line 17) and then opens a session with the TA (UID=`0xABCDEF`) (line 24). It then sends the key (line 29) and the data buffer (line 34) to the TA for encryption. All communication with the TA happens over shared memory using SMC. The TA acquires the key (line 8 in TA code) and the data (lines 11-12), subsequently performing encryption on the data using the received key and storing the result in the output buffer (line 14). Since both input and output data buffers are stored on the shared memory, data written to the output buffer by the TA is also reflected at the CA.

Exchanging large amounts of data over the shared memory results in additional overheads in terms of managing the shared buffers across both the kernels. Additionally, the shared memory is a small region carved out of the Normal World physical memory and cannot accommodate large content. In the next subsection, we modify the CA to allow the TA direct access to the CA's memory instead of using the shared memory. An Overseer policy is created such that the TA can only read from the input buffer and only write to the output buffer.

B.2 Enabling Overseer via automatic Integrator

Enforcing Overseer policies requires defining access rules for Normal World memory, embedding these policies into the application binary, and ensuring integrity via signing. To streamline this workflow, we introduce the Overseer Integrator, a tool that generates a signed ELF binary by parsing annotated source code.

To start, the CA developer adds the macro definition `#define OVERSEER(...) {}` to the source code. Access controls are then specified by placing the `OVERSEER(TA UID, perm)` macro immediately preceding the target variable declaration. For the example in Appendix B.1, the annotation appears as follows:

```
OVERSEER(0xABCDEF, R)
char inbuf [1024 * 1024] = {0};
OVERSEER(0xABCDEF, W)
char outbuf[1024 * 1024] = {0};
```

This syntax supports multiple policies per variable. For example, using `OVERSEER(0xABCDEF, R, 0x123456, R)` before `inbuf` grants read access to both the original TA and an additional TA (UID `0x123456`).

Upon execution, the Overseer Integrator first parses the code to extract access control logic. Second, it adjusts the code to ensure that these protected variables will be pinned to specific locations in the CA's virtual memory during compilation. Specifically, Integrator replaces every `OVERSEER(TA UID, perm)` macro with `__attribute__((section(".private_data_section")))`, which specifies the section of the ELF where the variables would live. In this way, `inbuf` and `outbuf` are defined to be in the `.private_data_section` of the binary. To force the base address of the `.private_data_section` to be relocated to a specific virtual address (specified by the developer, say `0xA0000000`), Integrator compiles the code using:

```
# gcc  -Wl,--section-start=.private_data_section=0
    xA0000000 ca.c -o ca
```

To generate access control policies, Integrator determines the range of virtual memory address for each protected variable based on sizes and offsets of variables acquired from the binary. In our example, the tool calculates that the TA (with UID=`0xABCDEF`) can read 1024*1024 bytes starting at `0xA0000000` and write 1024*1024 bytes starting at `0xA0100000`. Then Integrator generates the policy as follows:

```
/* policy.c */
__attribute__((section(".policy_section")))
const char policy_data[] =
"0xABCDEF  0xA0000000-0xA00FFFFF R \n"
"0xABCDEF  0xA0100000-0xA01FFFFF W \n";
```

Integrator recompiles the source code to include this policy section, the CA UID and a placeholder for the signature into the binary. Finally, it signs relevant ELF sections and update the binary with the valid signature.

We now allow the TA to directly access the CA's memory and not rely on copying the data to and from the shared memory. Instead of sending the memory references from the shared memory, the CA will now pass the input and output buffer addresses using a new parameter type.

```
op.params[0].phytmpref.buffer = inbuf;
op.params[1].phytmpref.buffer = outbuf;
```

Note that the above are virtual addresses in the application space. The instrumented Linux kernel will convert these addresses to physical addresses before forwarding them to the Secure World over SMC. Similarly, when the Secure OS receives the SMC, it maps the physical addresses into virtual addresses specific to the TA, thereby aiding the TA's operation. The virtual addresses in the policy section are also converted by the Linux kernel to physical addresses before forwarding them to the Secure Monitor.

C CVE CASE STUDY

In the following, we describe an attack against QSEE where the attacker exploits CVE-2021-1961 [27] to arbitrarily read and write to any memory location in the Normal World. The complete details of the attack are mentioned in [44].

Qualcomm's TrustZone allows Android applications running in Normal World to communicate with TAs in the Secure World through shared memory from ION memory allocator [45]. To issue commands to the TA, the Android application utilizes the ioctl method for communicating with the qseecom kernel driver.

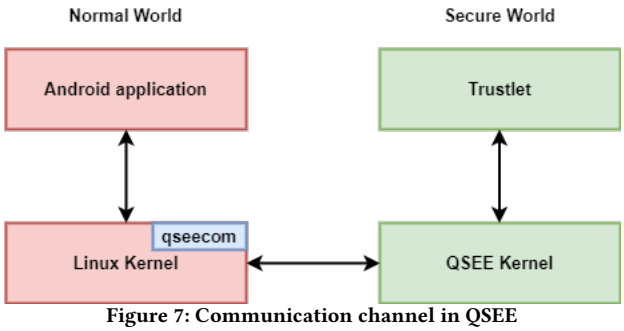


Figure 7: Communication channel in QSEE

qseecom subsequently relays the command to the QSEE kernel (Figure 7). The input and output buffers for the command are allocated from the ION heap and sent to the QSEE kernel. The TA (Trustlet) then reads from the input buffer and writes to the output buffer.

To support buffers that contain pointers to other ION heaps, qseecom writes the physical address of each ION buffer supplied by the application, into the input buffer at the specified offset. The top portion of Figure 8 shows how the Android application communicates with qseecom using ioctl. Each `qseecom_ion_fd_info` entry contains an ION handle (fd) to the user-allocated memory along with an offset into the input buffer. Once qseecom receives the ioctl, it adds the address of each ION buffer into the input buffer at the specified offset and sends the buffer along with an `sglist_info` structure to QSEE kernel. The `sglist_info` contains the same offset value as in `fd_info` but instead of the file descriptor, it contains the size of the ION buffer. The QSEE kernel goes through each address defined at the `sglist_info`'s offset field from the input buffer and gives the TA access to the memory at that address and size.

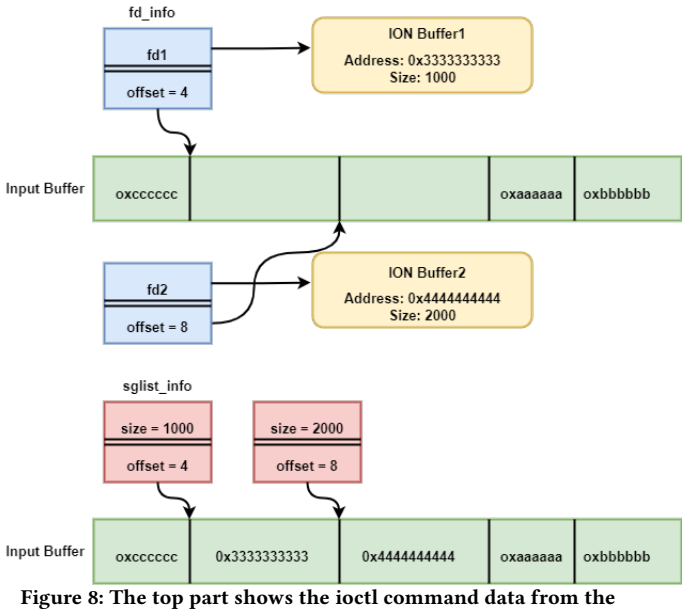


Figure 8: The top part shows the ioctl command data from the application to qseecom. fd_info points to the user allocated buffer. The bottom part shows how qseecom extracts data from fd_info to populate the input buffer and sglist_info structures, which are subsequently forwarded to QSEE kernel.

The attack. An attacker can choose the `fd_info` offsets in such a way that the addresses that qseecom puts in the input buffer, overlap. This can cause an `sglist_info` entry to point to a completely different address not belonging to any ION buffer. qseecom validates the addresses in each `fd_info` structure but fails to check if there is any overlap in the offsets when copying the addresses to the input buffer. As can be seen in Figure 9, the attacker sets the offset field in the second `fd_info` entry to 6 (instead of 8). qseecom thereby overwrites part of the address of the first ION buffer when updating the input buffer. The first `sglist_info` entry would now point to a corrupted address `0x3333334444` instead of `0x3333333333`. QSEE kernel will grant the TA access to this corrupted address. qseecom also allows user applications to use ION buffers which are non-contiguous. It does so by copying the memory address as well as its size to the input buffer. By carefully overwriting the size field in the input buffer (using the above approach), the attacker could gain read and write access to all of the Normal World kernel code and data.

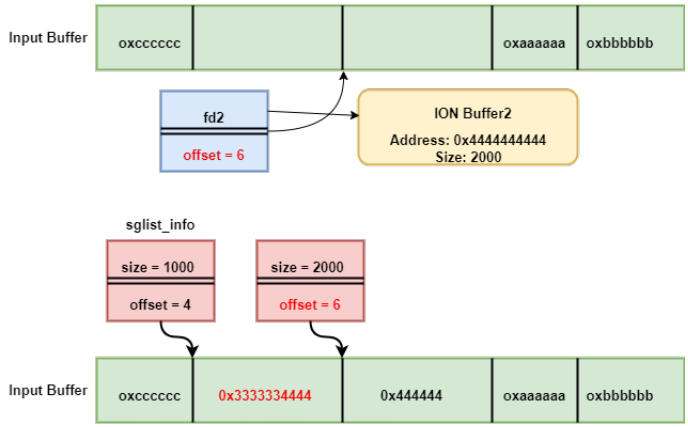


Figure 9: An attacker can modify the offset field in fd_info to overlap the addresses in the input buffer.

The fix. Qualcomm released a patch [46] for the issue that adds checks in qseecom to make sure addresses of different ION buffers don't overlap with each other.

The attack highlights a major problem where such bugs can result in the attacker having complete control over Normal World memory through the TA. Although the patch fixes the specific issue in QSEE, there could be similar vulnerabilities in other TrustZone implementations.

The effective defence with Overseer. The attack can be prevented with Overseer. Every access that the TA makes, is verified against the memory access policies. So in the attack scenario, when the TA tries to access the address `0x3333334444`, an SMC is triggered to Secure Monitor to check if the TA has the required permissions to the memory. Since the access policy would only allow access to `0x3333333333` and not to `0x3333334444` (as it does not own this address), any attempt to access the latter would be denied. Overseer relies on a whitelisting approach where the TA is granted memory access only when it is explicitly allowed by the policies, as opposed to blacklisting all possible attack vectors. Thereby, even if the Overseer developer does not explicitly define policies for the Normal World kernel memory, attempts to access any part of the kernel by the TA are denied. Overseer is also agnostic to the underlying TrustZone implementation and its design principles can be seamlessly integrated across all the implementations.