TEEzz: Fuzzing Trusted Applications on COTS Android Devices

Marcel Busch EPFL
Aravind Machiry Purdue University
Chad Spensky Allthenticate
Giovanni Vigna UC Santa Barbara
Christopher Kruegel UC Santa Barbara
Mathias Payer EPFL

***Abstract*—Security and privacy-sensitive smartphone applications use trusted execution environments (TEEs) to protect sensitive operations from malicious code. By design, TEEs have privileged access to the entire system but expose little to no insight into their inner workings. Moreover, real-world TEEs enforce strict format and protocol interactions when communicating with trusted applications (TAs), which prohibits effective automated testing.**

TEEzz is the first TEE-aware fuzzing framework capable of effectively fuzzing TAs *in situ* on production smartphones, i.e., the TA runs in the encrypted and protected TEE and the fuzzer may only observe interactions with the TA but has no control over the TA's code or data. Unlike traditional fuzzing techniques, which monitor the execution of a program being fuzzed and view its memory after a crash, TEEzz only requires a limited view of the target. TEEzz overcomes key limitations of TEE fuzzing (e.g., lack of visibility into the executed TAs, proprietary exchange formats, and value dependencies of interactions) by automatically attempting to infer the field types and message dependencies of the TA API through its interactions, designing state- and type-aware fuzzing mutators, and creating an in situ, on-device fuzzer.

Due to the limited availability of systematic fuzzing research for TAs on commercial-off-the-shelf (COTS) Android devices, we extensively examine existing solutions, explore their limitations, and demonstrate how TEEzz improves the state-of-the-art. First, we show that general-purpose kernel driver fuzzers are ineffective for fuzzing TAs. Then, we establish a baseline for fuzzing TAs using a ground-truth experiment. We show that TEEzz outperforms other blackbox fuzzers, can improve greybox approaches (if TAs source code is available), and even outperforms greybox approaches for stateful targets. We found 13 previously unknown bugs in the latest versions of OPTEE TAs in total, out of which TEEzz is the only fuzzer to trigger three. We also ran TEEzz on popular phones and found 40 unique bugs for which one CVE was assigned so far.

***Index Terms*—Fuzzing, Android, TEE, ARM TrustZone**

I. Introduction

Smartphones operate on private user data and perform sensitive functionality, e.g., financial transactions [31], user authentication [76], or handling digital rights management (DRM) protected media [30]. To defend against various application- and kernel-level exploits [21], [70], [84] applications leverage TEEs [24] (e.g., ARM TrustZone (TZ) [7]) as an additional hardware-based defense. TEEs enforce the integrity and confidentiality of their applications. Partially due to recent research that demonstrated the usefulness of TEE applications [25], [36], [42], [56], [87], [89], called TAs, their number, as well as their complexity, is steadily increasing, leading to more TA-based vulnerabilities [15], [16], [34]. Unlike regular applications, where the vulnerability affects only the application, a vulnerability in a TA compromises the security of the entire system [88], potentially even the secure boot process [66].

While the security of these TAs is foundational to the security of the device, performing effective testing (e.g., fuzzing) remains an open challenge. Smartphones ship with the trusted OS (tOS) and numerous pre-installed TAs, prohibiting the normal world (e.g., Android) from inspecting their code at runtime. TA interactions are stateful and use complex proprietary message formats [39]. The entities in the secure world (TEE and TAs) are often encrypted and get decrypted in secure memory at runtime, prohibiting the use of static analysis-based vulnerability detection techniques. Dynamic analysis, i.e., fuzzing, is an effective alternative.

There are two principled approaches for fuzzing TAs: *rehosting through emulation* or *on-device instrumentation*.

Rehosting the TEE in an emulated environment overcomes the inaccessibility of the TEE's internal state. PartEmu [39] rehosts Samsung's proprietary TEE software stacks. They rehost the tOS and its TAs, to an emulated system-on-a-chip (SoC), gaining unrestricted access to the TEE's internal state. Limitations to this approach are (1) the reverse engineering and implementation effort for emulated software and hardware components, (2) the inaccuracy of these implementations, (3) the lack of public data sheets, and (4) industry involvement leading to non-disclosure agreements for existing solutions. Especially the last limitation deserves further emphasis. PartEmu is the only existing rehosting solution targeting multiple TEEs. The prototype validates the feasibility of rehosting proprietary software stacks deployed on Samsung devices and is not publicly available.

The second approach, on-device fuzzing, mitigates these limitations and inaccuracies of emulation approaches. However, it lacks access to the TEE's internal state and must fall back to blackbox fuzzing techniques. Unlike typical fuzzing techniques, which can analyze the binary, system memory, and executed instructions, an on-device TEE fuzzer must infer bugs from a far more limited view of the execution. Interactions with TAs happen through a vendor-provided interface (e.g., a driver [1], [3], [74]) in the rich OS (rOS), which ultimately generates an secure monitor call (SMC) to communicate with the secure world. The only observable execution effects are returned data (e.g., return values) and the status of the TA.

The gateway to the TEE is usually a TEE-specific kernel driver. While kernel fuzzers like Syzkaller [32] or DIFUZE [20] are capable of fuzzing kernel drivers, these tools fail to *capture the message semantics* required to interact with TAs, rendering them ineffective when targeting TEEs. Thus, novel techniques are required to find and fix bugs in these security-critical applications.

We present TEEzz, a fuzzing framework for TAs running on commercial smartphones. TEEzz targets three popular TEE implementations: the Qualcomm Secure Execution Environment (QSEE) [62], used on many phones including the popular Nexus and Pixel series; TrustedCore (TC) [79], found on Huawei devices; and the Open Portable Trusted Execution Environment (OPTEE) [51], the de-facto reference implementation for TZ-based TEEs. The analysis first identifies the TAs within the TEE and then manually triggers interactions with them. During these interactions, TEEzz records the data passed both into and out of the TEE to automatically reconstruct the message format and complex message and value dependencies. Lastly, this message format, along with the dependencies of the interaction (i.e., generating a cryptographic key before it is used for encryption), are fed into our fuzzer. The fuzzer explores the TA while continuously checking for liveness and monitoring for crashes.

TEEzz necessitates diverse contributions. First, the complex and proprietary data structures of TAs require fuzzed inputs to be well-formed, or else the parsing logic in the tOS will reject them. Thus, we designed TEEzz as a *mutation-based* fuzzer that operates on type- and state-aware seeds generated from legitimate interactions with TAs. To infer the necessary knowledge of the API, we design an inference mechanism that maps high-level abstractions to low-level messages used to communicate with the TA. TEEzz *automatically* generates memory introspection logic for each parameter type of the exposed interface and then abstracts the interaction protocol from the recorded traces. At runtime, we dynamically instrument this interface, parse the values corresponding to each type on-the-fly from memory, and save the type-aware token sequence to disk. The observed type- and state-aware interactions become the specification for efficient mutation.

Second, we *automatically* generate type-aware mutators for the enriched seeds. We convert the type definitions used by TA-facing interfaces into type-aware mutator plugins for TEEzz's mutator engine. While fuzzing, TEEzz leverages these type-specific mutators to manipulate input tokens.

Third, many TAs are stateful, and value dependencies between invocations need to be resolved, i.e., a value returned from one invocation *must* be used as an input for a future invocation. Leveraging the previously recorded type-enriched interaction sequences, which include ingoing and outgoing data, TEEzz employs a novel value dependency inference technique to add state-awareness to its seeds.

Finally, TEEzz is the first end-to-end solution capable of continuously fuzzing TAs on COTS Android devices. Although we use known techniques such as dynamic binary instrumentation-based introspection (DBII) and semantic reconstruction, the novelty of TEEzz stems in solving technical challenges (Section IV) to apply these techniques to a restricted environment of a COTS device with no direct access to secure world entities. In addition, TEEzz features an extensible type-aware mutation engine, a state-aware fuzzing paradigm that considers entire interaction sequences as seeds and resolves interaction dependencies during runtime. Further, it supports state reset mechanisms to deterministically build up TA state to facilitate the reproduction of crashes.

We evaluated TEEzz's performance in terms of coverage and bug-finding capabilities in a ground-truth experiment. For this purpose, we extended the OPTEE platform with (1) permanently shared memory between client applications (CAs) and TAs, (2) TA instrumentation to populate the coverage bitmap, (3) TA instrumentation to collect program counters during post-processing, and (4) support for TA constructors to initialize the instrumentation. Due to the non-availability of related fuzzers, we truthfully replicate the state-of-the-art based on AFL++ and compare TEEzz against three TA-aware AFL++ variants. Our results show that TEEzz finds bugs that are unreachable for existing fuzzers. In fact, TEEzz was the only fuzzer capable of finding three previously unknown bugs in OPTEE TAs. Further, we tested TEEzz on 18 TAs covering four popular Google and Huawei phones. TEEzz successfully generated enriched seeds, inferred interaction dependencies, and fuzzed each TA. Across these proprietary targets, TEEzz successfully found 40 unique bugs that we responsibly reported to the corresponding vendors. One CVE (CVE-2019-10561) was assigned so far, and we await further replies. Some of these crashes force the phone into a factory reset—wiping all user data— to resume normal functionality. In contrast, others allowed us to extract protected cryptographic keys from the TEE, a stepping stone to launch brute-force attacks against a device's disk encryption.

In summary, our contributions are as follows:

- We developed TEEzz, available at https://github.com/HexHive/teezz-fuzzer, the first end-to-end automated fuzzing framework capable of fuzzing TAs on commercially available smartphones;
- an automated, dynamic-analysis-based technique for inferring field types of messages, as well as their dependencies, to facilitate type-aware fuzzing of stateful TAs;
- type and state-aware fuzzing mutators that leverage the message and dependency information inferred from analyzing the interactions with the TAs; and
- a thorough evaluation of TEEzz against other state-of-the-art fuzzing techniques on production TAs.

II. Background

In this section, we present the relevant background for ARM TrustZone and its usage on the Android platform. Then, we cover the fuzzing concepts necessary for our research. These concepts include type-awareness and state-awareness.

a) TEEs on the Android Platform: The vast majority of smartphones uses modern ARMv8-based SoCs that leverage TZ to provide a hardware-supported TEE for security-critical functionality. Figure 1 shows the privilege levels (i.e., exception levels) of ARMv8 TrustZone and the location of the complex feature-rich code (e.g., Android) in the "normal" world,

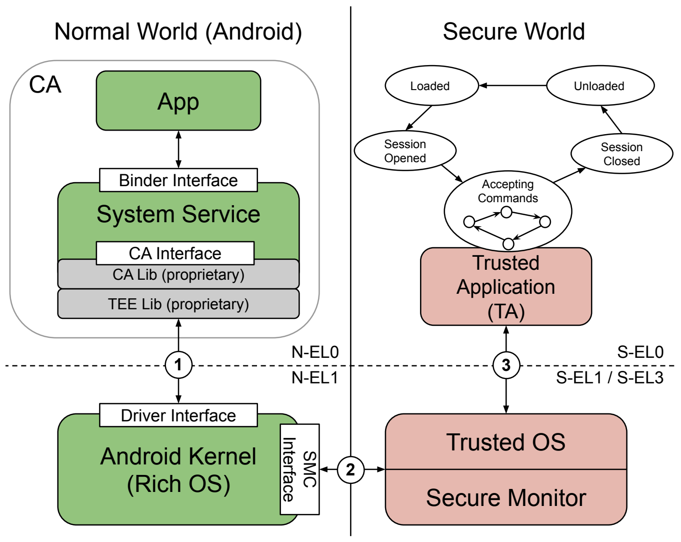


Fig. 1. Communication flow between a CA and a TA in Android. For simplicity, we combine the app that performs the request to the TA and the service that marshals and dispatches the request and refer to them as CA.

and the sensitive code (e.g., the management of authentication keys or DRM media) in the "secure" world. Software in each of these worlds is referred to as either *untrusted* in case of the normal world (i.e., the rOS on N-EL1 and its CAs on N-EL0), or *trusted* in case of the secure world (i.e., the tOS on S-EL1 and its TAs on S-EL0). The secure monitor on the highest privilege level (EL3) is responsible for switching a central processing unit (CPU) core from one world to the other; therefore, both OSes have to call into the secure monitor to interact with the respective other world.

The security-critical features in TEEs are usually implemented through dynamically loadable TAs. On Android, the interaction with TAs is primarily initiated by system services that expose their functionality to regular apps via the Android Framework application program interface (API) (i.e., Binder-based inter-process communication (IPC)). Figure 1 depicts a simplified but typical communication paradigm between CAs and TAs on the Android platform.

An Android app uses system services to access hardware features (e.g., microphones, cameras, and sensors). Many of these system services leverage the capabilities provided by a TEE in the background (e.g., `keystore`, `gatekeeperd`, and `fingerprintd`). Thus, a TEE is treated as a hardware device in Android's architecture. To have common abstractions across vendors and their hardware-specific implementations, Android defines hardware abstraction layers (HALs) [33] for all system services dealing with hardware. Regarding the TEE, each system service uses a proprietary CA library (CAlib), which exposes the common HAL interface and implements the TA-specific marshaling of the request. Additionally, this CAlib usually invokes a further proprietary TEE library that implements abstractions for interacting with the vendor-specific TEE driver (1 in Figure 1).

The TEE driver, usually invoked using an `ioctl` [20] system call, is responsible for forwarding the request to the tOS (we use the terms driver interface and `ioctl` interface interchangeably). Since the rOS and tOS have different virtual address spaces (i.e., they use separate page tables), the rOS needs to provide the request in a coordinated shared memory region before it calls the privileged SMC instruction (2 in Figure 1) to switch worlds. Once the message is received by the tOS, the tOS forwards the message to the corresponding TA (3 in Figure 1).

b) Type-Awareness: Naïve mutation primitives consist of random modifications, appending data, and removing data. Other mutations may depend on the type of the data. For example, an `enum` type with a set of possible values could be mutated to ensure that it will have one of the possible values with high probability. The mutations aim to reduce the range of interesting values for a mutation to trim the search space for inputs overall. We refer to this principle's extension from primitive to complex data types as *type-awareness*.

Automated type-aware fuzzers such as DIFUZE [20] and HFL [44] require source code for the generation of their fuzzing templates. They thus cannot be effectively applied to TAs as they are available only in binary form and often encrypted (e.g., Huawei).

c) State-Awareness: Many applications and libraries have internal finite state machines. For instance, a file cannot be read from or written to without first opening it. This dependency is enforced programmatically by requiring that a valid file descriptor be passed to the `read` and `write` function. This descriptor can be obtained by opening a file through `open`. We say that these functions have a *value dependency* and a consumer of this API needs to be *state-aware* [35], i.e., use functions in an expected order and resolve value dependencies, e.g., `open` before `read` and correctly passing the file descriptor.

Fuzzers that have an automated approach to state-awareness either require source code (FuzzGen [40] and HFL [44]), or fine-grained logs (IMF [38] and MoonShine [59]), both of which are hard or rather impossible to get on COTS devices with a proprietary TEE.

III. Interacting with TAs

One of the fundamental requirements to fuzz TAs is the ability to execute them by providing interesting inputs. This section discusses different ways to interact with TAs along with their tradeoffs, thereby providing the rationale for choices made in TEEzz.

A. Executing TAs

TAs require the corresponding TEE, whose execution in turn requires the specific hardware (see Section II). The two straightforward choices are executing the TAs on-device or rehosting the TAs [37], i.e., executing a hardware-dependent program in an emulated environment.

a) On-device execution: Executing a TA directly on the original (or at least compatible) hardware along with the corresponding TEE is the most precise way to capture the runtime behavior of TAs. In cases like Huawei, where the TEE firmware is encrypted, on-device execution is the only way to execute corresponding TAs.

No Introspection: For the majority of available devices, the original equipment manufacturer (OEM) restricts modifications to the TEE. [1] This prevents the analyst from introspecting the TA on the device as it requires modifying or instrumenting the TEE. TEE software exploits or hardware attacks to gain code execution within the TEE are out of scope. These options would enable more powerful introspection capabilities but are challenging to carry out and not universally available.

b) Rehosting: These techniques enable the execution of a hardware-dependent program, such as a TEE, in an emulated environment. Samsung's not-publicly-available PartEmu [39] rehosts proprietary TEE software stacks, consisting of the tOS and TAs, to an emulated environment based on QEMU [10] and Panda [23].

Incompleteness: Existing approaches suffer from inaccuracy of emulated hardware and software. Chip designs and corresponding data sheets are often unavailable publicly, and accurately emulating hardware without this information requires non-trivial reverse-engineering. Although PartEmu emulates peripherals such as the TrustZone address-space controllers (TZASC) and protection controllers (TZPC), chip manufacturers add custom components that are difficult to emulate. For example, Qualcomm chips contain a proprietary *eXtended Protection Unit* "XPU" enforcing dynamic memory protection of shared buffers between the secure world and the normal world, making it an interesting fuzzing target. Furthermore, PartEmu does not model certain proprietary and necessary peripherals like the fingerprint sensor, ARM CryptoCell [91], or face identification hardware. This lack of *perfect* emulation results in a lack of fuzzing accuracy causing false positives and negatives. For an emulator, practically feasible emulation often involves sacrificing some hardware accuracy.

B. Input Injection

Due to restrictions placed on the secure world (i.e., the inability to modify or debug code), inputs to TAs must necessarily originate from the normal world. Within the normal world, fuzzing inputs can be injected into the system (with the intent of reaching TAs) at several locations, each having accompanying tradeoffs:

Client Application (CA): Requests to TAs typically originate from a CA. Thus, directly exercising the CA generates valid requests to the corresponding TAs. However, the input to the CA has *little or no control* over the values that are ultimately sent to the TAs. As the vendor-specific proprietary TEE client library (see Figure 1) sanitizes the forwarded input, this interface is ineffective for fuzzing.

[1]Note that bootloader unlocking allows deploying custom software in N-EL1 and N-EL0 (see Figure 1) but the TEE is off limits.

Driver Interface (① in Figure 1): There are already techniques, specifically Syzkaller [32] and DIFUZE [20], that can provide fuzz inputs at the *driver interface*. However, they fail to generate valid requests through the SMC interface (② in Figure 1) since they are neither aware of the input format accepted by TAs nor their internal finite state machines.

SMC Interface (② in Figure 1): While it is straightforward to fuzz at the SMC call interface, the generated inputs are likely to fail to reach the TA since they most probably do not adhere to the TA-specific protocol or data format.

This multi-interface invocation of a TA presents a tension between the high-fidelity, low-control interface of the CA and the low-fidelity, raw interface of the SMC call.

As fuzzed input injection moves farther away from the tOS, the amount of control over the fuzzed data decreases while the likelihood of the data ultimately reaching the targeted TA increases. For example, the SMC interface permits complete control over the data but requires an accurate reconstruction of the entire communication protocol. On the contrary, the CA provides limited ability to modify the handled data, as numerous checks and mutations are performed, but creates valid packets for the TA.

C. Interaction with TAs in TEEzz

To prevent inaccuracies introduced by software and hardware emulation, we interact by executing TAs on the device. We handle the lack of introspection by using a coarse-grained feedback mechanism, as explained in Section VI, to determine the state of TAs.

We inject fuzzed inputs through the driver interface, as it gives sufficient control over the data sent to a TA and makes our input injection mechanism independent of the SMC calling convention (as determined by the SoC). However, as mentioned before, generating valid requests to a TA through the driver interface requires us to know the input format expected by the TA, along with the status of its internal finite state machine. We will discuss these challenges and how TEEzz handles them in the next section.

IV. Challenges and TEEzz's Approach

Effectively sending arbitrary but well-formatted data to a TA through the driver interface requires (i) *Type Awareness*, an understanding of the structure and type of data accepted by the TA; and (ii) *API State-Awareness*, an understanding of the interdependencies between the data sent to a TA across multiple requests. For instance, a `sign` request to the `keymaster` TA has to contain information about the key to be used, which has been generated or imported by a preceding request.

A. Challenge #1: Structure Recovery (Type-Awareness)

The data expected by TAs through the TEE driver interface (③ in Figure 1) is proprietary and specific to the TA. But, the structured and typed data from the upper layers (i.e., the CAlib) enter the driver interface (① in Figure 1) in a serialized and untyped format.

To effectively send input to a TA, the format of the expected input and the marshaling method expected by the driver interface must be known. We cannot analyze the entities in the secure world (right half of Figure 1) as the firmware is mostly encrypted with device keys. Even in the normal world, recovering the structure and type of data expected by a TA has the following two sub-challenges:

Java Native Interface (JNI): As shown in Figure 1, in the normal world, the data from the app to the TA goes through various layers before reaching the driver interface. Each layer marshals the incoming data in a specific format before sending it to the next layer. We need to track the data flow to understand which parts of the data sent to the driver interface are needed by the TA. However, transitions between these layers are complex, such as the app calling from a Java context into a native library using JNI [50] and the inter-process communication with the system service using Binder [69]. But, precise static tracking of information flow across the JNI layer and process boundaries is still an open research problem [5], [83].

Type and structure recovery on binaries: Few of these layers involved in the data flow from the app to the driver interface contain potentially closed-source, stripped, and obfuscated binary code, e.g., CAlib. These libraries contain essential information about the structure and type of the data accepted by TAs. Recovering this information requires precise structure and type inference at the binary level, which is also an open research problem [47], [57].

TEEzz's Approach: TEEzz recovers formats expected by TAs using a novel combination of dynamic binary instrumentation (DBI), multi-interface message recording, and semantic deduction [45] (i.e., automatically bridging the semantic gap between the low-level interface and high-level API). As we will explain in Section V-C, TEEzz exploits the availability of types from the CAlib layer interfacing the proprietary vendor code to reconstruct the format carried within the messages crossing the driver interface.

B. Challenge #2: API Statefulness (State-Awareness)

As mentioned before, interactions with TAs are stateful [39], [52], [88]. Existing techniques, such as PartEmu [39], reset (or restart) the state of TAs after each request and hence do not account for the accumulation of state and fail to explore the state machine of TAs.

An effective fuzzer must account for the TEE implementation-specific state machine for loading TAs and establishing sessions as depicted in Figure 1. Moreover, it also has to account for the TA-specific state machine. Any fuzzing technique must adhere to the proper TA-specific protocol to go beyond shallow bugs.

TEEzz's Approach: As mentioned before, we use DBI to record the messages exchanged with TAs. In addition, we also preserve the order of these recorded messages. Using this ordered list of messages and their corresponding data, TEEzz attempts to infer the API model of TAs for a given message sequence (i.e., the appropriate fields and invocation order) by analyzing data dependencies between messages (as explained in Section V-C).

V. System Design

TEEzz is designed as a pluggable framework to ensure its portability and extensibility as existing TEEs continue to evolve, and new TEE vendors enter the landscape. For each platform, TEEzz must be initialized to ensure effective fuzzing. Figure 2 provides a high-level overview of the workflow of our system. The first step is to identify the CAlibs interacting with TAs that are installed within a given TEE, obtain their interfaces, and create CAlib consumers, if none exist, to trigger interactions (Section V-A). Next, TEEzz automatically generates DBII recorders and type-aware mutators from these CAlib interfaces (Section V-B). Then, TEEzz leverages these recorders to carry out a multi-interface interaction capturing. Having the same interaction recordings on a semantically-rich high-level interface and a semantically-poor low-level interface serves us to propagate types to the lower level where we have more flexibility to inject mutated inputs for fuzzing. Further, based on the recordings of entire interaction sequences, TEEzz attempts to infer value dependencies between interactions, resulting in type- and state-aware TA interactions that can be used as seeds by the *fuzzing engine* (Section V-C). Finally, TEEzz employs these enriched seeds and the previously generated CA-specific mutators to fuzz TAs on COTS Android devices (Section V-D).

A. CA Identification

CAs are built to interact with TAs, and they are part of the normal world execution context that we fully control. TEEzz is based on the idea of extracting the knowledge of CAs about the protocol and message formats needed to interact with TAs. We chose a dynamic approach to capture this knowledge by recording interactions (Section V-C) and therefore need a way to trigger these interactions.

From our observation, we can find four usage scenarios for CAlibs on COTS Android devices.

- **Android Open Source Project (AOSP) System Service.** These are services usually present on all Android devices, and Google specifies the open source CAlib interface. The well-known `keystored`, `gatekeeperd`, `fingerprintd`, and `mediaserver` are examples for these services.
- **Vendor System Service.** These are vendor-specific services, and the CAlib interface is not publicly available. CAs for `secure storage` or `anti theft` features are examples for these services.
- **Unused CA.** Some vendors deploy CAlibs to their devices that are not used by any component. In this case, the corresponding TAs are present and fully functional. They are just not used.
- **Non-Existent CA.** This scenario applies to the situation where a TA exists without having a corresponding CA deployed on the device. Since TEEzz's approach requires CAs, this scenario is out of scope.

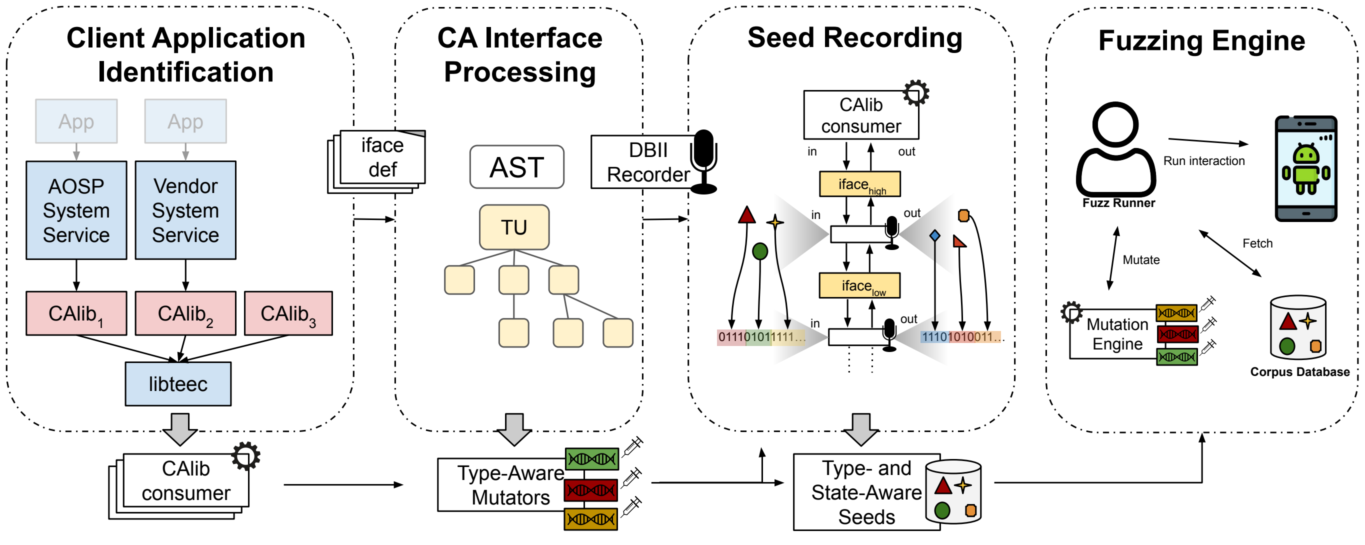


Fig. 2. The TEEzz approach. First, TEEzz identifies the CAlibs capable of communicating with their corresponding TAs and the consumers of this library. Then, it automatically generates DBII recorders and type-aware mutators from the interface definitions of these CAlibs. Next, the interactions are simultaneously recorded at two interfaces, (1) the high-level CA interface and (2) the low-level TEE driver interface. This recording approach allows TEEzz to propagate types and inter-interaction dependencies to the high-control driver interface resulting in type- and state-aware fuzzing seeds. Finally, TEEzz employs these enriched seeds and the type-aware mutators to fuzz TAs running on COTS Android devices.

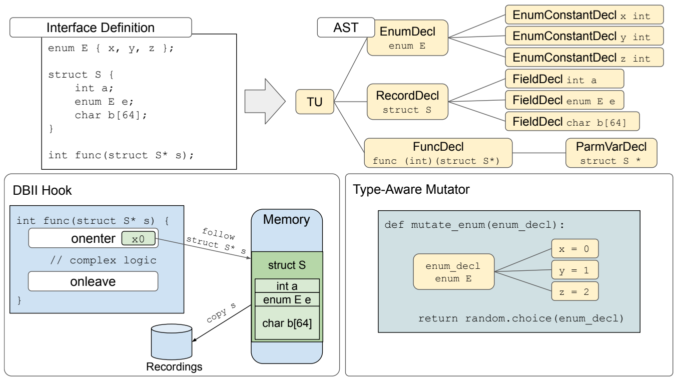
TEEzz identifies CAs based on our observation that all vendors use a single TEE client library that serves as an abstraction to interact with the TEE driver (e.g., `libQSEEComAPI.so` on Qualcomm, `libteec.so` on Huawei, `libmcclient.so` on Samsung). By generating the dependency tree using a static analysis that recursively traverses all dependent objects, we identify all CAs and, if applicable, their corresponding services.

We obtain the CAlib interface for all CAlibs used in AOSP system services from the AOSP repository. For CAlibs used in vendor system services and unused CAlibs an analyst has to manually spend the effort to obtain interface definitions and trigger interactions with TAs.

The automatic extraction of these interface definitions and automatic unit-test generation are open research problems. For example, FUDGE [8] and FuzzGen [41] deal with this problem. Both approaches rely on source code and the existence of library consumers. These prerequisites are not given regarding the problem of fuzzing TAs on COTS devices. Thus, we consider the automation of this step as an orthogonal research problem and opted for a manual approach. From our experience, the CA layer is usually designed for interaction, meaning that the affected libraries have exported symbols. Furthermore, we did not encounter any obfuscation techniques. For vendor system services, we can trigger interactions and have a dynamic component for the interface analysis.

B. CA Interface Processing

Given a CA interface definition in C or C++, TEEzz automatically generates DBII recorders and type-aware mutators from the abstract syntax tree (AST) representation of the interface.



```
enum E { x, y, z };

struct S {
    int a;
    enum E e;
    char b[64];
}

int func(struct S* s);
```

```
int func(struct S* s) {
    // complex logic
}
```

```
def mutate_enum(enum_decl):
    return random.choice(enum_decl)
```

Fig. 3. TEEzz's CA interface processing. TEEzz automatically generates DBII hooks and type-aware mutators from the AST of the CA's interface definition.

a) DBII Recorder Generation: Memory introspection is a well-known technique used in the context of virtual machines. Virtual machine introspection describes the process of a host that reads and parses raw bytes of a guest to reconstruct the meaning of values. For TEEzz, we leverage this technique to record each incoming and outgoing parameter of the CAlib interface according to its type using DBI. For some predefined data types, this technique might be known from the commonly known `strace` tool. `strace` can parse and print information about parameter types of system calls while tracing. The parsing logic for these types is hard-coded and limited to a few widely used complex types.

TEEzz, similar to `strace`, is capable of parsing primitive types, like `char`, `short`, and `int`. In contrast to `strace`, TEEzz can automatically generate the parsing logic for complex types from the definition of this type. Consider the function `func()` in Figure 3. It accepts the parameter `s` of type `struct S*`. TEEzz generates an `onenter` and an `onleave` hook for each function of a given interface. These hooks, triggered on function entry (`onenter`) and exit (`onleave`), contain the parsing logic to retrieve the complete parameter from memory according to its type. For the `struct S*` type, this logic would dereference the pointer passed as the first parameter and read three chunks of memory: four bytes for `int a`, four bytes for `enum E e`, and 64 bytes for `char b[64]`.

Our approach misses to accurately figure out the size of particular objects, e.g., `void*` and the size of buffers pointed to by an `int*` parameter. In these cases, we use specific heuristics similar to the prior work [53].

For interaction recording, the `onenter` hook is responsible for recording all incoming parameters, and the `onleave` hook is responsible for recording all outgoing parameters and the function's return value.

b) Type-Aware Mutator Generation: From the CAlib interface, we know which types are passed to the lower layers and eventually to the TA. The intuition for the generation of type-aware mutators is that during fuzzing, we want to create semi-valid inputs, and the knowledge of types helps to reduce mutations that likely lead to invalid inputs.

TEEzz converts the type definitions known from the CAlib interface into type-aware mutators. Given the token sequence of an enriched seed, TEEzz can choose from several type-specific mutations. In Figure 3, we illustrate a mutator for an `enum`. For example, the constants of the enumeration `enum E { x, y, z };` are encoded using four bytes in C/C++. A naive mutator would flip random bits of these four bytes and disregard the fact that the type already indicates that only `x=0`, `y=1`, and `z=2` are valid values. This mutation usually leads to many wasted cycles because the parsing component immediately rejects the input. Using the mutators generated by TEEzz, mutations on this enumeration take the value space of these four bytes into account and assign the values indicated by the type with a higher probability.

C. Seed Recording

In this stage of TEEzz, we obtain type- and state-aware interaction sequences for each TA. We capture TA interactions at two interfaces, the semantically-rich and low-control CA interface and the semantically-poor and high-control TEE driver interface. This multi-interface interaction recording allows us to propagate the types observed at the CA interface to the TEE driver interface. This lower-level interface is more appropriate for fuzzing because we have more control over the inputs eventually passed to TAs and can bypass the sanitization logic of the CA layer. The seed preprocessing is finalized by inferring the inter-interaction value dependencies. We can identify outgoing data utilized as input in a later interaction using the interaction recordings. The availability of types facilitates this inference.

The recording of seeds is a TA-specific process, meaning that each TA has its command handlers, expected parameters, and interaction sequences. The corresponding CA knows about these specific internals. Given a component that exercises the CA interface (i.e., an AOSP system service, a vendor system service, or a manually crafted CA driver), we can capture valid interaction sequences that can be used as seeds for fuzzing.

a) Multi-Interface Recording: Interaction recording can be carried out on several levels of abstraction. The CA layer exposes a semantically rich interface containing API calls that map directly to TA-implemented command handlers. Unfortunately, the CA layer also implements sanitization logic to reject invalid inputs early before they even reach the TEE.

The TEE driver layer exposes an interface that is primarily designed as a transport layer to pass serialized interactions back and forth between the rOS and the tOS. This interface allows for arbitrary manipulations of the inputs passed to TAs but does not give us any high-level semantics about types or dependencies between interactions.

Based on this observation, we decided for a multi-interface interaction recording to get the best of both worlds. Using DBI, we record input and output messages at the CA interface *and* the driver interface to correlate fields and type information later. Once a variety of messages have been recorded (e.g., every exposed function is exercised), TEEzz automatically generates type- and state-aware TA interaction seeds.

b) Type Propagation: TEEzz leverages the types and parameters for each function, which are automatically extracted from the interface definition (i.e., C- and C++-header files), to map the high-level semantic information to the raw values that were recorded in memory. Consider Figure 4, which shows the data recorded for the parameter `P` and `K` at a representative CAlib interface method, `f1()`. By also recording the input and output buffers at the low-level driver interface, TEEzz identifies matching subsequences and similarly maps the high-level semantic information. A linear scan yields the offsets of recorded CAlib parameters within the data sent to (and from) the driver interface. For example, the four byte `size_t` field `len` is easily observed at the beginning of the driver input buffer.

In addition to the CAlib parameter identification, TEEzz applies further structure reconstruction heuristics to also identify length fields, offsets, and strings (e.g., `off_t` the `0x10` offset to the `uint8_t[] k` at the beginning of the buffer). Each pair of recordings at the CAlib and driver layer results in a model for an enriched seed that TEEzz later uses for type-aware fuzzing.

c) Interaction Dependency Inference: TEEzz is also capable of stateful fuzzing. The function, `f1()`, in Figure 4 has an output parameter $\mathtt{K}_{\mathtt{out}}$ of type `keyblob_t**`. Note that in order for the subsequent function, `f2()`, to succeed, this return value, $\mathtt{K}_{\mathtt{out}}$, which is an output parameter to `f1()`, must be correctly generated and passed as the input parameter. To meet this stateful requirement, TEEzz tracks multiple calls and automatically identifies output parameters of a call that are used as input parameters in subsequent calls. With the raw buffer from the driver recorder and the recorded type-annotated parameters from the CAlib interface, TEEzz can infer the structure used at the driver interface. This type of

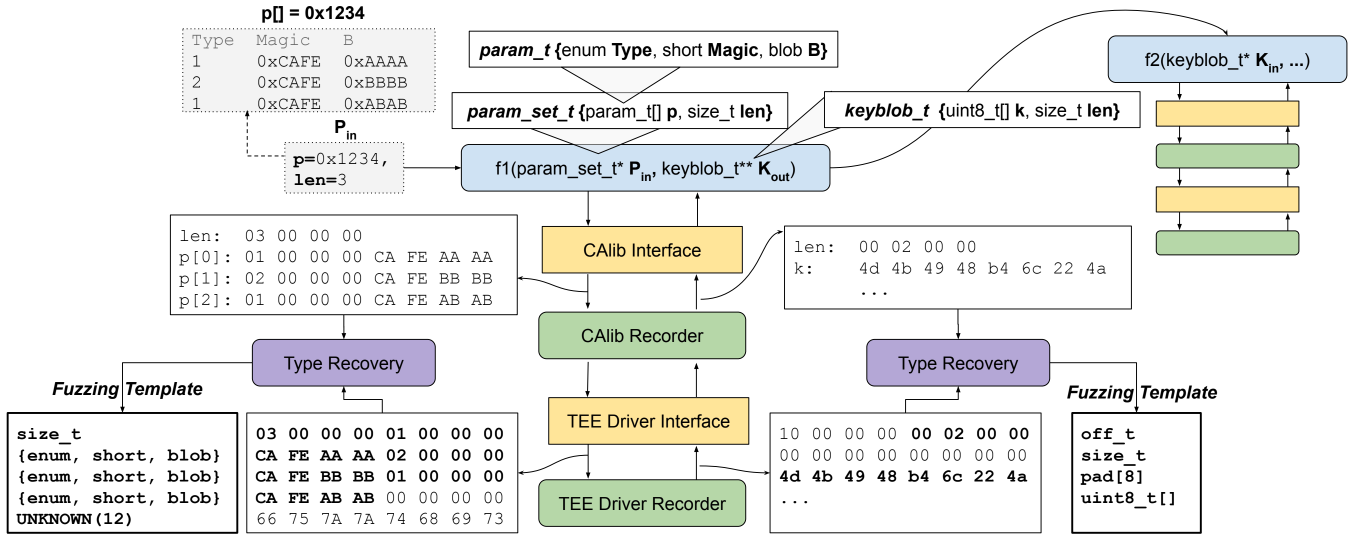


Fig. 4. An indicative function call sequence of a crypto API as exposed by the `keystore` system service. Our recorders (green) log the incoming (left) and outgoing (right) parameters of the CAlib and the driver interfaces (yellow). After recording, we match the typed data logged by the CAlib recorder with the raw buffers from the driver recorder using our type recovery (purple) to generate fuzzing enriched seeds for fuzzing. Beyond that, we recognize `K` being an output parameter of `f1()` and an input parameter of `f2()`, thus, accounting for the TA internal stateful API.

interaction is quite common in real-world TEE interactions. For example, to perform cryptographic operations using a key inside the TEE, the key must be generated first, with a known reference value.

Furthermore, we replay a recorded sequence multiple times to eliminate false positives and disable the resolution of individual value dependencies. If the dependent call still succeeds, we remove the value dependency.

Identifying these value dependencies is crucial to be later able to fuzz stateful APIs. Overall, performing type-aware and stateful fuzzing of TAs is an essential contribution of TEEzz.

D. Target Fuzzing

The actual fuzzing is carried out by a host component, called TEEzz^H, and a target component, called TEEzz^T. TEEzz^H selects the call sequences to be fuzzed according to the API model and performs type-aware mutation based on the types inferred in the type-aware model generation step. Then, it sends the mutated call sequence to TEEzz^T and evaluates the response. TEEzz^T is a proxy that forwards the inputs to the TEE driver and returns its responses.

VI. Implementation

In this section, we present the implementation details of TEEzz.

A. Hook-based Requests Recording

TEEzz is able to *automatically* inject hooks using dynamic binary instrumentation (DBI) by leveraging Frida [58], a popular and stable DBI framework. By specifying a function to be hooked (e.g., the `ioctl`-wrapper function within `libc` or functions of the CAlib), Frida allows for the injection of logic at the very beginning (`onenter`) and the very end (`onleave`) of a function. This technique allows TEEzz to record input and output parameters, as well as the return value, without corrupting the hooked function's logic.

Frida expects the recording logic executed by these hooks to be specified in JavaScript. The complexity of this logic is directly dependent on the complexity of the data structures to be recorded. For the TEE driver interface, the recording logic is relatively simple because the data structures are flat and consist primarily of length fields and their corresponding buffers (i.e., *uint8_t[]*). Since it is a one-time effort, we manually implemented simple cases like this for each targeted TEE driver interface.

However, consider the parameter `P` of type `param_set_t*` from our request recording illustration in Figure 4. The recording logic for this parameter needs to traverse a nested `struct`, and also account for the runtime value of `len` that indicates how many elements (not bytes) of type `param_t` are referenced by `p`. In reality, the structure of the CAlib interface parameter types is often complex. For complex cases like these, TEEzz implements a DBII recorder generator capable of generating complex introspection logic for Frida hooks based on an interface definition. This generator is written in Python and uses Python's `libclang` bindings to parse the header files describing the CAlib interfaces. After the parsing step, TEEzz emits Frida-hooks in JavaScript for each CAlib function containing proper introspection logic for parameters according to their types.

By using the AST, TEEzz's DBII recorder generator produces hooks that can traverse each parameter node of the AST down to its leaf nodes (primitive types) potentially following pointers through memory. We record the corresponding value from memory for each leaf node and annotate it with its respective type. A simplified example of this process is illustrated in Figure 3 where the leaf nodes of `func`'s parameter `struct S* s` are recorded.

In cases where explicit information about the relation of parameters or structure members is unavailable, TEEzz uses heuristics. For example, given a struct that encodes a buffer's length (i.e., one parameter being a buffer and the other one being its size), it may not be explicit from the header files that the second parameter describes the first parameter's size. Thus, TEEzz recognizes that there probably is a size associated to the first parameter and looks one parameter ahead to see if the current parameter name is a substring of the neighboring parameter suffixed with a size indicator (e.g., `buf` and `buf_{len,sz,length,size}`). If so, the size parameter's value is read from memory and used to record the buffer. This heuristic is implemented conservatively to prevent reading random memory content; we do not record the buffer if our heuristic does not succeed.

B. Type Recovery

To perform type recovery of the buffers recorded at the `ioctl` interface, TEEzz first matches parameters from the CAlib interface with byte sequences of the recorded buffers at the `ioctl` interface.

The implementation of the matching algorithm originates from the intuitive way of implementing serializers. Given a nested data structure, it is common to apply a depth-first traversal and store the data of neighboring AST leaf nodes next to each other. Thus, our matching algorithm traverses the recorded leaf node values of the high-level interface using a depth-first approach and tries to find identical values within the byte sequences captured from the low-level interface. If we encounter collisions, we apply a greedy strategy and prioritize the match with a higher number of bytes.

In the second stage, TEEzz applies heuristics to identify offset, size, and constant fields. TEEzz assumes that the size and offset fields will be four or eight bytes depending on the bitness of the targeted TA. For offset fields, TEEzz scans for any offset-sized byte sequence that points to data from the beginning of the buffer (alignments of 4, 8, or 16 bytes are considered). When an offset candidate points to the beginning of an already identified type from the previous step, it is a good indicator that our heuristic identified the offset correctly. For size fields, TEEzz scans for any size-sized byte sequence that matches the size of an already identified type from the previous step. Size fields are not only identified in terms of their number of bytes (i.e., a buffer length) but also by the number of elements in a list (i.e., structures stored in an array). If a value is identified that does not change across all observed recordings, we consider it a constant.

C. API Model Inference

Our recorded interaction sequences keep the chronological order of calls to the `ioctl` interface of the TEE driver to ensure that our fuzzed inputs satisfy the state and protocol requirements of the underlying TA. Furthermore, TEEzz identifies value dependencies between outputs and inputs. For example, one interaction produces a value consumed by a later interaction. To identify these dependencies, we scan through the typed fields of the output for each interaction and search for a matching typed field within the inputs of later interactions. Our current heuristic creates a value dependency if the type and value of a field in the output and input match. This approach allows TEEzz to satisfy dependencies from a request to a prior response, indicating which bytes from a given response need to be replayed verbatim in future requests, permitting the fuzzing of stateful TA APIs.

To eliminate false positives, we replay each sequence and systematically remove value dependencies while comparing the output behavior to the output of the original recording. If we can remove the dependency and achieve the same behavior, we find a false positive and do not consider this dependency for fuzzing.

D. Fuzzing

TEEzz's fuzzer is implemented in two parts: a program running on a machine connected to the phone that generates inputs and logs results, called TEEzz^H, and a stub that runs on the phone and works as a proxy to pass the fuzzed inputs to the appropriate interface, called TEEzz^T.

TEEzz^H orchestrates TEEzz^T to load/unload TAs and open/close sessions which is necessary to reach the core logic of any TA (see Figure 1). Given an opened session, TEEzz^T accepts inputs that it forwards to the TA and sends the output (the TA's response) back to TEEzz^H. TEEzz^T uses the TEE-specific API to interface the TEE driver, establish shared memory with the TEE, and send commands to TAs.

TEEzz^H observes the output behavior of the TA and initiates reboots via Android Debug Bridge (ADB) if necessary. TAs potentially accumulate state over time which is why we reboot the device after a configurable number n ($n = 500$, by default) of inputs sent to the target to reset the TA's state.

TEEzz's mutations are composed of type-aware mutations, bit-flips, and well-known mutations. For the type-aware mutators, TEEzz compiles all types used in the CAlib header files into Google's Protobuf interface description format. This translation is again based on traversing the AST of all type definitions leveraging Python and `libclang`. Given all available types specified in the Protobuf format, we use Google's Protobuf compiler for Python to access the AST types from our mutators. An example for a mutator for an `enum` declaration is illustrated in Figure 3. These type-aware mutators can be combined with the bit-flip and well-known mutators. The well-known mutations include edge cases for various width signed and unsigned integers, null-byte insertion for strings, increment and decrement operations, and the population with random data.

After mutating an input, TEEzz^H resolves value dependencies by propagating values from prior TA outputs to the current input. Figure 5 illustrates these value dependencies for the `keymaster` TA. We store these dependencies in a DAG-like data structure to resolve them while fuzzing.

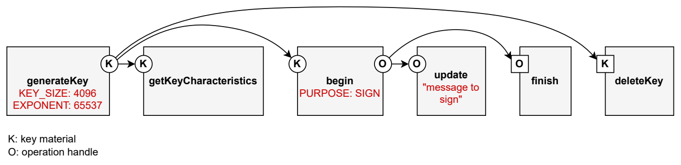


Fig. 5. Value dependencies of the `keymaster` TA API. TEEzz identifies these dependencies and stores them in a DAG-like data structure to resolve them while fuzzing.

TA crashes come in two flavors. TEEzz recognizes crashes that immediately reboot the phone and uses return codes from the TEE to evaluate target-specific crash conditions. When a potential crash is detected, a message is propagated to the runner, which persists the entire sequence of mutants. All the recorded seeds and mutants can later be checked for reproducibility using TEEzz.

VII. Evaluation

In this section, we perform a comprehensive evaluation of TEEzz and show the effectiveness of each of our techniques. First, we conduct ground-truth experiments exploring different fuzzers' capabilities to achieve coverage within TAs based on OPTEE (Sec VII-A). Second, we examine TEEzz's type-recovery and value-dependency identification capabilities (Section VII-B). Third, we evaluate TEEzz and its various techniques on COTS TAs (Sec VII-C). As a bonus, we include our unsuccessful attempt to use Kernel driver fuzzers to fuzz TAs in Section A and a comparative evaluation with the closed-source PartEmu [39] system in Section B of our appendix.

A. State-of-the-Art Ground-Truth Comparison

To the best of our knowledge, PartEmu [39] is the only fuzzer that targets TAs of COTS devices. Unfortunately, neither the PartEmu prototype nor the datasets used for its evaluation are publicly available.

Due to the lack of availability, a first-order comparison with PartEmu is impossible. Therefore, we chose to reimplement PartEmu's fuzzing approach as truthfully as possible and evaluate it on the available software. PartEmu's fuzzing module is based on AFL, TriforceAFL in particular. After reviewing this project and corresponding with the PartEmu authors, we integrated AFL++ into OPTEE (hence *optee-afl*) to establish a baseline for fuzzing TAs. For this purpose, we extended the OPTEE platform with (1) permanently shared memory between CAs and TAs, (2) source-code-based instrumentation for TAs to populate the AFL coverage bitmap during fuzzing and collect program counters during post-processing, and (3) support for TA constructors to initialize the instrumentation. For our evaluation, we run optee-afl with three different configurations against four OPTEE TAs within the QEMU emulator and compare the coverage and bugs found against our TEEzz:

Instrumented+noseed mode. This mode uses an instrumented target TA and follows the typical AFL fuzzing model using no seeds. This mode mimics the AFL PartEmu module and resembles the experiment carried out by Harrison et al. [39]. We consider this configuration to be our baseline.

Instrumented mode. This mode uses the same setup as in the previous configuration, but initializes the fuzzer with the seeds obtained through TEEzz's seed recording.

Multi-interaction mode. This mode goes one step further than the AFL PartEmu module due to its ability to fire multiple inputs against a target TA and therefore allows for building up state. We also initialize this configuration with seeds obtained through TEEzz.

Our dataset consists of four TAs available for OPTEE: `keymaster`, `gatekeeper`, `secure storage`, and `acipher`. For `keymaster` and `gatekeeper`, we recorded the seeds from the vendor test suite (VTS) binaries available from the AOSP. For `secure storage` and `acipher`, we recorded the seeds from the CA command-line executables which are part of the OPTEE project. We use the `keymaster` and `gatekeeper` TAs from tag 3.8.0 to leverage the known vulnerabilities from the publicly available security advisory. The two important metrics we experimentally evaluate are each fuzzer's coverage and capability to find bugs.

For the coverage experiment, we run each fuzzer ten times for 24 hours in a dockerized Ubuntu 20.04 setup on a machine featuring a Xeon Gold 5218 and 64GB of RAM. The coverage results from this experiment are shown in Figure 6.

TEEzz achieves a higher initial coverage compared to the other fuzzers for `keymaster` and `gatekeeper` due to its ability to resolve value dependencies between interactions. The API of `keymaster` contains calls that only succeed when a transitive value dependency across multiple calls is resolved. This property is also indicated by TEEzz's coverage, which is the only fuzzer that reaches 52% coverage in the best case. Compared to the second best coverage of 25% in the multi-interaction mode configuration, TEEzz covers more than double the code in this target.

For `gatekeeper`, the seeds provided by TEEzz significantly support the exploration of the instrumented and multi-interaction modes. The configuration without proper seeding, as carried out in PartEmu's evaluation [39] reaches a maximum coverage of 12% after 24h while the multi-interaction mode already starts at 38% and finishes at 58%.

From our experience, the `keymaster` and `gatekeeper` TAs of our dataset are similar to TAs found on production devices in terms of their complexity and interaction patterns. `acipher` and `secure storage` are example TAs of the OPTEE project and would likely not be used on production phones in their current implementation. For example, OPTEE's `secure storage` TA differs significantly from the `secure storage` TA found on Huawei devices in terms of its missing session management to hold a session-specific state. Furthermore, `acipher` only implements two static cryptographic operations that do not require complex input formats, as seen for TAs on COTS devices. Hence, it is not surprising that the instrumented mode, even without proper seeding, can significantly increase coverage and cover 69% of `acipher` and 74% of `secure storage`, respectively.

We found 13 previously unknown bugs in our OPTEE TA dataset. The AFL-based fuzzers combined could detect ten, and TEEzz was the only fuzzer that detected all 13 of these bugs. The three additional bugs found by our system required the target TA to build up state before they could be triggered. Thus TEEzz's state-aware seeds facilitate finding state-dependent bugs.

In summary, TEEzz's capability to resolve value dependencies across multiple interactions significantly improves the capabilities of modern fuzzers. Given that our coverage experiment featured coverage-guided fuzzer configurations that are commonly unavailable, whereas TEEzz is a blackbox fuzzer, it is a notable result that our fuzzer outperforms the competitors in realistic scenarios.

B. Accuracy of Type Recovery and State Awareness

In this section, we present an evaluation of TEEzz's type recovery and state awareness. For the presented experiments, we leverage the Google VTS for the `keymaster` subsystem and execute it on a HiKey620 development board featuring an Android/OPTEE deployment setup. The vast set of test cases included in the Google VTS allows us to measure TEEzz's type-recovery capabilities on many interactions and provides us with the ground truth for the behavior of sequences of interactions with the target. We use the latter to examine the accuracy of identified value dependencies (i.e., output values required as input values by later interactions).

a) Type Recovery: In TEEzz's seed recording stage, we record all inputs and outputs propagating through the high-level and low-level interface. Due to the DBII recorders targeting the high-level interface, we obtain the AST for every single parameter passed to a function and have the run-time values (associated with the AST's leaf nodes) readily available. This process allows us to map these leaf nodes of the high-level interface to the untyped byte sequences recorded from the low-level interface to recover types.

After executing and recording the entire Google VTS consisting of 102 sequences and 1,874 interactions with the `keymaster`, we map the input and output AST leaf nodes of the high-level interface to the untyped buffers of the low-level interfaces.

For the input, we were able to uniquely map $8,059$ out of $20,676$ AST leaf nodes. Our heuristics discarded $9,090$ leaf nodes since they consisted of all-zero byte sequences and are indistinguishable from padding in buffers. We could not map $3,527$ (17%) leaf nodes.

For the output, we uniquely mapped $14,821$ leaf nodes out of $33,614$ AST leaf nodes in total. $13,686$ nodes were discarded because of our all-zero byte sequence heuristic, and $5,107$ (15%) nodes could not be mapped automatically.

Besides uniquely mapped AST nodes, the coverage of the low-level buffers with types is an interesting accuracy metric. Table I shows the type coverage of low-level input and output buffers grouped by their high-level function and indicates the typical size of these buffers. For the eleven functions implemented by the `keymaster` TA, we can almost perfectly recover the types of inputs provided to six functions (97-100% recovery), perform well on another four functions (65-79% recovery), and only miss some of the types for one function ($\sim$30% recovery).

TABLE I
TYPE COVERAGE OF INPUT AND OUTPUT LOW-LEVEL BYTE SEQUENCES.

Function	Avg Cov Input		Avg Cov Output	
addRngEntropy	100.00%	(1,029.50)	0.00%	(4.00)
abort	100.00%	(8.00)	0.00%	(4.00)
deleteKey	100.00%	(1,269.24)	0.00%	(4.00)
getKeyCharacteristics	99.75%	(4,480.00)	58.06%	(124.00)
exportKey	98.90%	(1,424.65)	3.99%	(11.50)
begin	97.48%	(1,949.78)	30.07%	(22.14)
finish	79.18%	(203.26)	41.67%	(61.38)
importKey	71.83%	(195.53)	78.24%	(582.78)
update	69.17%	(55.14)	28.57%	(36.57)
attestKey	65.43%	(2,139.00)	0.00%	(5.33)
generateKey	30.28%	(75.67)	65.42%	(1,062.00)

b) State Awareness: Our seed recording stage (Section V-C) also tries to infer the value dependencies of interactions occurring in a recorded interaction sequence.

To assess the accuracy of identified value dependencies, we leverage Google's VTS for the `keymaster` TA and, in contrast to the other experiments, filter the tests for interactions that result in successful status codes to facilitate the creation of the ground truth for value dependencies. The `keymaster` target is representative due to the availability of many test cases and its stateful API. Figure 5 illustrates a typical value dependency graph of this target. In total, we obtain 95 sequences consisting of 728 interactions. Replaying all sequences, we manually mark dependent interactions and monitor if these interactions return successfully. If they return with an error code, we did not manage to identify and resolve the correct value dependencies.

Out of the 728 interactions in our dataset, we manually identified 565 as dependent on prior interactions' output values. In comparison to this time-consuming manual approach, TEEzz automatically resolved 457 (81%) correctly and only failed to resolve 108 (19%).

Regarding false positives, we replay a recorded sequence several times and disable the resolution of individual value dependencies. If the dependent call succeeds, we remove the value dependency and effectively eliminate false positives.

C. Fuzzing COTS TAs

Using TEEzz, we fuzzed 18 TAs that can be found on COTS Android devices. The detailed TAs are listed in Table II.

First, we prepared all devices by rooting them. Then, we obtained the dependency graph to the respective TEE client library (e.g., `libteec.so` and `libQSEEComAPI.so`). Based on each dependency graph, we decided for a subset of CAs to fuzz their corresponding TAs. We excluded TAs that require human interaction during their usage (i.e., TAs related

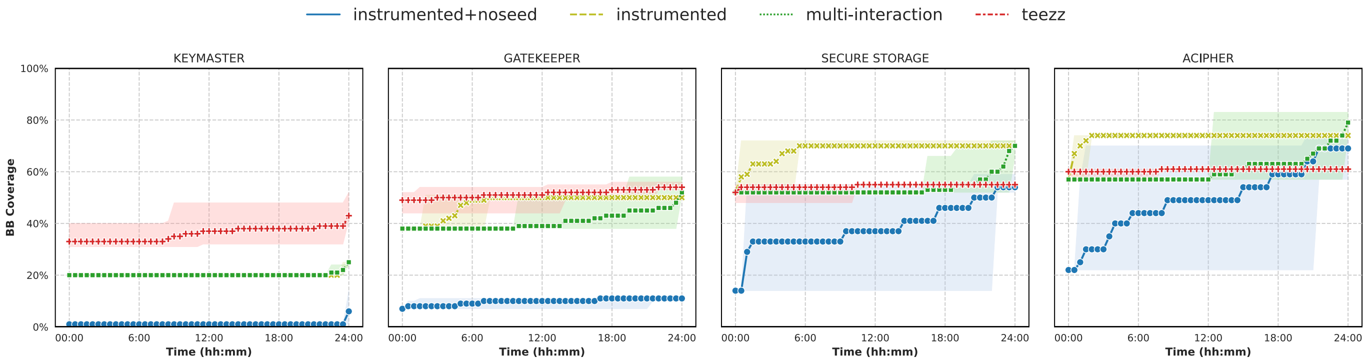


Fig. 6. Coverage of TEEzz and optee-afl (in instrumented+noseed, intrumented, and multi-interaction mode). The target TAs are `keymaster`, `gatekeeper`, `secure storage`, and `acipher`. Due to TEEzz's TA-awareness, it can mutate inputs in a type-aware way and resolve inter-interaction value dependencies, resulting in an up-to two times higher coverage than other state-of-the-art fuzzers.

to fingerprint authentication or face identification) since there is no appropriate way to fuzz them.

For each non-AOSP CA, an analyst reverse engineered the CA-layer interface and wrote a small CA driver program that correctly uses the interface. On average, the analyst spent five hours recovering the interface and writing a CA driver. Given that the analyst will realistically spend many more hours on triaging crashes anyway, we argue that writing these drivers is a reasonable engineering trade-off and a great practice to familiarize with the interface.

Having a driver and the CA interface definitions for all of our targeted TAs, we first generated the DBII recorders and type-aware mutators. Then, we installed TEEzz's multi-interface interaction recorders and triggered the TA interaction either via the user interface of the phone or our manually developed CA drivers. Based on the interaction recordings, we propagated the types to the low-level high-control TEE driver interface recordings and performed the dependency inference. Finally, we were able to fuzz each of the TAs listed in Table II.

We ran TEEzz for 24 hours for each TA and analyzed all found crashes regarding reproducibility and uniqueness. In total, TEEzz detected $1,541$ crashes of which we could reproduce $1,387$. We deduplicated the crashes on QSEE manually and used the `/dev/hisi_teelog` device on Huawei devices to obtain stack traces of crashing TAs. In total, we found 40 unique crashing inputs. We reported these bugs to the corresponding vendors and already got CVE-2019-10561 assigned.

We reset the target's state during this experiment by rebooting the device after $n = 500$ interactions. On average, this mechanism resulted in $1,517$, $1,073$, 956, and 756 resets during our 24h experiment for the P9 Lite, P20 Lite, Nexus 5X, and Pixel 2XL devices, respectively.

Fuzzing the `keymaster` and `gatekeeper` TAs on the Nexus 5X resulted in unstable ADB connections and corrupted data partitions. These states did not occur on the other devices and are problematic for continuous fuzzing because the host component of TEEzz cannot access the device anymore. We had to perform 130 factory resets and 20 hard resets during this experiment for the `keymaster` and `gatekeeper` TA,

TABLE II
RESULTS OF FUZZING COTS TAS USING TEEZZ. WE FUZZED EACH TA ON EACH DEVICE FOR 24H. ◊ – ENCRYPTED TA

TEE	Device (OS Vers.)	TA	CA	Req/Sec	#Crashes
TC	P9 Lite (6.0)	keymaster	AOSP	9.6	681
TC	P9 Lite (6.0)	gatekeeper	AOSP	11.1	645
TC	P9 Lite (6.0)	secure storage	Vendor	10.7	0
TC	P9 Lite (6.0)	rpmbkey ◊	Vendor	7.3	0
TC	P9 Lite (6.0)	antitheft ◊	Vendor	7.1	0
TC	P9 Lite (6.0)	hwsign ◊	Vendor	7.1	0
TC	P20 Lite (8.0)	keymaster ◊	AOSP	12.7	10
TC	P20 Lite (8.0)	gatekeeper ◊	AOSP	6.4	0
TC	P20 Lite (8.0)	secure storage ◊	Vendor	6.5	0
TC	P20 Lite (8.0)	rpmbkey ◊	Vendor	5.3	0
TC	P20 Lite (8.0)	antitheft ◊	Vendor	0.7	0
TC	P20 Lite (8.0)	hwsign ◊	Vendor	5.8	0
QSEE	Nexus 5X (7.1.2)	keymaster	AOSP	3.8	55
QSEE	Nexus 5X (7.1.2)	gatekeeper	AOSP	6.0	0
QSEE	Nexus 5X (7.1.2)	widevine	AOSP	7.2	80
QSEE	Pixel 2 XL (9.0)	keymaster	AOSP	5.0	0
QSEE	Pixel 2 XL (9.0)	gatekeeper	AOSP	3.3	0
QSEE	Pixel 2 XL (9.0)	widevine	AOSP	5.3	70

respectively. To automatically handle these cases and allow for continuous fuzzing, TEEzz is capable of booting the phone into recovery mode and restoring a functional system. Additionally, our fuzzer can carry out hard resets using a phone case equipped with a servo motor that pushes the phone's power button until it reboots.

TEEzz found 40 unique bugs in TAs deployed across 4 different COTS Android phones. This result provides evidence that TEEzz is capable of effectively finding bugs in TAs deployed on production devices.

VIII. LIMITATIONS

Although TEEzz provides an effective way to fuzz test TAs, it suffers from several limitations.

Unlocked device: TEEzz requires complete control of the rOS for which the bootloader needs to be unlocked. Vendors who do not allow unlocking their bootloaders are challenging targets for TEEzz.

Availability of CAlibs: In the worst-case scenario, some one-time manual effort is required per TA to obtain its CAlib's interface definition and implement test cases that trigger interactions with the TA. We argue that synthesizing test cases is a problem at the frontier of science and has barely been solved in situations with source code available (see Fudge [8] and FuzzGen [41]), and is thus an orthogonal problem. With TEEzz we will release and document the tooling that allowed us to fuzz each TA listed in Table II.

Extending to other TEEs: Our current prototype can fuzz TAs on three popular platforms. While the concepts generally apply to other TEEs, TEEzz needs adaptation to run on those platforms. We plan to support a further target in the future: Samsung's TEE called TEEGRIS [68]. As far as we know, TEEGRIS is similar to TC and OPTEE, and porting TEEzz to this platform should be straightforward.

IX. Related work

Several researchers have studied and exploited vulnerabilities in TZ-based TEEs [46], [67], [72], including a class of flaws, called BOOMERANG [52]. Furthermore, there have been various side-channel attacks on TZ [14], [48], [78], [82], [90]. The static analysis techniques employed by the works mentioned above are difficult to generalize for vulnerability detection in TAs, as the structure and implementation of TAs depend on their tOS [4], [55], [75], [80]. Regardless, vendors started to encrypt TAs [85], making it difficult to retrieve their binaries, which renders static analysis techniques inapplicable.

Commercial TEEs with encrypted TAs are essentially blackbox systems that expose certain functionalities or APIs [27] that are accessible from the rOS through SMC [6] instructions. Fuzzing [13], [22], [26], [28] is a well-known technique to test blackbox systems. There has been significant progress in whitebox [26] and greybox [11] fuzzing regarding performance [54], [86], coverage [12], [49], [64], [71], [73] and bug finding ability [17], [18], [61]. However, these techniques need access to the binary of the program under test. Although, Harrison *et al.* recently proposed a tOS emulation technique, PartEmu [39], which can be used to fuzz the corresponding TAs, this technique fails when the tOS and TAs are encrypted. Consequently, these techniques cannot be applied directly to fuzz TAs.

A further technique to fuzz blackbox systems is grammar-based fuzzing [29]. This technique uses the grammar of the target program's input to generate fuzz inputs. Peach [60] is a well-known commercial grade tool for grammar-based fuzzing capable of processing complex inputs. Despite this, getting the input format accepted by TAs is difficult because there is no standard input format. Furthermore, well-known interface recovery techniques [9], [19], [63], [65], [81] cannot be used as they rely on the availability of the program binary.

Almost all commercial TEEs expose an interface in the rOS, usually in the form of a device driver [1], [3], [74]. Most of these device drivers are open-source and expose high-level formats of the input accepted by the TAs, like raw request and response buffers [2]. Recently, fuzzing techniques targeting device drivers have been proposed [20], [38], [43], [77]. Specifically, DIFUZE [20] uses static analysis to infer the accepted input format. This format is then used to effectively fuzz the driver, or applied by other device-driver fuzzers, like Syzkaller [32]. As we show in Section A in our appendix, these techniques are ineffective in generating inputs for TAs. In comparison, TEEzz is the first work opportunistically using different techniques to fuzz commercial TAs effectively.

X. Conclusions

While TEEs in modern smartphones are intended to provide a safe haven for sensitive data and computations, they also pose a major security risk. Privileged code running within the TEE has *complete* access to every aspect of the smartphone (e.g., cryptographic keys, hardware peripherals, and sensitive user data). Unfortunately, despite this potentially catastrophic security risk, traditional security analyses, such as fuzz testing, are rendered useless due to the limited access to and lack of feedback from production TAs. To address this analysis gap, we present TEEzz, the first TEE-aware fuzzer capable of effectively fuzzing TAs on production smartphones. TEEzz leverages a combination of DBI and stateful replay techniques to ensure that both the protocol and structure expected by the targeted TAs are met, enabling almost *all* of the fuzzed inputs to be processed. Indeed, TEEzz discovered over 40 unique crashes on QSEE and TC combined, resulting in one CVE so far, and found 13 previously unknown bugs in TAs running on OPTEE, an open-source reference implementation for TZ-based TEE.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback and appreciate the opportunity for a major revision to conduct additional experiments. This project was supported by European Research Council (ERC) grant No. 850868, SNSF grant PCEGP2_186974, DARPA grants R001119S0089-AMP-FP-034 and N6600120C4031, DHS grant FA8750-19-2-0005, and ONR grants N00014-20-1-2632 and N00014-17-1-2897. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Huawei Trusted Core Kernel Driver. https://github.com/OpenKirin/android_kernel_huawei_hi3650/tree/7.x/drivers/hisi/tzdriver.

[2] QSEE Request and Response Sizes. https://android.googlesource.com/kernel/msm.git/+/77cac325253126dd9e6c480d885aa51f1abf3c40/drivers/misc/qseecom.c#97.

[3] QSEECOM Driver. https://android.googlesource.com/kernel/msm.git/+/77cac325253126dd9e6c480d885aa51f1abf3c40/drivers/misc/qseecom.c.

[4] QSEEComAPI.h. https://android.googlesource.com/platform/hardware/qcom/keymaster/+/master/QSEEComAPI.h.

[5] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 1–15, 2016.

[6] ARM. SMC Calling Convention. http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf.

[7] ARM. Tee reference documentation, 2018. https://www.arm.com/why-arm/technologies/trustzone-for-cortex-a/tee-reference-documentation.

[8] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: fuzz driver generation at scale. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 975–985. ACM, 2019.

[9] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *CoRR*, abs/1608.01723, 2016.

[10] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the FREENIX*, pages 41–46. USENIX, 2005.

[11] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2329–2344. ACM, 2017.

[12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[13] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 122–131. IEEE Press, 2013.

[14] Sebanjila Kevin Bukasa, Ronan Lashermes, Hélène Le Bouder, Jean-Louis Lanet, and Axel Legay. How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip. In *Proceedings of the IFIP International Conference on Information Security Theory and Practice*, pages 93–109. Springer, 2017.

[15] Marcel Busch, Johannes Westphal, and Tilo Müller. Unearthing the trustedcore: A critical review on huawei's trusted execution environment. In Yuval Yarom and Sarah Zennou, editors, *Proceedings of the Workshop on Offensive Technologies, WOOT*. USENIX Association, 2020.

[16] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 18–20, 2020.

[17] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, SP '15, pages 725–741, Washington, DC, USA, 2015. IEEE Computer Society.

[18] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307*, 2018.

[19] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, SP '09, pages 110–125, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: interface aware fuzzing for kernel drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2123–2138. ACM, 2017.

[21] CVE. Google android security vulnerabilities, 2018. https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html.

[22] Jared DeMott. The evolving art of fuzzing. *DEF CON*, 14, 2006.

[23] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In Jeffrey Todd McDonald, Mila Dalla Preda, and Natalia Stakhanova, editors, *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 4:1–4:11. ACM, 2015.

[24] Jan-Erik Ekberg, Kari Kostiainen, and N Asokan. Trusted execution environments on mobile devices. In *Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS)*, pages 1497–1498. ACM, 2013.

[25] Tao Feng, Nicholas DeSalvo, Lei Xu, Xi Zhao, Xi Wang, and Weidong Shi. Secure session on mobile: An exploration on combining biometric, trustzone, and user behavior. In *Proceedings of the Mobile Computing, Applications and Services (MobiCASE)*, pages 206–215. IEEE, 2014.

[26] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society.

[27] GlobalPlatform. *TEE Internal Core API Specification*, 1.1.1 edition, 2016.

[28] Patrice Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd international workshop on Random testing*, pages 1–1. ACM, 2007.

[29] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[30] Google. Drm, 2001. https://source.android.com/devices/drm.

[31] Google. Google play billing overview, 2001. https://developer.android.com/google/play/billing/billing_overview.

[32] Google. syzkaller - linux syscall fuzzer, 2017. https://github.com/google/syzkaller.

[33] Google. Android hal, 2018. https://source.android.com/devices/architecture/hal.

[34] Google. Android security bulletins, 2018. https://source.android.com/security/bulletin.

[35] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. An empirical study on api-misuse bugs in open-source c programs. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 11–20, 2019.

[36] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 488–501. ACM, 2017.

[37] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 135–150, 2019.

[38] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2345–2358. ACM, 2017.

[39] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, Michael Grace, Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, Hayawardh Vijayakumar, et al. Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[40] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. Fuzzgen: Automatic fuzzer generation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 2271–2287, 2020.

[41] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. Fuzzgen: Automatic fuzzer generation. In Srdjan Capkun and Franziska Roesner, editors, *Proceedings of the USENIX Security Symposium (USENIX Security)*, pages 2271–2287. USENIX Association, 2020.

[42] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang. Privatezone: Providing a private execution environment using arm trustzone. *IEEE Transactions on Dependable and Secure Computing*, 15(5):797–810, 2018.

[43] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, page 0. IEEE, 2018.

[44] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: hybrid fuzzing on the linux kernel. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2020.

[45] Stephan Kleber, Lisa Maile, and Frank Kargl. Survey of protocol reverse engineering algorithms: Decomposition of tools for static traffic analysis. *IEEE Communications Surveys & Tutorials*, 2018, 2018.

[46] laginimaineb. Exploring qualcomms secure execution environment, 2016. http://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html.

[47] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

[48] Paul Leignac, Olivier Potin, Jean-Baptiste Rigaud, Jean-Max Dutertre, and Simon Pontié. Comparison of side-channel leakage on rich and

trusted execution environments. In *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*, pages 19–22, 2019.

[49] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 627–637, New York, NY, USA, 2017. ACM.

[50] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[51] Linaro Limited. Open portable trusted execution environment, 2020. https://www.op-tee.org/.

[52] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[53] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. C to Checked C by 3C. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, October 2022.

[54] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. *arXiv preprint arXiv:1812.11875*, 2018.

[55] Hadi Nahari. TLK: A FOSS Stack for Secure Hardware Tokens. http://www.w3.org/2012/webcrypto/webcrypto-next-workshop/papers/webcrypto2014_submission_25.pdf, 2012.

[56] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In *Proceedings of the Collaboration and Internet Computing (CIC)*, pages 445–451. IEEE, 2016.

[57] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 27–41, New York, NY, USA, 2016. ACM.

[58] @oleavr. Frida, 2020. https://frida.re/.

[59] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In William Enck and Adrienne Porter Felt, editors, *Proceedings of the USENIX Security Symposium (USENIX Security)*, pages 729–743. USENIX Association, 2018.

[60] Peach. The peach fuzzer, 2017. http://www.peachfuzzer.com/.

[61] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 697–710. IEEE, 2018.

[62] Qualcomm. Qualcomm mobile security, 2018. https://www.qualcomm.com/solutions/mobile-computing/features/security.

[63] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[64] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[65] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, SEC'14, pages 861–875, Berkeley, CA, USA, 2014. USENIX Association.

[66] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Bootstomp: on the security of bootloaders in mobile devices. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2017.

[67] Dan Rosenberg. Reflections on trusting trustzone. *BlackHat USA*, 2014.

[68] Samsung. Samsung teegris, 2020. https://developer.samsung.com/teegris/overview.html.

[69] Thorsten Schreiber. Android binder. *A shorter, more general work, but good for an overview of Binder. http://www. nds. rub. de/media/attachments/files/2012/03/binder. pdf*, 2011.

[70] SecWiki. Android kernel exploits, 2018. https://github.com/SecWiki/android-kernel-exploits.

[71] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. *machine learning*, 89(46):38, 2018.

[72] Di Shen. Exploiting trustzone on android. *Black Hat USA*, 2015.

[73] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[74] STMicroelectronics and Linaro Security Working Group. OP-TEE non-secure world-secure world driver. https://github.com/linaro-swg/linux/blob/optee/drivers/tee.

[75] STMicroelectronics and Linaro Security Working Group. OP-TEE non-secure world-secure world smc call. https://github.com/linaro-swg/linux/blob/optee/drivers/tee/optee/call.c:L117.

[76] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 976–988. ACM, 2015.

[77] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, pages 291–307, 2018.

[78] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Clkscrew: exposing the perils of security-oblivious energy management. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, pages 1057–1074, 2017.

[79] Huawei Technologies. Emui 8.0 security technical white paper, 2017. https://consumer-img.huawei.com/content/dam/huawei-cbg-site/en/mkt/legal/privacy-policy/EMUI8.0SecurityTechnologyWhitePaper.pdf.

[80] Trustonic. trustonic-tee-user-space. https://github.com/Trustonic/trustonic-tee-user-space/blob/e3b0b06025605b06fc1e19588098e5011f6afc83/MobiCoreDriverLib/Daemon/MobiCoreDriverDaemon.cpp, February 2015.

[81] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. Secfuzz: Fuzz-testing security protocols. In *Proceedings of International Workshop on Automation of Software Test (AST)*, pages 1–7. IEEE, 2012.

[82] Jie Wang, Kun Sun, Lingguang Lei, Shengye Wan, Yuewu Wang, and Jiwu Jing. Cache-in-the-middle (citm) attacks: Manipulating sensitive data in isolated execution environments. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1001–1015, 2020.

[83] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '18, pages 1137–1150, New York, NY, USA, 2018. ACM.

[84] xairy. kernel exploits, 2018. https://github.com/xairy/kernel-exploits.

[85] XePeleato. Huawei kirin trustzone, 2017. https://github.com/OpenKirin/Documentation/blob/master/04-Trustzone.md.

[86] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2313–2328. ACM, 2017.

[87] Sileshi Demesie Yalew, Gerald Q Maguire, Seif Haridi, and Miguel Correia. T2droid: A trustzone-based dynamic analyser for android applications. In *Proceedings of the Trustcom/BigDataSE/ICESS*, pages 240–247. IEEE, 2017.

[88] Google Project Zero. Trust issues: Exploiting trustzone tees, 2018. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html.

[89] Dongli Zhang. Trustfa: Trustzone-assisted facial authentication on smartphone. Technical report, Technical Report, 2014.

[90] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[91] Vincent Zimmer and Michael Krau. Establishing the root of trust, 2016.

Appendix A

From the perspective of a userland program on Android, the TEE appears as a device which is exposed by the Linux kernel as a device driver node in the filesystem. Since device driver fuzzing is an established research field, it comes naturally to use a kernel fuzzer to fuzz the TEE and TAs in particular. Our intuition for kernel fuzzers is that a general-purpose approach to device driver fuzzing is insufficient for the complex inputs and interactions required by the TEE. To validate this hypothesis and get a deeper understanding of its implications, we chose two representative kernel fuzzers, namely Syzkaller [32] and DIFUZE [20], to fuzz TEE drivers on COTS Android devices. We prepared both of these fuzzers according to the available documentations and deployed them on two COTS devices, the Pixel 2 XL (Android 9.0) and the Huawei P20 Lite (Android 8.0). The Pixel 2 XL is based on a Qualcomm chipset and, hence, runs QSEE. The Huawei P20 Lite is based on a HiSilicon chipset and runs TC. For this experiment, we rooted both devices and deployed custom kernels. The modifications to the original kernels are minimal. We instrumented the locations where the Linux kernel executes a *secure monitor call* instruction to count the number of context switches to the TEE and to count the number of `ioctl` calls that the fuzzers generated. Additionally, we enabled *kcov* for the Syzkaller experiment. Since DIFUZE is not a coverage-guided fuzzer, we did not enable this feature.

For Syzkaller, we manually specified the TEE driver's syscalls and different `ioctl`-handlers, including the different `argp` data structures expected by these handlers. Furthermore, Syzkaller's grammar allowed us to specify stateful APIs. For example, a filedescriptor that is returned from an `open` syscall can be linked to the filedescriptor consumed by an `ioctl` syscall.

DIFUZE automatically generates data models to fuzz `ioctl` calls from the driver's source code, but, in comparison to syzkaller, it does not support stateful APIs or coverage-guidance.

TABLE III
RESULTS OF USING DIFUZE AND SYZKALLER TO FUZZ TEE DRIVERS ON COTS PHONES. BOTH FUZZERS RAN FOR 24 HOURS.

Technique	Σ_{ioctl} (per sec)	Σ_{SMC} (% Σ_{ioctl})
QSEE (Pixel 2 XL - taimen)		
$DIFUZE$	7,370,756 (85.3)	112,854 (1.53%)
$Syzkaller$	7,006,579 (81.1)	229,784 (3.28%)
TrustedCore (Huawei P20 Lite)		
$DIFUZE$	1,513,929 (17.5)	540,019 (35.67%)
$Syzkaller$	10,847,220 (125.5)	2,245,970 (20.71%)

According to our experiment shown in Table III, DIFUZE and Syzkaller have a respectable `ioctl` throughput of up to 85.3 and 125.5 requests per second (Σ_{ioctl} per sec on the Pixel 2XL and the Huawei P20 Lite, respectively).

Contrary to our hypothesis that driver interface fuzzing techniques would not be able to generate any SMC requests (i.e., Σ_{SMC} = 0), DIFUZE and Syzkaller were both able to generate those requests by *just* fuzzing the driver interface. On QSEE, a small fraction (1.53% and 3.28%) of invocations reach the TEE, and on TC, up to a third of the calls reaches the TEE (35.67% and 20.71%).

To better understand this counterintuitive observation, we investigated the specific `ioctl` requests that yield SMCs. A TEE driver supports several command handlers that are not related to the TA lifecycle. Those handlers include facilities to query the tOS's version or synchronize the time with the tOS. Not a single generated SMC is triggered by a command handler related to the interaction with a TA. Consequently, the interface to communicate with TAs is not reached at all.

Looking at the design of both kernel fuzzers, it is apparent that they would require extensive adaptations to incorporate a fuzzing harness that supports establishing sessions to TAs. Hence, we conclude that kernel fuzzers are ineffective for fuzzing TAs.

APPENDIX B

Since PartEmu [39] is the only fuzzer targeting proprietary TAs, we contacted the authors to support our evaluation of TEEzz because PartEmu is not open to the public. According to the authors, the prototype cannot be made available due to parts of it being under a non-disclosure agreement with Samsung Research, and the individual TAs or firmware images used as a dataset in their evaluation cannot be revealed due to the security-sensitive nature of TZ.

We agreed with them to compare our results of a six-hour fuzzing session of the `widevineTA` on the Pixel 2XL platform since this target was a common denominator. PartEmu was able to discover two shallow bugs in the target. One of these bugs was already discovered and reported during the development of TEEzz and the corresponding CVE-2019-10561 is assigned to us. The other bug discovered by PartEmu was not found by TEEzz because we never captured a seed during the interaction of `mediaserver` with the `widevine` TA that triggers the affected command handler. This limitation is due to our reliance on captured seeds and could be mitigated by adding an exploration stage to TEEzz in order to discover the supported commands of a given TA.

TEEzz was able to find one more bug that is located five function calls deep from the command handler in the `widewine` TA. According to its authors, PartEmu only discovered shallow bugs and never reached deeper into the target's logic than two to three function calls deep.

An inherent limitation of PartEmu is that it cannot be utilized for the encrypted TEE firmware images used on the Huawei P20 Lite. In comparison, TEEzz could not only fuzz the TAs on this device but also found a bug in the `keymaster` TA.