Horizontal Privilege Escalation in Trusted Applications

Darius Suciu, *Stony Brook University;* Stephen McLaughlin and Laurent Simon, *Samsung Research America;* Radu Sion, *Stony Brook University*

https://www.usenix.org/conference/usenixsecurity20/presentation/suciu

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Horizontal Privilege Escalation in Trusted Applications

Darius Suciu
Stony Brook University
dsuciu@cs.stonybrook.edu

Stephen McLaughlin
Samsung Research America
s.mclaughlin@samsung.com

Laurent Simon
Samsung Research America
cam.lmrs2@gmail.com

Radu Sion
Stony Brook University
sion@cs.stonybrook.edu

Abstract

Trusted Execution Environments (TEEs) use hardware-based isolation to guard sensitive data from conventional monolithic OSes. While such isolation strengthens security guarantees, it also introduces a semantic gap between the TEE on the one side and the conventional OS and applications on the other. In this work, we studied the impact of this semantic gap on the handling of sensitive data by Trusted Applications (TAs) running in popular TEEs. We found that the combination of two properties, (*i*) multi-tenancy and (*ii*) statefulness in TAs leads to vulnerabilities of Horizontal Privilege Escalation (HPE). These vulnerabilities leaked sensitive session data or provided cryptographic oracles without requiring code execution vulnerabilities in TEE logic. We identified 19 HPE vulnerabilities present across 95 TAs running on three major ARM TrustZone-based trusted OSes. Our results showed that HPE attacks can be used to decrypt DRM protected content, to forge attestations, and to obtain cryptographic keys under all three evaluated OSes. Here, we present HOOPER an automatic symbolic execution based scanner for HPE vulnerabilities, in order to aid manual analysis and to dramatically reduce overall time. In particular, in the Teegris Trusted OS HOOPER is able to identify 19 out of 24 HPE-based attack flows in 24-hours contrasted with our original manual analysis time of approximately four weeks.

1 Introduction

Traditional OS-based protection mechanisms are routinely bypassed due to vulnerabilities in their monolithic code bases. As a response to this limitation, hardware-isolated Trusted Execution Environments (TEEs) have gained widespread use, particularly in mobile devices. TEEs provide hardware-isolated memory and compute resources with Trusted Applications (TAs) that handle highly sensitive operations on behalf of applications running on the monolithic OS.

The most widely used TEE in mobile devices is ARM TrustZone [1]. TrustZone provides a higher privilege level in the form of the *Secure World*. The Secure World runs a Trusted OS (TZOS) and TAs in hardware isolated memory, CPU, and I/O. TAs handle requests on behalf of Client Applications (CAs), which run on the traditional OS in the *Normal World*. Sensitive data such as private keys, biometric data, and device integrity measurements should never leave the Secure World.

The hardware isolation between the two worlds (Secure and Normal) enables each to run independently from each other. However, it also introduces a *semantic gap* between them. In this case neither world has sufficient information about the semantics (i.e., data structure layout and locations) of the other to accurately identify or authenticate principals in the other world. A previous result of this semantic gap was the Boomerang [18] attack, which leverages memory safety errors in TAs to exploit the Normal World OS on behalf of malicious Normal World applications. Here, we find that cross-CA attacks are still possible without relying on TA memory exploits.

In this paper, we present our work on Horizontal Privilege Escalation (HPE) attacks [20] against TAs. These attacks result in unauthorized cross-principal data access between Normal World services. HPE does not require a compromised service to escalate its privileges to access data belonging to other principals. Instead, it leverages persistent state mismanagement by TAs acting on behalf of other victim services. In this light, HPE is a type of confused deputy [11] attack where the attacker accesses victim data without directly escalating privilege.

We manually examined 95 TA binaries from mobile devices running three popular TEEs: Trustonic's Kinibi, Qualcomm's QSEE, and Samsung's Teegris. We identified 19 unique HPE vulnerabilities (52 when counting duplicate from porting between TZOSes). These 19 vulnerabilities led to 27 unique attacks against different TA APIs (78 when counting duplicates) in our study. Here, We classify the discovered issues using known Common Weakness Enumeration [19] (CWEs) and provide case studies to show the impact of each.

To aid in the analysis of HPE attacks in TAs, we implemented HOOPER, a tool based on the angr [28] symbolic execution framework to search for HPE bugs in TAs for the Teegris TZOS. HOOPER uses memory and storage API inspection along with state matching to track TA handling of CA data across invocations. These locations represent opportunities for HPE attacks. In Teegris HOOPER found 19 out of 24 HPE-based attacks in 24-hours contrasted with our original manual analysis time of approximately four weeks.

The contributions of this work include:

1. a first study of HPE attacks in TAs. Attacks identified are caused both by TA session management design flaws, and the TEEs - Normal World semantic gap.
2. an extensive analysis and detection of HPE vulnerabilities in 95 TAs from three widely used TEEs. Results are broken down according to known CWEs [19] and case studies showing the real-world impact of each.
3. an evaluation of HOOPER, an automatic scanner for HPE vulnerabilities in TAs. In the Teegris OS HOOPER evaluating 31 TAs can detect in less than 24 hours 80% of the HPE vulnerabilities previously identified through manual analysis in four weeks.

2 Background

Trusted Execution Environments (TEEs) are secure partitions of hardware providing isolated CPU, memory, and I/O access for sensitive data and trusted code. The traditional OS and applications run in the Rich Execution Environment (REE), which lacks permission to access resources reserved for the TEE. A typical TEE will consist of both hardware isolation mechanisms, such as ARM TrustZone described in 2.1, and a trusted software stack, which communicates with the REE as described in 2.2.

2.1 TrustZone architecture

ARM Cortex processors implement the hardware portion of a TEE using the ARM TrustZone security extensions, or *TrustZone*. Under TrustZone, each physical processor core is split into two virtual CPUs. The security state of a core depends on the value of a special *Non-Secure (NS) bit*. If NS=0, then the core runs in *Secure World (SW)*. This is where the TEE software is run. If NS=1, then the core runs in *Normal World (NW)*[1], where the REE is run. The TrustZone memory and peripheral bus fabrics maintain NS bits for each memory region and I/O peripherals. Additionally, some peripherals, such as touch screens may run with either NS=0 or NS=1 at different times depending on the needs of the Secure World for exclusive hardware access.

TrustZone's fundamental security mechanism is the isolation of Secure World resources. Code running in Secure World can access memory and I/O designated for both Secure and Normal World, whereas code running in Normal World is restricted to Normal World resources. Thus, any operations on secure data (e.g., secret keys) or hardware (e.g., fingerprint reader) must be done by Secure World on behalf of Normal World. The transition of control from Normal World to Secure World is known as a *world switch* (Section 2.2).

The Secure World software stacks considered in this paper all closely resemble that of a traditional operating system. A TrustZone OS provides resource management and device drivers from a supervisor privilege level. Complementing this, a set of Trusted Applications (TAs) provide task-specific functionality from the user privilege level. Thus, TAs are restricted to their own address spaces, and are dependent on the TZOS and drivers for I/O and IPC. TAs request access to such resources through system calls to the TZOS.

Some TAs are completely driven by CA requests. A typical example of this is a cryptographic keystore to manage keys not accessible to Normal World. When a Normal World app requires an operation such as encipherment or signing, it must prepare the inputs in a shared buffer and specify which TA it wants to perform the operation (see next section for details). The TA, for its part, will wait for the TZOS to provide it with the request from Normal World, and then process the request. Thus, a typical TA will consist of a main loop to retrieve each request, and a switch structure to dispatch the specific request type (signing, decryption, etc.) to the appropriate handler.

2.2 TrustZone communication

A Client Application (CA) running in Normal World communicates with a TA as follows. First, the CA provides the Normal World kernel with the request and UUID of the destination TA. The Normal World kernel then issues a Secure Monitor Call (SMC) instruction to invoke the *Secure Monitor*, which runs at ARM exception level EL3. The monitor then interrupts the TZOS with the request and UUID, and finally the TZOS either creates a new instance of the TA, or uses it in existing instance in the case of an already-running multi-tenant TA (see 2.3). The process for returning results to the TA follows a similar path.

The request passed to the TA consists of a command ID that dictates which function to run and a shared buffer for any arguments. The shared buffer is kept in *world shared* memory, a small memory region that is accessible to both the Normal World and the Secure World. This buffer has a fixed format for all TAs depending on the underlying TZOS. For example, in Kinibi, a single shared buffer is provided for both input and output, whereas in QSEE and Teegris, separate input and output buffers are provided.

Note that at no point in the above description did any component check the TA UUID to determine if the CA was authorized to communicate with that TA. In current Android-based Normal World implementations, communication between CAs and TAs is many-to-many. In other words, communication between a given Normal World service and the set of TAs is all or nothing. This is typically regulated by using SELinux policy to restrict access to the pseudo device node and/or daemon used to notify the kernel of CA requests.

2.3 Multi-tenancy in TrustZone

The GlobalPlatform [10] defines two types of TA processes: (1) *Multi-instance TA*, created on demand for every CA initiated connection and destroyed once the connection is terminated; (2) *Single-instance TA*, created to handle all incoming requests in a single TA instance. Because multi-instance TAs start a new instance for each communicating CA, we refer to them as *single-tenant*, i.e., it is impossible for an attacker to break

[1] Also known as the Non-Secure World

into an existing session. We refer to single-instance TAs as *multi-tenant*, because the TA must manage sessions for all communicating CAs.

The TAs running under commercial TZOSes (Kinibi, Teegris, QSEE) fall into one of the above two categories. Under QSEE, all TA processes are executed as multi-tenant. In the case of Kinibi and Teegris, each TA defines its operation mode inside a signed binary segment. Then, every TA is executed accordingly in one of the previously described modes. For single-tenant configurations, a TA process is spawned by the TZOS to handle each CA initiated connection.

2.4 Storing data in Secure World

CAs leverage the communication channel described in Section 2.2 to send sensitive data to TAs. A TA accepts CA requests through a set of APIs, which either process or store received CA data, while protecting it against untrusted Normal World access. A TA's API is either *stateful* or *stateless*. Stateless APIs are straight forward. A TA receives CA data, processes it and returns a result. No data related to this process is retained by the TA across invocations. An API is stateful if it maintains state across multiple CA invocations.

Figure 1 illustrates the two methods a stateful TA API can use to persist CA data across invocations. They are as follows:

- **(A) Session state.** Typically, a TA with a stateful API will be called a number of times throughout a session with a TA. In between calls in the session, the TA maintains CA and other data in global variables, in what is essentially, the .bss section.

- **(B) Persistent storage.** TZOSes supply TAs with APIs to store data across instances. Examples include: Replay Protected Memory Blocks (RPMBs), memory blocks protected by authenticated counters to prevent malicious replays of old values and wrapped objects, which are objects encrypted and signed in a TA before being persisted in the Normal World filesystem.

While session state should only exist within a single session, persistent objects can last over many separate instances of a TA over time. Wrapped objects in particular complicate matters, because while it is infeasible for a Normal World process to decrypt them directly, they are stored in Normal World filesystem, and thus access to wrapped objects is governed by often overly permissive access control policies.

3 Problem Overview

TAs are Secure World applications that wait for Normal World CA requests, process them and return results. CAs entrust TAs with their confidential information and delegate performing sensitive operations to them. For example, CAs typically protect their private keys by storing them in Secure World cryptographic keystores.

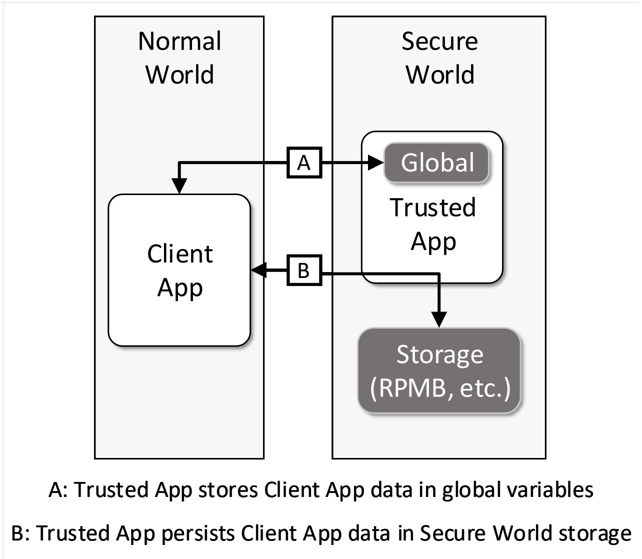
Normal World
Secure World
Global
Trusted App
Client App
A
B
Storage (RPMB, etc.)
A: Trusted App stores Client App data in global variables
B: Trusted App persists Client App data in Secure World storage

Figure 1: CA data storage in Secure World

Each TA has the responsibility of protecting CA-provided information from unauthorized Normal World access. In the case of keystores, the CA keys stored inside must not be revealed or used without the CA's explicit consent. When they fail, we have HPE.

3.1 HPE vulnerabilities

An HPE vulnerability arises when TA exposed APIs enable untrusted processes to access or manipulate CA provided data. For example, keystores contain HPE vulnerabilities if a malicious CA can obtain or use keys belonging to other CAs. In the infrastructure used by TrustZone systems, the CA-TA communication channel described in Section 2.2 allows vulnerable Normal World processes to send arbitrary messages to TAs. Each message can be used by attackers to target HPE vulnerabilities within TAs.

In this work, we study two types of vulnerabilities within TAs that allow attackers to leak, compromise or use cross-invocation maintained information.

TA multi-tenant interference. As described in Section 2.3, a TZOS either routes messages from all CAs to a multi-tenant TA or starts separate instances for each incoming CA connection. In consequence, some portions of TA instances are designed to handle simultaneous CA connections, while others operate assuming all requests are incoming from a single CA.

Ideally, every multi-tenant TA should employ proper session management to prevent CAs from affecting each other's cross-invocation states (e.g., keystores should provide an isolated key storage for each CA). In practice, these security measures can be imperfect, due to either the semantic gap between the two worlds or implementation errors. For example, TAs configured to execute as multi-tenant can be designed to only handle incoming connections from a single CA (e.g., missing session management). In such TAs, multiple concurrent CA connections can lead to HPE attacks. We detail such scenarios in Section 3.3.

Unintentional resource sharing. Section 2.4 introduces the ability of TA APIs to maintain data across multiple CA requests. Each TA can temporarily maintain CA-provided information in its global variables or requests the TZOS to persist it in Secure World storage. In both cases, this data needs to be protected against unauthorized access. In the case of global variables, each TA has exclusive access to its own memory and protecting the data within from multi-tenant interference is a matter of employing proper CA session management. In contrast, the TZOS provides all TAs with shared access to other Secure World resources (e.g., RPMB memory). Thus, a different, TZOS-enforced access control is required to ensure that CAs and TAs cannot overwrite or leak each other's data.

The TZOS manages TA access and prevents them from illegally accessing each other's contents. For example, GlobalPlatform defines a set of sharing rules in the case of PersistentObjects, under which each TA has exclusive access to its created objects, unless explicitly stated otherwise at creation. However, this is not enough to prevent CA confidential data from being exposed to malicious CA access, as the TZOS enforced access control is only concerned with access controlling access between different TAs and resources.

The lack of a fine-gained access control between CA and the Secure World resources holding their data enables multiple attack vectors to be used in order to launch HPE attacks. For example, multiple instances of the same TA have shared access to their resources. This enables malicious CAs to connect to any TA instance with access to these resources and trick them into leaking or compromising data stored inside. Details presented in Section 4.3.

3.2 Threat Model

We assume the attacker's goal is to obtain or manipulate sensitive data processed by a certain Normal World CA but is unable to compromise that CA directly. Thus, attackers cannot access its memory, hijack its execution or escalate privileges via the Normal World OS. Additionally, we assume the attacker is unable to gain code execution in any Normal World daemon, TA or the TZOS. This makes man-in-the-middle attacks out of scope. Given these limitations, the attacker may still leverage HPE vulnerabilities to access the victim CA's data.

We also assume that the victim CA depends on one or more TAs that either maintains session-level data for multiple tenants simultaneously, or persists data in secure objects, RPMB or other system-level resources protected by the TEE. Any flaws in isolating a given TA's data in these environments may lead to HPE. In order to leverage such a flaw, the attacker must compromise *any other* CA in the system, and use it to issue requests that will leak or modify the victim's data.

The attack surface of available CAs is substantial and non-static. Each CA is an application containing a library for sending `ioctl`s to a device node (/dev/mobicore, /dev/qseecom and /dev/tzdev in Kinibi, QSEE and Teegris respectively). While a Normal World daemon assists in setting up the communication channel between CA and TA, e.g., for loading the TA and setting up shared memory[2], actual requests go directly from the CA to the device node. The kernel then converts these to SMCs to schedule TAs in the TZOS. Thus, compromising a given CAs does not allow for man-in-the-middle attacks against other CAs, but is sufficient for HPE. The number of CAs is actually larger than the number of TAs seen below, as quite a few third party and mobile payment CA/TA pairs were not evaluated for this study.

3.3 Exploiting HPE vulnerabilities

The lax access control enforced by the TZOS and TAs enables a compromised CA to send malicious requests to any TA running inside the Secure World. Such a CA can leverage their access to stateful TA API in order to obtain access to data belonging to another CAs. Figure 2 illustrates how in the presence of HPE vulnerabilities a malicious CA can leak, compromise or use other CA data maintained inside the Secure World. In this figure, *Cdata* and *CKey* correspond to CA confidential data and CA cryptographic keys.

As described in Section 3, stateful TAs can store CA data either temporarily in their memory or rely on external resources to persist it for future usage upon CA requests. Six attacks are presented in Figure 2. (A)-(C) target data stored in TA sessions (e.g., global variables), while (D)-(F) target data persisted in Secure World storage.

Some TAs can store temporally values (e.g., keys, processing results) in memory to avoid retrieving or recalculating them again. These values are derived from CA-provided inputs. A malicious CA can access any data stored in TA memory between a target CA's requests, provided it can time its own requests before a shared TA clears the respective data and this data is retrievable though at least one exposed API exposed that do not perform any origin checks regarding the requests (e.g., sessions). Such attacks are most damaging when CA provided keys are cached between CA requests.

In contrast to data stored in TA memory, CA data persisted on the flash drive or Secure World resources is retained even after a TA instance is killed. This data is accessible from any TA that is allowed access to the respective resource. Moreover, this data is not usually cleared after a CA connection is terminated, providing less restrictions for attackers on timing their malicious requests. For example, in the case of single-tenant TA, any TA instance can be used by malicious CAs due to the fact that all share the same Secure World resources. The (D)-(F) data flows illustrate how malicious CA are not required to have access to TA storing the victim's data (*Cdata*). Instead, communication with other TA's could provide them with required access to the Secure World resource. Of course, in the absence of a fine-grained access control *TrustedApp1* can also be used to leak or compromise the victim's data.

Data leakage. Caching CA data inside TA memory or storing it in Secure World resources expose it to leakage through HPE attacks. The (A) and (D) data flows illustrate how once a

[2]Note that the Normal World daemons do not currently perform any access control, and will help set up any requested TA.

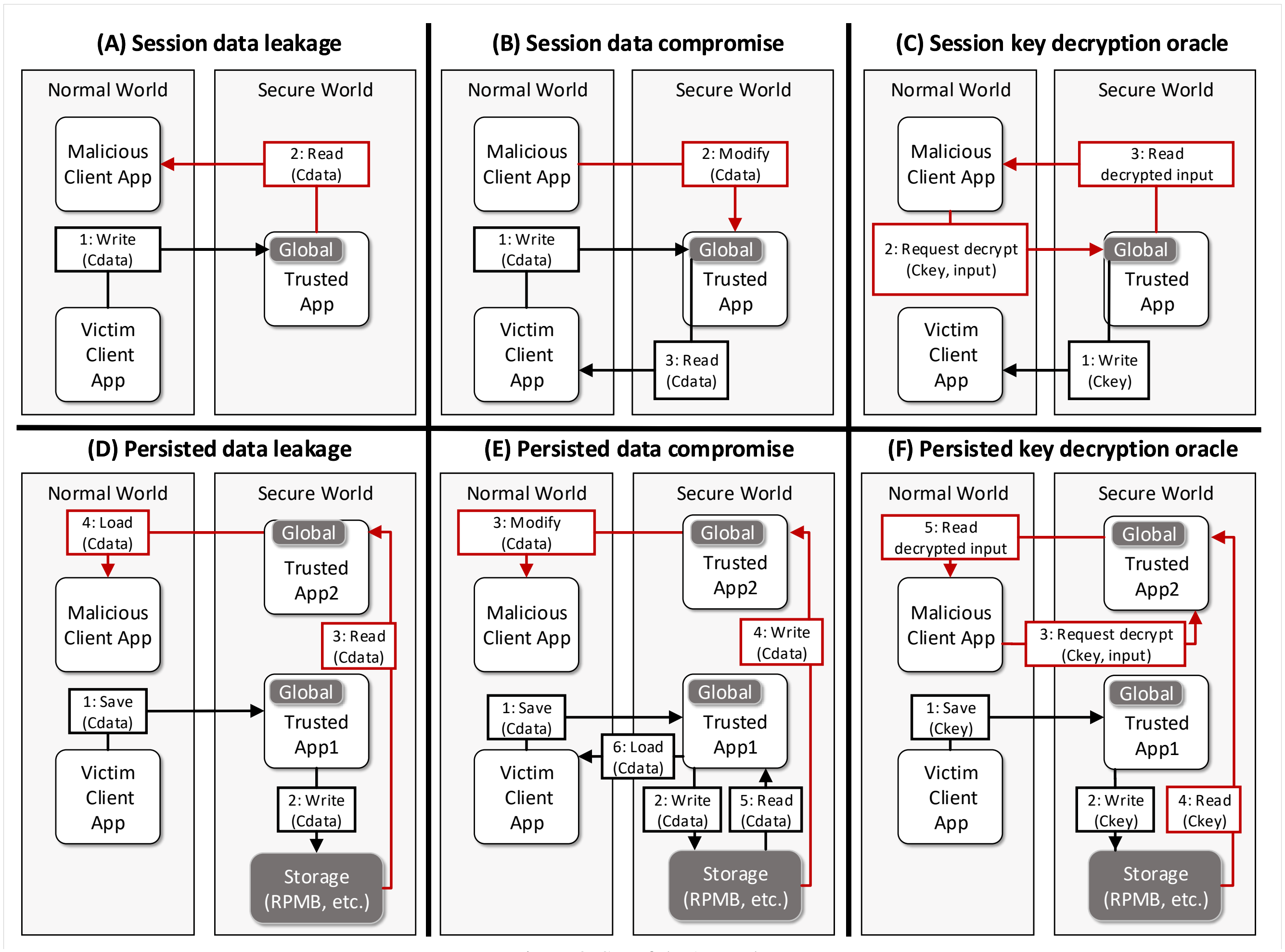

Figure 2: Stateful TA attacks

victim CA sends data to stateful TA API, the attackers can trick the TAs into providing it from the location where it maintained (e.g., TA memory or resource). This data can be leaked until it is overwritten, either through CA requests or due to other factors (e.g., TA process termination, resource failure, etc.).

Data compromise. Storing data across multiple CA requests in the Secure World protects it against Normal World access, but presents opportunities for malicious CA's to compromise it through HPE attacks. The (B) and (E) data flow illustrate the process in which a victim CA data stored or persisted across invocations can be altered by attackers before it is read back by the victim. In order to corrupt a target CA's data, the malicious CA needs to time its malicious request to execute between the victim CA's requests. Thus, in contrast to HPE data leakage attacks, data compromise through HPE is restricted to a narrower timing window.

While handling CA requests, sometimes stateful TAs store data (e.g., CA provided keys, CA verification results, etc.) in global variables in order to optimize the processing of further CA requests. Even if this data is never provided to CAs, compromising it can alter all subsequent CA request handling. For example, by overwriting cached encryption or signing keys in the Secure World, attackers can ensure that future data encryption or signing performed using the respective keys can be easily undone. Similarly, by overwriting decryption keys attackers can trick CAs into using attacker generated information in their operations, leading to data compromise. In consequence, compromising certain critical ("key") information (e.g., encryption, signing or decryption keys) can enable attackers to achieve both data leakage and compromise.

Cryptographic oracles. Some stateful TA API exposed can be leveraged by attackers even without leaking or compromising data in the TA memory or TA controlled external resource. Instead, the attackers can achieve their goals by just timing their attacks to leverage a specific state of their targeted TAs. For example, cryptographic keys maintained across multiple CA requests can be used by attackers for encrypting, decrypting or signing data on their behalf. We will refer to such TAs vulnerable to such attacks as cryptographic oracles. Data flow (C) presents an example of how a decryption key, stored in a TA session could be used by a malicious CA to decrypt data, including those ciphertexts belonging to the victim. Data flow (F) shows the same scenario in the case of keys persisted in Secure World storage.

In order for a TA to expose an API that can be used as a cryptographic oracle, the following conditions must be met:

1. A CA provides a cryptographic key to a TA.
2. The TA stores the key in TA memory or external storage.
3. The TA uses stored key to encipher CA provided data.

Opposed to the data leak and compromise attacks, to abuse these cryptographic oracles, the attackers need to perform an extra step. For decryption oracles, the attackers require a method to retrieve the target CA's encrypted data, while for encryption they need to find a way to inject their encrypted payload into the storage used by the CA (e.g., memory, flash drive, etc.). In the case of signing oracles, attacker signed data is usually used to impersonate the CA in communications with local or remote entities.

4 Results

Most TrustZone-enabled commercial devices run under either a QSEE, Kinibi or Teegris TZOS [12]. In this section we present a study of the susceptibility of TAs operating under these three TZOSes to HPE attacks and show that several of these TAs contain HPE-enabling vulnerabilities.

4.1 Evaluation Approach

For our investigation we have extracted TA binaries from the newest TrustZone-enabled mobile devices running each TZOS. We have separated the TA binaries found in Kinibi(26)-, QSEE(38)- and Teegris(31)-based devices into seven categories, based on an analysis of their functionality. Each category corresponds to the main functionality exposed through APIs by each analyzed TA. For example, Attestation TAs provide functions for creating attestation tokens, while Hardware Driver TAs are in charge of communicating with security-sensitive I/O devices. A total of 95 TA binaries were extracted on June 2019, each representing the latest version of a TA executing under one of the three TZOSes.

For approximately two months, two engineers manually analyzed each TA binary using IDA Pro [8] and reported any vulnerabilities that could enable HPE attacks. Each vulnerability has then been examined further by investigating TA and CA logic in order to determine how attackers can exploit them. The results of this analysis are presented in Section 4.3. A total of 23 vulnerabilities that enable HPE attacks have been identified (Kinibi-eight, QSEE-eight, Teegris-seven. These vulnerabilities have been identified in DRM, Key Management and Attestation TAs, which typically are either multi-tenant TAs or rely on data persisted across multiple CA sessions (either in Secure World storage or as ciphertext files on the flash drive). The other categories are typically do not perform substantial CA resource management and thus did not receive much scrutiny.

4.2 Disclosure and Vendor Response

All issues covered here were reported to the device vendor upon discovery between July 2019 and January 2020 under NDA. A final report was provided in February 2020, which included several bugs outside the scope of this paper (failure to clear keys from memory and errors when parsing secure objects). While prioritization and triage are at the vendor's discretion, their response strategy can be outlined as follows. For issues that can be addressed by modifying a single TA, (CWE-639 and CWE-862) fixes will be deployed via FOTA update. For semantic gap issues, the vendor is planning on modifying the Normal World kernel to do access control at the granularity of CA/TA pairs. This will substantially reduce the attack surface for multi-tenant TAs. We explore the full possibilities of such an approach in Section 6.5.

4.3 Discovered Vulnerabilities

Table 1 presents the breakdown of the TA categories analyzed for each vendor, indicating how many TAs have been found to contain at least one vulnerability in each category.

Table 1: Vulnerable TA summary table.
Vulnerable TAs identified / Total TAs examined.

TA Category	Kinibi	QSEE	Teegris
DRM	2 / 2	2 / 7	1 / 2
Key Management	3 / 10	3 / 11	3 / 12
Attestation	3 / 3	3 / 3	3 / 3
Hardware Drivers	0 / 1	0 / 3	0 / 5
Device Integrity	0 / 2	0 / 4	0 / 3
Authentication	0 / 6	0 / 5	0 / 3
Utility	0 / 2	0 / 5	0 / 3
Total	8 / 26	8 / 38	7 / 31

We have discovered instances of 3 types of vulnerabilities in the TAs examined, which facilitate HPE attacks:

a. CWE-639: Authorization Bypass Through User-Controlled Key (Auth-Bypass): present in multi-tenant TAs employing low entropy assignments of CA session identifiers.

A guessable session ID enables malicious CAs to obtain access to data stored in sessions maintained by the TA for CA communication in the Secure World and leak or compromise the data within. This data can include sensitive CA information (e.g., cryptographic keys, passwords, user information). Section 4.4.1 presents an example of how an instance of this vulnerability can be leveraged in order to obtain data necessary to bypass DRM license restrictions.

b. CWE-862: Missing Authorization (Missing-Auth): encountered when CA provided information is kept by multi-tenant TAs in global variables across multiple CA requests, without isolation mechanisms (e.g., session management).

c. CWE-732: Incorrect Permission Assignment for Critical Resource (Storage-Hijacking): vulnerability discovered when CA provided information is stored by TAs in TZOS provided resources that lack fine-grained access control. In contrast to Auth-Bypass and Missing-Auth, this vulnerability is not specific to multi-tenant TAs.

Table 2 shows a summary of the vulnerabilities discovered in each TA binary categories extracted from each vendor. Instances of vulnerabilities Auth-Bypass and Storage-Hijacking have been discovered in TA binaries running under all examined vendors, while Missing-Auth instances have only been discovered in TA running under Kinibi and QSEE. Under Teegris, all TAs are either configured as multi-tenant TAs and manage CAs through sessions or are configured as *Multi Instance Trusted Applications* [10] and only accept connections from a single tenant.

For each vulnerability discovered, we have studied how it can be exploited from a compromised CA. Table 3 shows a breakdown of different HPE attacks vectors that each vulnerability enables. Even though we have not identified any CA data leakage or compromise through Auth-Bypass, our results indicate that all HPE attack vectors described in Section 3 are possible using any of these vulnerabilities. We have examined each vulnerability discovered and studied how it can be exploited from compromised CAs.

Finally, Table 4 summarizes every HPE attack identified in binaries analyzed from each TZOS. These attacks can be performed through one or more HPE vulnerabilities depicted in Table 2. For example, as presented in Section 4.4.2, an Attestation TA could provide a signing oracle through both a Missing-Auth and a Storage-Hijacking vulnerability. Additionally, every one of those vulnerabilities enable one or more HPE attacks. For example, an Auth-Bypass vulnerability inside a DRM TA can enable all five classes of HPE attacks, depending on the TA provided functionality.

4.4 Vulnerability Case Studies

In this section we examine three representative reverse-engineered TA code snippets that contain real-world vulnerabilities. For each code snippet, we describe the vulnerabilities they contain, and show how they can be exploited by attackers. Case A describes how DRM protected content can be leaked through either a Auth-Bypass or Missing-Auth vulnerability. Case B presents how a how a Missing-Auth and Storage-Hijacking enable attackers to trick TAs into forging attestations. Finally, Case C presents how Storage-Hijacking vulnerabilities inside Key Management TAs can be used to obtain or alter the keys within.

4.4.1 Case A: Accessing DRM-protected content

A DRM service relies on TAs to establish secure communication channels with authorized TrustZone-enabled devices. The TA is provisioned by the DRM service with a set of cryptographic keys that can be used to decrypt protected content. We refer to these TAs as **DRM TAs**. Each DRM TA is responsible for providing access to decrypted content only to authorized CAs and ensuring the decryption keys are never leaked to the Normal World.

Listing 1 presents the basic logic present in a multi-tenant DRM TA, which is used to provide DRM-protected content to several CAs. Lines 1-25 contain the logic used for managing multiple CA sessions. Lines 27-38 present the functions `load_key` and `decipher_text`, which contain the logic for loading keys into CA sessions and using the respective keys to decrypt copyrighted content on behalf of CAs.

A CA receives a session ID from the DRM TA by calling `open_session`. The CA provides the session ID on all subsequent calls to the TA. Upon each call, the TA looks up the appropriate session via `get_session`, which searches the global map, sessions, for the provided session ID.

To play protected content, the CA receives an encrypted key from the DRM service, which it will load into the TA using the `load_key` function. This key is decrypted by the DRM TA using a manufacturer provided DRM TA unique key and stored inside the CA's session. Once the decryption key is loaded in the CAs session, the CA can request the TA to decrypt DRM-protected content by calling the `decipher_text` function.

```
1  Context struct {
2    int session_id;
3    int key[128];
4  };
5
6  /* global variables: */
7  int unique_id;
8  Context sessions[100];
9
10 void open_session
       (CA_struct input, CA_struct output) {
11     context = allocate_context_memory();
12     /* Deterministic session id assignment*/
13     context.id = ++unique_id;
14     sessions.add(context);
15     output.session_id = context.id;
16     }
17
18 /* TA-private method. Not exported. */
19 int get_session(session_id){
20     for id in range(1, 100) {
21         if (id == session_id)
22             return sessions[id];
23         }
24     return error;
25     }
26
27 void load_key(CA_struct input) {
28     Context
         current_ctx = get_session(input.session_id);
29     /* Decrypt using device unique key */
30     current_ctx.key = unwrap(input.encrypted_key);

31     }
32
33 void decipher_text
       (CA_struct input, CA_struct output) {
34     Context
         current_ctx = get_session(input.session_id);
35     /*Decrypt provided cipher text using key
36     stored in context and return result */
37     decrypt(current_ctx
       .key, input.ciphertext, output.plaintext);
38     }
```

Listing 1: Vulnerable DRM TA code

The code presented in Listing 1 contains two attack vectors that malicious CAs can use to decrypt DRM content:

(1) The session management code contains an Auth-Bypass vulnerability. First, the `get_session` function allows a malicious CA to use the key within session for the `load_key` and `decipher_text`, provided it can provide its corresponding session id. Second, the `open_session` function generates session identifiers using a monotonically increasing function

Table 2: Vulnerabilities identified inside TAs extracted from each TZOS. For each TA category row, we present the number of unique vulnerabilities identified in TA binaries, grouped by CWE type.

TA Category	**Kinibi**			**QSEE**			**Teegris**		
	CWE 639	CWE 862	CWE 732	CWE 639	CWE 862	CWE 732	CWE 639	CWE 862	CWE 732
DRM	1	4	3	1	4	3	1	0	1
Key Management	0	0	6	0	0	6	0	0	6
Attestation	0	1	8	0	1	8	0	0	8
Total	1	5	13	1	5	13	1	0	13

Table 3: Vulnerability impact breakdown.

HPE attack	**All TZOSes**		
	CWE-639	CWE-862	CWE-732
Data leakage	0	4	21
Data compromise	0	4	15
Decryption oracle	9	2	9
Encryption oracle	3	2	9
Signing oracle	6	2	15
Total	18	14	69

at Line 13. This enables attackers to guess a victim CA's session id in a reasonable time by trying to decipher texts using random session ids between 1 and 100.

(2) A Storage-Hijacking vulnerability is present in the `load_key` function. On Line 30, the DRM TA loads the decryption key inside the CA's session by decrypting it from a CA provided ciphertext. This enables a malicious CA to load other CA decryption keys inside its own session, provided it possesses the respective key's ciphertext. Section 6.1 details how attackers can obtain ciphertexts containing victim CA's keys from the Normal World filesystem.

Under Auth-Bypass, the attackers have to wait for a victim CA to load the key in its session and then obtain access to the key by guessing the corresponding session id. Once the key is loaded, the attacker can decrypt protected content until the victim CA asks the TA to close its session. In contrast, once an attacker obtains a victim's ciphertext, the Storage-Hijacking vulnerability allows it to use the key inside at any time.

4.4.2 Case B: Forging device attestation

Attestation TAs provide a signed attestation blob that acts as a proof of device identity and low-level software integrity. Typically, the attestation blob is provided to external services as proof of the identity and integrity of the device. The attestation TA accesses measurements of the bootloader, TZ and kernel that ran when the device was powered on. All of this information is collected into an attestation blob, along with additional configuration information.

The attestation blobs are signed using attestation keys generated by remote parties. These keys are provided to device manufacturers, which encrypt them using an Attestation TA unique key and store them inside the Normal World filesystem. These blobs should only be signed by Attestation TAs running on uncompromised devices and provide reliable information regarding the device's identity to remote parties.

Listing 2 presents an overview of how an Attestation TA generates and provides attestation blobs to CAs. A CA first initializes the TA by providing an attestation key ciphertext to the `init_attestation` API. After verifying the device's integrity, this API unwraps (decrypts) the key into a global variable. Once the TA is initialized, the CA can call the `sign_attestation_data` API in order to ask the TA to generate attestation data from a CA provided International Mobile Equipment Identity (IMEI) and Media Access Control address (MAC) sign it using the key stored inside the global variable.

```
1  /* global variable: */
2  int attestation_key[128];
3
4  void init_attestation(CA_struct input) {
5      if (device_integrity_intact()) {
6         attestation_key = unwrap(input.
       encrypted_key); {
7      }
8  }
9
10 void sign_attestation_data
       (CA_struct input, CA_struct output) {
11     attestation_data
       = generate_attestation(input.IMEI, input.MAC);
12     output
        = sign(attestation_key, attestation_data);
13 }
```

Listing 2: Vulnerable Attestation TA code

Two vulnerabilities can be observed in Listing 2's code:

(1) The `sign_attestation_data` API assumes that a CA has to call the `init_attestation` API in order to provide the key used for signing attestation data. However, this assumption is only valid in single-tenant TA instances. In multi-tenant TA instances, this code contains a Missing-Auth vulnerability, which enables malicious CAs to sign attestation data using keys installed by other CAs.

(2) A Storage-Hijacking vulnerability is present in the `init_attestation` API. This decryption logic used at Line 6 enables a malicious CA to trick the TA into installing another CA's attestation key in the global variables, provided they can obtain the corresponding key's ciphertext. Section 6.1 details the process of obtaining this ciphertext. This vulnerability affects both single-tenant and multi-tenant Attestation TA instances.

Using either vulnerability, a malicious CA can use a victim CA's attestation key to sign attestation blobs containing the IMEI and MAC of another, compromised device. These blobs can then be moved onto the compromised devices. This

Table 4: HPE-attack vectors identified in each TZOS. For each TA category row the columns present HPE attack vectors identified through one or more HPE vulnerabilities. Multiple HPE attack vectors also stem from a single vulnerability.

	TA Category	**HPE attack**					
		Data leakage	Data compromise	Decryption oracle	Encryption oracle	Signing oracle	Total
Kinibi	DRM	2	2	3	2	2	12
	Key Management	3	3	0	0	0	6
	Attestation	1	0	2	1	5	9
QSEE	DRM	2	2	3	2	2	12
	Key Management	3	3	0	0	0	6
	Attestation	1	0	2	1	5	9
Teegris	DRM	2	2	2	1	2	9
	Key Management	3	3	0	0	0	6
	Attestation	1	0	2	1	5	9

enables any application running on the compromised device to circumvent remote attestations by spoofing the victim CA's identity using the previously generated attestation blob. The applications would appear to the remote party as the victim CA running on the uncompromised device.

4.4.3 Case C: Leaking & altering other CA keys

We refer to TA's designed to protect CA provided information from unauthorized Normal World access as **Key Management TAs**. These TAs are typically responsible for generating key material, protecting it on behalf of the CAs and using it to perform cryptographic operations (e.g., encryption, decryption, signing) inside the Secure World. Most also enable CAs to provide them with keys for safekeeping, allowing them to retrieve them when needed.

Listing 3 presents two APIs provided by such a Key Management TA. A CA can request the TA to protect its cryptographic keys using the `store_key` API and retrieve them back when needed using the `load_key` API.

```
1 void
      store_key(CA_struct input, CA_struct output) {
2     output.encrypted_key = wrap(input.
      plaintext_key);
3     }
4 }
5
6 void load_key(CA_struct input, CA_struct output) {
7     output.plaintext_key = unwrap(input.
      encrypted_key);
8 }
```

Listing 3: Vulnerable Key Management TA code

The TA has to persist the received keys for an undefined amount of time. Thus, the keys cannot be maintained in memory, where they would be lost when the TA process is killed or the device is rebooted. Instead, the Key Management TAs uses the `wrap` function at Line 2 to encrypt keys received from the CA using a Key Management TA-specific key and provides the resulting ciphertext back to the CA, relying on the CA to maintain it on the flash drive until needed.

In the intended scenario, a CA's key integrity should be maintained inside the Normal World filesystem and only the respective CA should be able to recover it using the `unwrap` function exposed by the TA as the `load_key` API. In practice, each API in Listing 3 contains a Storage-Hijacking vulnerability that enables performing an HPE attack:

(1) The `load_key` API decrypts at Line 7 any CA provided ciphertext, without verifying if the respective CA should be allowed to access the contents within. This API enables a malicious CA to obtain the keys within any Key Management TA-generated ciphertext, provided they have permissions to read its corresponding file.

(2) The `wrap` function used by the `store_key` API at Line 2 enciphers any CA provided data using the same Key Management TA-specific key. A malicious CA can use this API to replace CA ciphertexts with its own enciphered keys, provided it has permissions to alter the ciphertext files.

Depending on the key's purpose, vulnerability (1) can enable attackers to sign or encrypt data using a victim CA's key or decipher any CA encrypted information. Similarly, vulnerability (2) can be used to trick CAs into performing their own cryptographic operations using attacker provided keys.

Similar vulnerabilities have been discovered inside Key Management TA that rely on RPMB protected storage to protect CA provided keys. A key difference is that instead of providing the ciphertext to the CAs, these TA's rely on the TZOS to write and read chunks of RPMB blocks. The vulnerabilities inside these TAs stem from the reliance of data inside RPMB storage, which is only protected by a coarse-grained access control. Under this access control, malicious CAs can use the Key Management TA APIs to leak or modify keys stored inside RPMB storage, without even requiring access to their ciphertexts.

5 HOOPER: Automating HPE detection

TA API interface security is not uniform across all devices. Even devices operating under the latest TEE coding standards specified in the GlobalPlatform's *TEE Internal Core API Specification* [10] contain TAs vulnerable to HPE attacks. Moreover, mobile devices are not patched uniformly, so some run older TA versions that might be still vulnerable to such attacks. Inspecting all deployed TAs on all devices requires either the development of automatic HPE attack detection tools or manually inspecting each TA binary version manually.

In this section, we present the design of HOOPER, an angr [28]-based tool designed to detect the HPE categories presented in Section 3. HOOPER uses symbolic execution to locate paths where CA data is persisted across invocations. Figure 3 illustrates the HOOPER's analysis process. This analysis consists of three phases:
Phase 1: Track TA's internal handling of CA provided data though each of its execution paths. At the end of this process, a set of path semantics is obtained. These path semantics capture all events in which CA data is stored in TA memory, external resource or information loaded within these locations is returned CAs.
Phase 2: Identify potential flows of CA data across multiple TA execution paths by analyzing cross-invocation data flows through TA global variables and external resources.
Phase 3: Analyze the obtained CA data flows, identifying the sequences of TA API invocations that lead to TA data processed data to be leaked or corrupted.

The rest of this section details the inner-workings of each described phase and how they were implemented to analyze TA binaries under the Teegris TZOS.

5.1 Phase 1: Inner-invocation data flows

Phase 1 identifies execution paths that write or read CA data to or from global namespaces. This implies that first we have to identify how TAs receive data from CAs and then emulate its processing. Data can be passed to TAs from the Normal World only though a set of standardized API interfaces. These API interfaces only allow TAs to send or receive data through a set of predefined input/output buffers. Tracking CA data received by TAs implies first marking the TA API's input buffer contents at the beginning of each CA request processing.

To simulate all possible processing of CA data, we symbolize the data within TA API input buffers and provide it with a semantically meaningful name, indicating the respective data represents data provided by the CA. We then build our analysis on top of the angr [28] symbolic execution engine to simulate its processing.

The names placed on symbolic data are preserved during the symbolic execution of TA binary code, propagating automatically through arithmetic operations. However, TA instances also rely on external functions (e.g., kernel APIs, IPCs) and libraries in their data processing flows.

As it is impractical to symbolically execute all library and IPC dependencies of a given TA, we leverage angr's simulated procedures (SimProcedures) as lightweight replacements. To ensure that semantically meaningful symbols are preserved, we carefully construct the SimProcedures to propagate the input symbols to the output. For example, the SimProcedure for AES encipherment performs no cryptography on symbolic data, as this can lead to constraint explosion. Instead, it produces a symbolic ciphertext consisting of new symbols that inherit names derived from the plaintext input's symbols.

For each execution path, we record path semantics corresponding to (1) data being written to TA global variables, external resources and TA API output buffers; (2) data being read from TA global variables and external resources; (3) data used as keys for cryptographic operations.

The path semantics of the recorded events pertaining to CA data or uninitialized global variables are then forwarded for further processing.

5.2 Phase 2: Cross-invocation data flows

In the second phase, we identify data that flows across sequences of TA API invocations. For example, during an API invocation data might be stored in a global variable. This global variable could then be read and used during a subsequent API invocation.

The path semantics recorded during Phase 1 provide the required information that enable identifying these cross-invocation data flows, missed during symbolic execution. Each path semantic retains the execution path, data being read/written and its size alongside with the source of data reads and data write destinations.

In this phase, HOOPER correlates read/write pairs of semantic paths. Figure 4 illustrates the 2 types of data matching performed in this phase: (1) every data copied into global variables on one execution path is paired with all uninitialized reads from the respective global variable, encountered on the other execution paths; (2) any data provided to a TZOS storage location (e.g., RPMB block) on one execution path is paired with all attempts to read data from the respective TZOS storage location encountered on other execution paths.

At the end of the second phase the detection tool produces a set of paired execution paths, each representing an API call. These paths are linked together by the data flowing between them. For example, in Figure 4 the path writing data into *X* is linked together with the paths reading from *X*. Similarly, the paths loading and storing *Y* are paired together.

5.3 Phase 3: Identifying exploitable TA execution data flows

In this phase the execution paths paired during Phase 1 are examined for the HPE attacks detailed in Section 3.3. For each pair, the CA data provided as input is tracked and the HOOPER determines if this information can be used for performing HPE attacks using two rules: (1) Data leakage or compromise: CA data read during one execution path flows through other execution paths back to the CA unencrypted. (2) Cryptographic oracle: CA data read during one execution path is used in other execution paths as the cryptographic key for decrypting, signing or encrypting CA information.

Information maintained inside the TA binary (e.g., TA configuration) is used to prune out any execution paths that cannot be reproduced on real devices. For example, under Teegris and Kinibi, the execution paths paired through global variables have to be discarded in the case of single-tenant configured TAs. Such TAs would not be to multi-tenancy related vulnerabilities.

At the end of this phase, the set of execution paths that can be used as cryptographic oracles or enable either data leakage

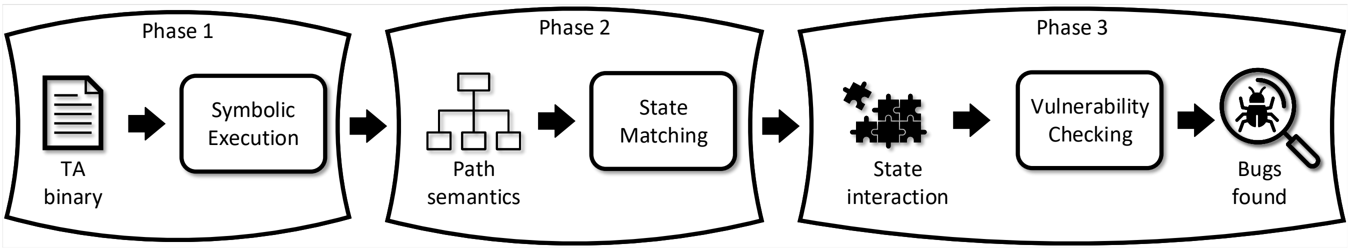

Figure 3: Automatic stateful TA vulnerability detection process

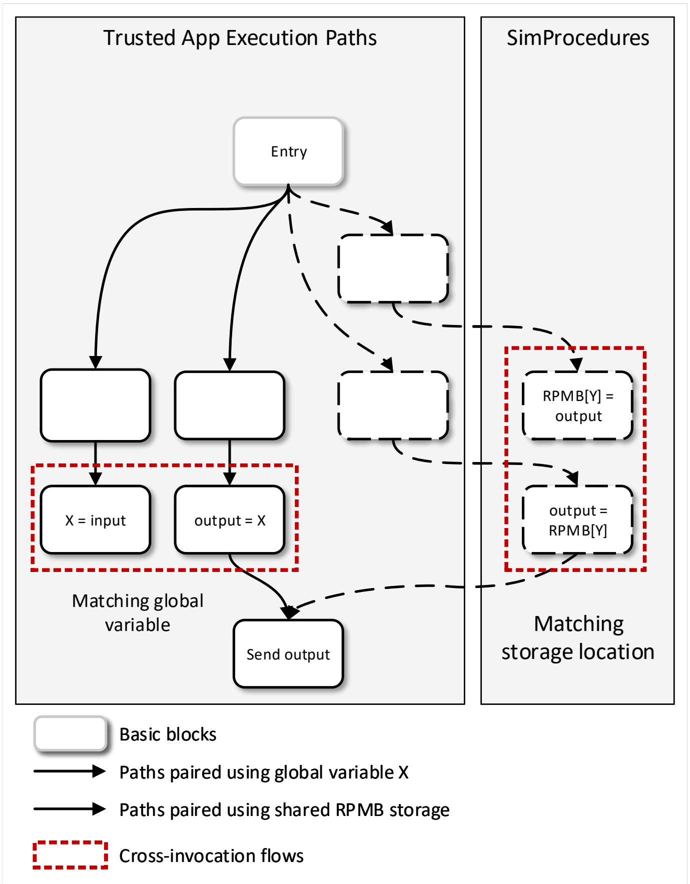

Figure 4: Detecting cross-invocation data flows

or compromise are reported along with their constrained and input required to reproduce the attack vector.

5.4 HOOPER Implementation Details

The total codebase of HOOPER consists of 5088 LOC, 1324 of which corresponds to emulating Teegris specific functionality, including CA input data modeling and TZOS provided APIs as SimProcedures. We implemented 71 SimProcedures to emulate 68 external library calls and three intractable methods encountered that caused state explosion. Note that the majority of Teegris-specific code was pre-written for a different memory-safety analysis.

Four TA storage methods discovered to be available under Teegris: **Global variable**, **Persistent Objects**, **Secure Objects** and **RPMB blocks**. The later three are all accessed through TZOS APIs and thus have been SimProcedures. Each corresponding SimProcedure collects the path semantics and details regarding the operation performed. For example, when an execution path writes to an RPMB block, the corresponding SimProcedure collects details regarding the data written, its length, the RPMB block offset and basic blocks leading to this operation. In the case of global variables, reads and writes Global variable data read and write semantics are tracked by hooking a set of logging methods into operations performed on the .bss ELF segment mapped into memory. These logging methods also collect similar path semantics details.

By tracking the start and end of each data read and written, HOOPER identifies each cross-invocation data flows. Cross-invocation data flows are represented as the data contained within the intersection of bit arrays written with bit array read. For example, an execution path that writes ten bits on RPMB at offset 50 is only paired with those reading any data from the RPMB blocks between 50 and 60 and others that read from other storage or from RPMB blocks outside this range. Once all the cross-invocation flows are identified, their corresponding paths are pruned using the process described in Phase 3 and the remaining paths are signaled as enabling a type of HPE, based on the operation performed.

5.5 HOOPER Evaluation

Teegris binaries are analyzed using our symbolic analysis tool HOOPER. Tracking cross-invocation data flows enables HOOPER to identify data leakage, compromise and the various cryptographic oracles directly, instead of identifying potential HPE vulnerabilities. Moreover, the execution paths leading to each are provided for analysis and can be used to easily reproduce the identified attacks. As a result, the manual analysis time of four weeks can be reduced to 24 hours for vulnerabilities reachable through symbolic execution.

In the case of each binary, the TZOS APIs are simulated using symbolic procedures and HOOPER is configured to track CA information through the TAs execution. For each Teegris binary, HOOPER is configured to run for twenty-four hours or until it cannot find any new execution paths in ten minutes. An overview of the vulnerabilities found using HOOPER is presented in Table 5.

False Positives. HOOPER does not report any false positive HPE attack during the 24 hour experiments. This is due to two main reasons. First, legitimate data sharing between CAs via TA APIs is not exceedingly common. During our manual analysis, we identified one case not reached by HOOPER where data is intentionally shared between CAs. Specifically, at boot time, a CA saves a set of secure boot flags using a Utility TA, which provides a read-only API for these values. Had HOOPER reached this case, it would have constituted a false positive.

The second reason for the lack of false positives is that most multi-tenant TAs that maintain session data exhibit either CWE-639 or CWE-862, and thus are unable to securely isolate data to a given session. Any inter-invocation flows found by HOOPER in such a TA would be true positives. For multi-tenant TAs with proper session handling, HOOPER will either need to be made aware of the session semantics, or it will exhibit some false positives. However, manually analyzing a relatively small set of false positives is still preferable to a full manual analysis in the absence of the tool.

False Negatives. All nine Attestation TA HPE attack vectors are signaled by the tool. In the case of Key Management TAs, two out of six HPE attacks are signaled. Four of the HPE attacks identified manually are not reported by HOOPER because Phase 1 does not record their corresponding path semantics. A series of path explosions during symbolic execution leads to missing these semantics. Path explosions represent an inherent limitation of symbolic execution which occur when the number of feasible paths grows exponentially.

In the case of one Key Management TA, the path explosion occurs during the exploration of a large number of complex TA functions. This path explosion is generated once the symbolic execution start exploring arbitrary one of the 23 API functions present inside this binary. Most of these functions perform cryptographic transformations (e.g., encryption, decryption, integrity verification) on symbolic data corresponding to CA input and RPMB stored information. The two vulnerabilities we have identified manually inside this TA are at the bottom of two such functions. In order for HOOPER to identify these vulnerabilities, both functions have to be completely explored during Phase 1. In our 24 hour window experiments, the complex functions containing the vulnerabilities have not been reached, leading to one data leakage and one data compromise false negative.

Complex input processing functions also lead to missing another set of data leakage and data compromise HPE attacks present in a second Key Management TA. This TA is designed to receive serialized CA information. As a result, a decoding operation takes place inside TA code, in the initial stages of the CA input processing. The decoding function transforms a provided CA buffer into data structures using a series of loops. The iteration number for each loop is also extracted from within the serialized buffer. This presents a problem for the symbolic execution performed in Phase 1. Here the CA provided buffer is made completely symbolic, including the loop iteration numbers. This leads to each loop being executed an arbitrary number of times. In consequence, the symbolic execution slows to a crawl once these decoding functions are reached. The data leakage and data compromise vulnerabilities are located beyond these decoding functions are never reached in our 24 hour window experiments.

Eight of the nine DRM HPE attacks are also reported by HOOPER. In this case the HPE attack is missed due to the semantics of the decryption oracle inside the DRM TA rather than a path explosion problem. The decryption function inside this TA does not explicitly decipher CA information itself. Instead, it provides the addresses of the buffers corresponding to the CA input and output to a cryptographic hardware using an ioctl. This operation is not emulated faithfully yet in our HOOPER prototype. As a result, the link between the CA encrypted buffer containing the key and the information deciphered using this key is lost during symbolic execution. The DRM false negative is a consequence of not emulating and tracking the external hardware decryption performed.

In summary, one false negative is a result of incomplete reproduction of all Secure World OS APIs available to TAs. The other four false negatives are a result of path explosion encountered in more complex TA binaries and can be addressed by incorporating in HOOPER advances in the field of symbolic execution. Solving the long-standing problem of path explosion is out of scope for this work.

Finally, the HOOPER prototype only analyzes cross-invocation data flows between pairs of execution paths, corresponding to the HPE attacks depicted in Section 3.3. More complex HPE attacks could require performing a series of TA API calls in a particular sequence. For example, some TAs have an initialization call that allocates heap memory, storing a pointer to that memory in the global section. Subsequent calls then place session data in the heap memory. Though we have not identified such HPEs, investigating their presence is subject of future work.

6 Mitigations

The vulnerabilities detailed in Section 4.3 are present in TA APIs that incorrectly manage CA provided information in their cross-invocation states. Auth-Bypass and Missing-Auth are caused by either missing or faulty session management. Missing-Auth is due to the reliance on encrypted data stored in attacker accessible locations. In this section, we review potential mitigations for each HPE vector.

6.1 Protecting TA data stored in Normal World

As described in Section 3.3, data leakage and data compromise HPE attacks only require access permissions to communicate with the corresponding vulnerable TA. In contrast, exploiting HPE attack vectors corresponding to encryption oracles additionally require altering a victim's ciphertext, while decryption oracles require this ciphertext to be provided to the TA. In this section we assess the difficulty for an attacker to access these ciphertexts stored inside Normal World and argue for increasing their isolation in order to prevent their use in HPE attacks.

Under all three TZOSes studied, the internal flash drive is under the control of the Normal World OS. In consequence, the TZOS cannot provide TAs with direct access to it. Instead, when TAs want to persist information on the flash drive, the TA have to encrypt the respective information and rely on CAs to store and retrieve it.

Table 5: HOOPER-detected HPE attacks. HOOPER-signaled HPE attack vectors / HPE attack vectors identified manually

	TA Category	HPE attack					
		Data leakage	Data compromise	Decryption oracle	Encryption oracle	Signing oracle	Total
Teegris	DRM	2 / 2	2 / 2	1 / 2	1 / 1	2 / 2	8 / 9
	Key Management	1 / 3	1 / 3	0 / 0	0 / 0	0 / 0	2 / 6
	Attestation	1 / 1	0 / 0	2 / 2	1 / 1	5 / 5	9 / 9

Maintaining TA sensitive information as ciphertexts on a Normal World controlled flash drives prevents attackers from reading the contents within. However, the encryption does not prevent attackers from altering or obtaining the respective ciphertexts. Orthogonal protection methods are required in order to prevent such unauthorized access. These protection methods can only be provided from inside the Normal World, because the Secure World cannot prevent access to data maintained inside Normal World storage.

Our investigation of the devices running the extracted TA binaries has revealed that most examined ciphertext files are located inside folders within the *efs* partition. Access to these files is guarded by SELinux policies. Thus processes are prevented from accessing these files unless they belong to one of the categories provided with access. However, we have discovered ciphertexts mostly inherit the labels assigned to folders inside the *efs* partition. As a result, numerous SELinux labels are granted write or read permissions to these ciphertext files. We have discovered a total of 157 SEAndroid labels have read permissions to labels assigned to at least one ciphertext file. 57 labels also have write permissions. In a particular case, these labels even include all *System Apps* preinstalled on the device (154 executables). Code-hijacking vulnerabilities within any process executing under one of these labels would be sufficient for obtaining access to CA stored ciphertexts. For example, among these labels there is a system process that is permitted by Android to both communicate with TAs and access all examined ciphertext files maintained inside the *efs* partition. Previous work [13] details how vulnerabilities inside this process have previously allowed attackers to send malicious SMCs. In conjunction to access to ciphertexts, vulnerabilities in such a process would be sufficient to exploit HPE vulnerabilities related to encryption and decryption oracles.

In summary, our investigation regarding the security of CA managed ciphertexts has revealed multiple vectors of obtaining permissions to access or alter their corresponding files. Increasing the isolation of these ciphertexts would help prevent attackers from obtaining or altering the contents within. For example, using finer-grained SELinux policies [23, 24, 34] or similar fine-grained access control could help mitigate the effects of decryption and encryption oracle HPE vulnerabilities.

6.2 Resolving multi-tenant interference

Missing-Auth vulnerabilities occur when multiple CA have access to a single-tenant designed TA. In this section we present two solutions for addressing this issue: (1) revising the TA's design to use sessions for managing connections incoming from multiple tenants or (2) restricting access to such TAs to a single Normal World process.

In Teegris and Kinibi devices, the Normal World OS uses a coarse-grained Linux policy to allow Normal World processes to communicate with TAs. Under this policy CAs can connect to any TA. Such instances can only receive requests from one CA during their lifetime. Each CA that tries to connect to such a single-tenant TA is provided with their own TA instance and access is denied if no such instance can be provided.

Our evaluation of the two TZOSes shows that providing each CA a TA instance can help avoid introducing Missing-Auth vulnerabilities. Under Teegris no Missing-Auth vulnerabilities have been identified, as all TA binaries examined either are configured as single-tenant instance or manage incoming CA connections through sessions. In Kinibi, Missing-Auth vulnerabilities have been only been identified in TAs misconfigured to run as multi-tenant instances.

Normal World SELinux policies are also used in QSEE devices to determine which Normal World process can access TA-provided APIs. However, since QSEE lacks the support for single-tenant TAs, all TAs are single instances that accept incoming requests from all CAs. In consequence, under QSEE, any TA that stores cross-invocation data in global variables is required to use session management to prevent Missing-Auth vulnerabilities.

In summary, all identified Missing-Auth vulnerabilities can be resolved by introducing session management into multi-tenant TAs. In the case of Kinibi, the exploitation of these vulnerabilities can also be prevented by re-configuring the vulnerable TAs to execute as single-tenant instances.

6.3 Standardizing session management

The presence of Auth-Bypass vulnerabilities in TAs running under all examined TZOSes indicates that relying on each TA to implement proper session management is not ideal. The Auth-Bypass vulnerabilities have to be individually identified and fixed by patching corresponding TA binary code.

In order to make multi-tenant TAs less prone to Auth-Bypass vulnerabilities we propose each TZOS provides a library for session management. Such a library would transfer the session management responsibility from the individual TAs to the TZOS. Auth-Bypass vulnerability would be eliminated once library implements proper session management and all TAs use it to manage CA connections. For example, the TZOS specified methods (e.g., `TA_OpenSessionEntryPoint`, `TA_InvokeCommandEntryPoint`) for CA-TA communication could be restricted to single-tenant TAs. In the case of multi-tenant TAs, these methods would be implemented instead in a library like SMlib [21] provided back to TAs in the form of APIs with inherent session management.

Under QSEE and Kinibi, the session management library would have to be statically linked inside each TA binary. In contrast, under Teegris, all TAs could use the same dynamically loaded library (DLL). Using a DLL would facilitate maintaining session management code, as only the library would have to be updated instead of individual TA binaries.

6.4 Protecting CA information stored by TAs

HPE vulnerabilities are introduced when CA information is stored inside attacker accessible resources. Thus, in order to resolve these vulnerabilities, strict access control can help both resolve existing HPE vulnerabilities and prevent them from being introduced in the future.

In the case of encrypted CA information stored in Normal World storage (e.g., the internal flash drive), access to ciphertexts is determined by the Normal World OS. Thus, under the threat model presented in Section 3.2, using the least privilege principle [26] can help prevent attackers from obtaining ciphertexts belonging to a CA and prevent using them to perform HPE attacks. For example, the HPE vulnerabilities targeting the wrap / unwrap functions in the three case studies presented in Section 4.4 can only be exploited if the attacker can read or alter ciphertexts used by CAs.

Strict access control enforced by the TZOS can only be used to prevent unauthorized access to CA information maintained in Secure World-provided resources. However, in order to enforce such access control, the TZOS has to (1) provide each CA with their own isolated space inside the Secure World resources and (2) determine on behalf of which CA a TA is attempting to read or write data within this space.

There are multiple ways a TZOS can implement (1). For example, the Secure World resources can be partitioned and each CA restricted to only to access its own partition. Alternatively, the TZOS can maintain and use CA specific tags to determine access to the CA information stored in such resources. Implementing (2) is more difficult under TrustZone, because only the Normal World OS is the only entity capable of correctly identifying each CA process. The TZOS has to rely on information provided by Normal World in order to only provide specific CAs access to Secure World resources.

In summary, fine-grained access control policies approaching least privilege provide a means to prevent malicious CAs from accessing CA information stored by TAs and launching HPE attacks. However, a benign and uncompromised Normal World OS is required for reliably enforcing these policies.

6.5 Minimizing Normal World access to TA's

Under all TZOSes studied, once a CA is granted access by the Normal World OS to communicate with TAs in inside the Secure World, the respective CA is permitted by the TZOS to send requests to any API of the TAs running under it. Under this coarse-grained access model, all vulnerabilities present inside TAs can be exploited by attackers once they manage to compromise a single Normal World CA.

Currently, the TZOSes do not restrict CA access to TAs because they are not able to reliably determine their identity due to the semantic gap between the two worlds. However, as mentioned in the previous Section, the TZOS can receive the required information from an uncompromised Normal World kernel. If the Normal World OS is modified to include the origin of requests sent to the TA APIs, a fine-grained access control could be enforced by the TZOS in order to prevent CAs from arbitrary accessing TA APIs.

Under a fine-grained access control, each TA could notify the TZOS of which CAs are authorized to communicate with it. For example, DRM TAs could provide a list of IDs of pre-approved DRM CAs that are allowed to access their APIs. These IDs would be read from signed CA binaries by the Normal World OS and provided to the TZOS.

In order to exploit vulnerabilities inside a TA, attackers would be forced to either compromise specific CAs or compromise the Normal World OS. Systems like TZ-RKP [2] and SPROBES [9] can be employed by the TZOS to maintain the integrity of the Normal World OS. Thus, minimizing access to TA API reduces the impact of vulnerabilities inside CA code and raises the bar for attackers that try to leverage them into escalating their privileges into the Secure World or other CAs.

7 Related Work

TrustZone has been developed to guard sensitive data from untrusted software. Initial TrustZone research has mainly focused on moving security-sensitive operations (e.g., confidential data storage [16], one-time-password generation [30], ad fraud detection [15], attesting logins [17], mobile payments [33], memory acquisition [31]) inside the Secure World environments for protection from the untrusted Normal World OS. Such operations are always performed on behalf of a Normal World application and are present in modern TrustZone devices in the form of TAs. For example, the Key Management and DRM TAs follow DroidVault's concept of trusted data vaults. In this work we show that HPE attack vectors are introduced when security-sensitive operations such as these are moved inside the Secure World.

Several works have identified memory corruption vulnerabilities in TAs that result in impact similar to this work such as leaking secrets and altering TA memory [3, 4, 12, 13, 25]. Cerdeira et al. [6] present a study of several such issues. In contrast, the HPE attack vectors presented in this work show how it is possible to leak secrets and alter TA information without requiring TA memory corruption vulnerabilities.

The HPE attack vectors are a manifestation of the TA confused deputy problem. The closest context in which this problem has been studied is Boomerang [18] attacks. Similar to HPEs identified in this work, Boomerang 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. In contrast, in this work we present vulnerabilities stemming from TAs providing untrusted applications with unauthorized access to CA data maintained in both worlds. The confused deputy also appears in the context of IPC between Android applications [5]. However, the framework proposed cannot be extended to TAs due to the semantic gap present between the two words.

Attestation forgery attacks have been introduced in the form of cuckoo attacks [22] in the context of physical TPMs [32] and Goldeneye [14] in the context of virtual TPMs. These attacks trick remote servers that rely on TPM generated attestations into establishing trust with potentially compromised clients. The vulnerable Attestation TAs identified in Section 4.3 similarly enable variants of these attacks in the context of TrustZone devices.

Binary dynamic analysis has been used in previous work to identify vulnerabilities within binaries. Firmalice [27] examines execution paths for authentication bypass vulnerabilities. Driller [29] and Mayhem [7] combine symbolic execution with fuzzing techniques in order to reach deep memory corruption bugs. These tools examine individual execution paths for vulnerabilities. In contrast, HOOPER identifies HPE-enabling vulnerabilities by examining data flows across multiple execution paths.

8 Conclusion

The semantic gap at the interface between TEEs (e.g., TrustZone Secure World) and external logic (e.g., Normal World OS) can introduce significant Horizontal Privilege Escalations and other security vulnerabilities. Properly bridging the gap requires careful coordination and co-design between the TEE and external logic. This is not always easy, due to the nature of today's software delivery chains involving numerous unrelated principals.

In TrustZone, multi-tenancy combined with statefulness in TEE-hosted code leads to significant HPEs leaking sensitive session data and providing cryptographic oracles, even in the absence of code execution vulnerabilities in TEE logic. Mitigations bridging this gap require tight coordination between the kernels in both worlds.

Acknowledgments

We thank Lee Harrison and Hayawardh Vijayakumar for providing us with insight regarding the inner-workings of TZOSes and TAs. We thank Zhongjie Wang for his guidance in using symbolic execution. We thank Michael Grace for his suggestions in improving the presentation of our work. We thank our shepherd Adam Bates and the anonymous reviewers for their helpful feedback. The authors would also like to thank Samsung Research America and the Office of Naval Research who funded parts of this work through award N000141812043.

References

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

[2] AZAB, A. M., NING, P., SHAH, J., CHEN, Q., BHUTKAR, R., GANESH, G., MA, J., AND SHEN, W. 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* (New York, NY, USA, 2014), CCS '14, ACM, pp. 90–102.

[3] BENIAMINI, G. TrustZone Kernel Privilege Escalation. http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html.

[4] BERARD, D. Kinibi TEE: Trusted Application exploitation. https://www.synacktiv.com/posts/exploit/kinibi-tee-trusted-application-exploitation.html.

[5] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Towards taming privilege-escalation attacks on android. In *NDSS* (2012), vol. 17, Citeseer, p. 19.

[6] CERDEIRA, D., SANTOS, N., FONSECA, P., AND PINTO, S. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *Proceedings of the IEEE Symposium on Security and Privacy 2020* (01 2020).

[7] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 380–394.

[8] EAGLE, C. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, USA, 2008.

[9] GE, X., VIJAYAKUMAR, H., AND JAEGER, T. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *CoRR abs/1410.7747* (2014).

[10] GLOBALPLATFORM. Tee client api specification v1.0. https://globalplatform.org/specslibrary/tee-client-api-specification/.

[11] HARDY, N. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev. 22*, 4 (Oct. 1988), 36–38.

[12] HARRISON, L., VIJAYAKUMAR, H., PADHYE, R., SEN, K., AND GRACE, M. Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020) (To Appear)* (August 2020).

[13] KOMAROMY, D. Unbox Your Phone. https://medium.com/taszksec/unbox-your-phone-part-iii-7436ffaff7c7, 2008.

[14] LAUER, H., SAKZAD, A., RUDOLPH, C., AND NEPAL, S. Bootstrapping trust in a "trusted" virtualized platform. In *Proceedings of the 1st ACM Workshop on Workshop on Cyber-Security Arms Race* (New York, NY, USA, 2019), CYSARM'19, Association for Computing Machinery, p. 11–22.

[15] LI, W., LI, H., CHEN, H., AND XIA, Y. Adattester: Secure online mobile advertisement attestation using trustzone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2015), MobiSys '15, ACM, pp. 75–88.

[16] LI, X., HU, H., BAI, G., JIA, Y., LIANG, Z., AND SAXENA, P. DroidVault: A trusted data vault for android devices. In *2014 19th International Conference on Engineering of Complex Computer Systems* (Aug. 2014), IEEE.

[17] LIU, D., AND COX, L. P. VeriUI. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications - HotMobile'14* (2014), ACM Press.

[18] MACHIRY, A., GUSTAFSON, E., SPENSKY, C., SALLS, C., STEPHENS, N., WANG, R., BIANCHI, A., CHOE, Y. R., KRUEGEL, C., AND VIGNA, G. BOOMERANG: Exploiting the semantic gap in trusted execution environments. In *Proceedings 2017 Network and Distributed System Security Symposium* (2017), Internet Society.

[19] MITRE. Common weakness enumeration. https://cwe.mitre.org/.

[20] MONSHIZADEH, M., NALDURG, P., AND VENKATAKRISHNAN, V. N. Mace: Detecting privilege escalation vulnerabilities in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 690–701.

[21] MOR, R. X session management library version 1.0. https://www.x.org/releases/X11R7.7/doc/libSM/SMlib.html, 1993.

[22] PARNO, B. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd Conference on Hot Topics in Security* (USA, 2008), HOTSEC'08, USENIX Association.

[23] RESHETOVA, E., BONAZZI, F., AND ASOKAN, N. Selint: An seandroid policy analysis tool. *Proceedings of the 3rd International Conference on Information Systems Security and Privacy* (2017).

[24] RESHETOVA, E., BONAZZI, F., NYMAN, T., BORGAONKAR, R., AND ASOKAN, N. Characterizing seandroid policies in the wild. *CoRR abs/1510.05497* (2015).

[25] ROSENBERG, D. Reflections on Trusting TrustZone. *BlackHat USA* (2014).

[26] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[27] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium* (2015).

[28] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Sok: (state of) the art of war: Offensive techniques in binary analysis.

[29] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium* (2016).

[30] SUN, H., SUN, K., WANG, Y., AND JING, J. TrustOTP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS'15* (2015), ACM Press.

[31] SUN, H., SUN, K., WANG, Y., JING, J., AND JAJODIA, S. TrustDump: Reliable memory acquisition on smartphones. In *Computer Security - ESORICS 2014*. Springer International Publishing, 2014, pp. 202–218.

[32] TRUSTED COMPUTING GROUP. Trusted platform module main specification. https://trustedcomputinggroup.org/resource/tpm-main-specification/, 2007.

[33] ZHENG, X., YANG, L., MA, J., SHI, G., AND MENG, D. TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms. In *2016 IEEE Symposium on Computers and Communication (ISCC)* (June 2016), IEEE.

[34] ZHOU, X., LEE, Y., ZHANG, N., NAVEED, M., AND WANG, X. The peril of fragmentation: Security hazards in android device driver customizations. *Proceedings - IEEE Symposium on Security and Privacy* (11 2014), 409–423.