SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices

Qinying Wang*†, Boyu Chang*, Shouling Ji*(✉), Yuan Tian‡, Xuhong Zhang*, Binbin Zhao§, Gaoning Pan*, Chenyang Lyu*, Mathias Payer†, Wenhai Wang*, Raheem Beyah§
*Zhejiang University, †EPFL, ‡University of California, Los Angelos, §Georgia Institute of Technology
E-mails: {wangqinying, bychang, sji}@zju.edu.cn, yuant@ucla.edu, zhangxuhong@zju.edu.cn, binbin.zhao@gatech.edu, {pgn, puppet}@zju.edu.cn, mathias.payer@nebelwelt.net, zdzzlab@zju.edu.cn, rbeyah@ece.gatech.edu

Abstract—**Trusted Execution Environments (TEEs) embedded in IoT devices provide a deployable solution to secure IoT applications at the hardware level. By design, in TEEs, the Trusted Operating System (Trusted OS) is the primary component. It enables the TEE to use security-based design techniques, such as data encryption and identity authentication. Once a Trusted OS has been exploited, the TEE can no longer ensure security. However, Trusted OSes for IoT devices have received little security analysis, which is challenging from several perspectives: (1) Trusted OSes are closed-source and have an unfavorable environment for sending test cases and collecting feedback. (2) Trusted OSes have complex data structures and require a stateful workflow, which limits existing vulnerability detection tools.**

To address the challenges, we present SYZTRUST, the first state-aware fuzzing framework for vetting the security of resource-limited Trusted OSes. SYZTRUST adopts a hardware-assisted framework to enable fuzzing Trusted OSes directly on IoT devices as well as tracking state and code coverage non-invasively. SYZTRUST utilizes composite feedback to guide the fuzzer to effectively explore more states as well as to increase the code coverage. We evaluate SYZTRUST on Trusted OSes from three major vendors: Samsung, Tsinglink Cloud, and Ali Cloud. These systems run on Cortex M23/33 MCUs, which provide the necessary abstraction for embedded TEEs. We discovered 70 previously unknown vulnerabilities in their Trusted OSes, receiving 10 new CVEs so far. Furthermore, compared to the baseline, SYZTRUST has demonstrated significant improvements, including 66% higher code coverage, 651% higher state coverage, and 31% improved vulnerability-finding capability. We report all discovered new vulnerabilities to vendors and open source SYZTRUST.

1. Introduction

Trusted Execution Environments (TEEs) are essential to securing important data and operations in IoT devices. GlobalPlatform, the leading technical standards organization, has reported a 25-percent increase in the number of TEE-enabled IoT processors being shipped quarterly, year-over-year [1]. Recently, major IoT vendors such as Samsung have designed TEEs for low-end Microcontroller Units (MCUs) [2], [3], [4] and device manufacturers have embedded the TEE in IoT devices such as unmanned aerial vehicles and smart locks, to protect sensitive data and to provide key management services. A TEE is composed of Client Applications (CAs), Trusted Applications (TAs), and a Trusted Operating System (Trusted OS). Among them, the Trusted OS is the primary component to enable the TEE using security-based design techniques, and its security is the underlying premise of a reliable TEE where the code and data are protected in terms of confidentiality and integrity. Unfortunately, implementation flaws in Trusted OSes violate the protection guarantees, bypassing confidentiality and integrity guarantees. These flaws lead to critical consequences, including sensitive information leakage (CVE-2019-25052) and code execution within the Trusted OS context [5], [6]. Once attackers gain control of Trusted OSes, they can launch critical attacks, such as creating a backdoor to the Linux kernel [7], and extracting full disk encryption keys of Android's KeyMaster service [8].

While TEEs are increasingly embedded in IoT devices, the security of Trusted OS for IoT devices remains under studied. Considering the emerging amount of diversified MCUs and IoT devices, manual analysis, such as reverse engineering, requires significant expert efforts and is therefore infeasible at scale. Recent academic works use fuzzing to automate TEE testing. However, unlike Trusted OSes for Android devices, Trusted OSes for IoT devices are built on TrustZone-M with low-power and cost-sensitive MCUs, including NuMicro M23. Thus, Trusted OSes for IoT devices are more hardware-dependent and resource-constrained, complicating the development of scalable and usable testing approaches with different challenges. In the following, we conclude two challenges for fuzzing IoT Trusted OSes.

Challenge I: The inability of instrumentation and restricted environment. Most Trusted OSes are closed-source. Additionally, TEE implementations, especially the Trusted OSes are often encrypted by IoT vendors, which implies the inability to instrument and monitor the code execution in the secure world. Accordingly, classic feedback-driven fuzzing cannot be directly applied to the scenario of testing TEEs including TAs and Trusted OSes. Existing works either rely on on-device binary instrumentations [9]

Shouling Ji is the corresponding author.

or require professional domain knowledge and rehosting through proprietary firmware emulation [10] to enable testing and coverage tracking. However, as for the Trusted OSes designed for IoT devices, the situation is more challenging due to the following two reasons. First, IoT devices are severely resource-limited, while existing binary instrumentations are heavy-weight for them and considerably limit their execution speed. Second, as for rehosting, IoT devices are mostly hardware-dependent, rendering the reverse engineering and implementation effort for emulated software and hardware components unacceptable. In addition, rehosting faces the limitation of the inaccuracy of modeling the behavior of hardware components. To our best knowledge, the only existing TEE rehosting solution PartEmu [10] is not publicly available and does not support the mainstream TEE based on Cortex-M MCUs designed for IoT devices.

Challenge II: Complex structure and stateful workflow. Trusted OSes for IoT devices are surprisingly complex. Specifically, Trusted OSes implement multiple cryptographic algorithms, such as AES and MAC, without underlying hardware support for these algorithms as would be present on Cortex-A processors. To implement these algorithms in a secure way, Trusted OSes maintain several state diagrams to store the execution contexts and guide the execution workflow. To explore more states of a Trusted OS, a fuzzer needs to feed syscall sequences in several specific orders with different specific state-related argument values. Without considering such statefulness of Trusted OSes, coverage-based fuzzers are unlikely to explore further states, causing the executions to miss the vulnerabilities hidden in a deep state. Unfortunately, existing fuzzing techniques lack state-awareness for Trusted OSes. Specifically, they have trouble understanding which state a Trusted OS reaches since there are no rich-semantics response codes to indicate it. In addition, due to the lack of source code and the inability of instrumentation, it is hard to infer and extract the state variables by program analysis.

Our solution. To address the above key challenges, we propose and implement SyzTrust, the first fuzzing framework targeting Trusted OSes for IoT devices, supporting state and coverage-guided fuzzing. Specifically, we propose an on-device fuzzing framework and leverage a hardware-in-the-loop approach. To support in-depth vulnerability detection, we propose a composite feedback mechanism that guides the fuzzer to explore more states and increase code coverage.

SyzTrust necessitates diverse contributions. First, to tackle Challenge I, we propose a hardware-assisted fuzzing framework to execute test cases as well as collect code coverage feedback. Specifically, we decouple the execution engine from the rest of the fuzzer to enable directly executing test cases in the protective TEE secure world on the resource-limited MCU. To support coverage tracking, we present a selective trace collection approach instead of costly code instrumentation to enable tracing instructions on a target MCU. In particular, we leverage the ARM Embedded Trace Macrocell (ETM) feature to collect raw trace packets by monitoring instruction and data buses on MCU with a low-performance impact. However, the Trusted OS for IoT devices is resource constrained, which makes storing ETM traces on board difficult and limits the fuzzing speed. Additionally, the TEE internals are complicated and have multiple components, which generate noisy trace packets. Therefore, we offload heavy-weight tasks to a PC and carefully scheduled the fuzzing subprocesses in a more parallel way. We also present an event- and address-based trace filter to handle the noisy trace packets that are not executed by the Trusted OS. Additionally, we adopt an efficient code coverage calculation algorithm directly on the raw packets.

Second, as for the Challenge II, the vulnerability detection capability of coverage-based fuzzers is limited, and a more effective fuzzing strategy is required. Therefore, we propose a composite feedback mechanism, which enhances code coverage with state feedback. To be specific, we utilize state variables that control the execution contexts to present the states of a Trusted OS. However, such state variables are usually stored in closed-source and customized data structures within Trusted OSes. Existing state variable inference methods either use explicit protocol packet sequences [11] or require source codes of target software [12], [13], which are unavailable for Trusted OSes. Therefore, to identify the state-related members from those complex data structures, SyzTrust collects some heuristics for Trusted OS and utilizes them to perform an active state variable inference algorithm. After that, SyzTrust monitors the state variable values during the fuzzing procedure as the state feedback.

Finally, SyzTrust is the first end-to-end solution capable of fuzzing Trusted OSes for IoT devices. Moreover, the design of the on-device fuzzing framework and modular implementation make SyzTrust more extensible. With several MCU-specific configurations, SyzTrust scales to Trusted OSes on different MCUs from different vendors.

Evaluation. We evaluate SyzTrust on real-world Trusted OSes from three leading IoT vendors Samsung, Tsinglink Cloud, and Ali Cloud. The evaluation result shows that SyzTrust is effective at discovering new vulnerabilities and exploring new states and codes. As a result, SyzTrust has discovered 70 new vulnerabilities. Among them, vendors confirmed 28, and assigned 10 CVE IDs. The vendors are still investigating others. Compared to state-of-the-art approaches, SyzTrust finds more vulnerabilities, hits 66% higher code branches, and 651% higher state coverage.

Summary and contributions.

- We propose SyzTrust, the first fuzzing framework targeting Trusted OSes for IoT devices, supporting effective state and code coverage guided fuzzing. With a carefully designed hardware-assisted fuzzing framework and a composite feedback mechanism, SyzTrust is extensible and configurable to different IoT devices.
- With SyzTrust, we evaluate three popular Trusted OSes on three leading IoT vendors and detect several previously unknown bugs. We have responsibly reported these vulnerabilities to the vendors and got acknowledged from vendors such as Samsung. We release SyzTrust as an open-source tool for facilitating further studies at https://github.com/SyzTrust.

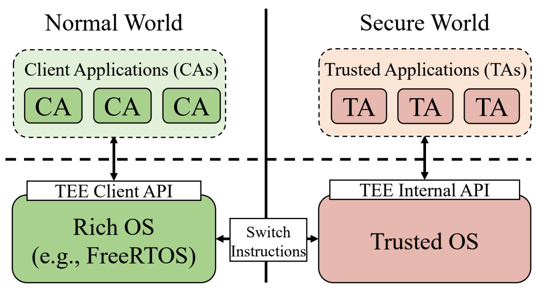


Figure 1: Structure of TrustZone-M based TEE.

2. Background

2.1. TEE and Trusted OS

A TEE is a secure enclave on a device's main processor that is separated from the main OS. It ensures the confidentiality and integrity of code and data loaded inside it [14]. For standardizing the TEE implementations, GlobalPlatform (GP) has developed a number of specifications. For instance, it specifies the TEE Internal Core API implemented in the Trusted OS to enable a TA to perform its security functions [15]. However, it is difficult for vendors to implement a Trusted OS correctly since there are lots of APIs with complex and stateful functions defined in the GP TEE specifications. For instance, the TEE Internal Core API defines six types of APIs, including cryptographic operations APIs supporting more than 20 complex cryptographic algorithms. In addition, the TEE Internal Core API also requires that a Trusted OS shall implement state diagrams to manage the operation. The two aforementioned factors make it challenging for device vendors to develop a secure Trusted OS.

ARM TrustZone has become the de-facto hardware technology to implement TEEs in mobile environments [16] and has been deployed in servers [17], low-end IoT devices [18], [19], and industrial control systems [20]. For IoT devices, the Cortex-M23/33 MCUs, introduced by the ARM Community in 2016, are built on new TrustZone for ARMv8-M as security foundations for billions of devices [21]. TrustZone for ARMv8-M Cortex-M has been optimized for faster context switch and low-power applications and is designed from the ground up instead of being reused from Cortex-A [22]. As Figure 1 shows, instead of utilizing a secure monitor in TrustZone for Cortex-A, the division of the secure and normal world is memory-based, and transitions take place automatically in the exception handle mode. Based on the TrustZone-M, IoT vendors provide Trusted OS binaries to the device manufacturers and then the device manufacturers produce devices with device-specific TAs for the end users. This paper focuses on the Trusted OSes from different IoT vendors and provides security insights for device manufacturers and end users.

2.2. Debug Probe

A debug probe is a special hardware device for low-level control of ARM-based MCUs, using DAP (Debug Access Port) provided by the ARM CoreSight Architecture [23]. It bridges the connection between a computer and an MCU and provides full debugging functionality, including watchpoints, flash memory breakpoints, memory, as well as register examination or editing. In addition, a debug probe can record data and instruction accesses at runtime through the ARM ETM feature. ETM is a subsystem of ARM Coresight Architecture and allows for traceability, whose function is similar to Intel PT. The ETM generates `trace elements` for executed signpost instructions that enable reconstruction of all the executed instructions. Utilizing the above features, the debug probe has shown its effectiveness in tracing and debugging malware [24], unpacking Android apps [25], or fuzzing Linux peripheral drivers [26].

3. Threat Model

Our attacker tries to achieve multiple goals: gaining control over, extracting confidential information from, and causing crashes in other Trusted Applications (TAs) hosted on the same Trusted OS or the Trusted OS itself. We consider two practical attack scenarios. First, an attacker can exploit our discovered vulnerabilities by providing carefully crafted data to a TA. They can utilize a malicious Client Application (CA) to pass the crafted data to a TA. For instance, in mTower, CVE-2022-38511 (ID 1 in Table 1) can be triggered by passing a large key size value from a CA to a TA. Second, an attacker can exploit our discovered vulnerabilities by injecting a malicious TA into the secure world. They can do this through rollback attacks or electromagnetic fault injections (CVE-2022-47549).

4. Design

Figure 2 gives an overview of SyzTrust's design. SyzTrust includes two modules: the *fuzzing engine* on the Personal Computer (PC) and the *execution engine* on the MCU. The fuzzing engine generates and sends test cases to the MCU via the debug probe. The execution engine executes the received test case on the target Trusted OS.

At a high level, we propose a hardware-assisted fuzzing framework and a composite feedback mechanism to guide the fuzzer. Given the inaccessible environment of Trusted OSes, we design a TA and CA pair as a proxy to the Trusted OS and utilize a debug probe to access the MCU for feedback collection. To handle the challenge of limited resources, we decouple the execution engine from SyzTrust and only run it on the MCU. This allows SyzTrust, with its resource-demanding core components, to run more effectively on a PC. To handle the statefulness of Trusted OSes, we include state feedback with code coverage in the composite feedback. State variables represent internal states, and our inference method identifies them in closed-source Trusted OSes.

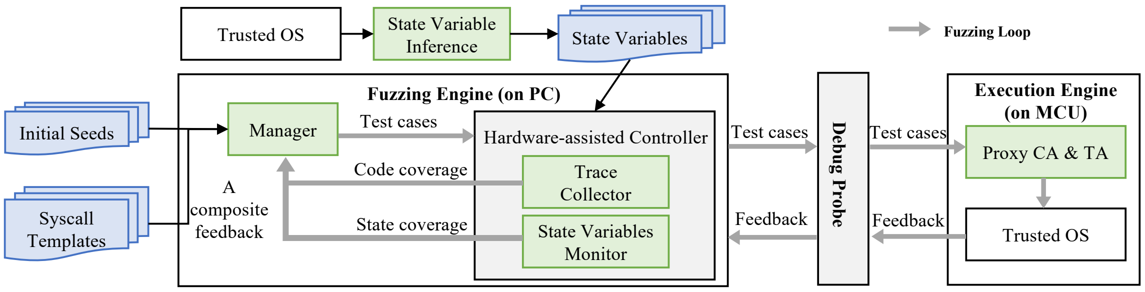


Figure 2: Overview of SYZTRUST. SYZTRUST consists of a fuzzing engine running on a PC, an execution engine running on an MCU, and a debug probe to bridge the fuzzing engine and the execution engine.

The main workflow is as follows. First, SYZTRUST accepts two inputs, including initial seeds and syscall templates. Second, the manager generates test cases through two fuzzing tasks, including generating a new test case from scratch based on syscall templates or by mutating a selected seed. Then, the generated test cases are delivered to the execution engine on MCU through the debug probe. The execution engine executes these test cases to test the Trusted OS. Meanwhile, the debug probe selectively records the executed instruction trace, which is processed by our trace collector as an alternative to code coverage. Additionally, the state variable monitor tracks the values of a set of target state variables to calculate state coverage via the debug probe. Finally, the code and state coverage is fed to the manager as composite feedback to guide the fuzzer. The above procedures are iteratively executed and we denote the iterative workflow as the fuzzing loop.

4.1. Inputs

Syscall templates and initial seeds are fed to SYZTRUST.

Syscall templates. Provided syscall templates, SYZTRUST is more likely to generate valid test cases by following the format defined by the templates. Syscall templates define the syscalls with their argument types as well as special types that provide the semantics of an argument. For instance, `resources` represent the values that need to be passed from an output of one syscall to an input of another syscall, which refers to the data dependency between syscalls. We manually write syscall templates for the Internal Core API for the GP TEE in Syzlang (SYZKALLER's syscall description language [27]). However, a value with a complex structure type, which is common in Trusted OSes, cannot be used as a resource value due to the design limitation of Syzlang. To handle this limitation, we extend the syscall templates with helper syscalls, which accept values of complex structures and output their point addresses as `resource` values. In total, we have 37 syscall templates and 8 helper syscalls, covering all the standard cryptographic APIs.

Initial seeds. Since syscall templates are used to generate syntactically valid inputs, SYZTRUST requires initial seeds to collect some valid syscall sequences and arguments and speed up the fuzzing procedure. Provided initial seeds, SYZTRUST continuously generates test cases to test a target Trusted OS by mutating them. However, there is no off-the-shelf seed corpus for testing Trusted OSes, and constructing one is not trivial, as the syscall sequences with arguments should follow critical crypto constraints, and manually constructing valid sequences will need much effort. For example, a valid seed includes a certain order of syscall sequences to initiate an encryption operation, and the key and the encryption mode should be consistent with supporting encryption. Fortunately, OP-TEE [28], an open-source TEE implementation for Cortex-A, offers a test suite, which can be utilized to construct seeds for most Trusted OSes. Specifically, we automatically inject codes in TAs provided by the test suite to log the syscall names and their arguments. Then we automatically convert those logs into seeds following the format required by SYZTRUST. In addition, we automatically add data dependencies between syscalls in the seed corpus. Accordingly, we identify the return values from syscalls that are input arguments of other syscalls and add helper syscalls for the identified values.

Although the syscall template and seed corpus construction require extra work, it is a one-shot effort and can be used in the further study of Trusted OS security.

4.2. A Hardware-assisted Framework

Now our hardware-assisted fuzzing framework enables the generated test cases to be executed in protective and resource-constrained trusted environments on IoT devices. To handle the constrained resources in Challenge I, we decouple the execution engine from the other components of SYZTRUST to ease the execution overhead of the MCU. As shown in Figure 2, the fuzzing engine, which includes most core components of SYZTRUST and requires more computing resources and memory space, runs on a PC, while only the execution engine runs on the MCU. Thus, SYZTRUST does the heavy tasks, including seed preservation, seed selection, type-aware generation, and mutation.

As for the protected execution environment in Challenge I, we design a pair of CA and TA as the execution engine that executes the test cases to test the Trusted OSes. We utilize a debug probe to bridge the connection between the fuzzing engine and the execution engine. In the following, we introduce the design of delivering test cases and the pair of CA and TA. First, as shown in Figure 2, the debug probe transfers test cases and feedback between the fuzzing engine and the execution engine. Specifically, a debug probe can directly access the memory of the target MCU. Thus, the debug probe accepts generated test cases from the manager engine and writes them to a specific memory of the target MCU. As for test case transportation, we design a serializer to encode the test cases into minimum binary data denoted as payloads before sending them to the MCU. A payload includes a sequence of syscalls with their syscall names, syscall argument types, and syscall argument values. In addition, SYZTRUST denotes syscall names as ordinal numbers to minimize the transportation overhead. Second, the pair of CA and TA plays the role of a proxy to handle the test cases. Accordingly, the CA monitors the specific memory in the MCU. If a payload is written, the CA reads the binary data from the specific memory and delivers them to the TA. Then the TA deserializes the received binary data and executes them one by one to test a Trusted OS implementation. Accordingly, the TA invokes specific syscalls according to the ordinal numbers and fills them in with arguments extracted from the payload. To this end, the TA hardcodes the syscalls declaration codes that are manually prepared. The manual work to prepare the declaration codes is a one-shot effort that can be easily done by referring to the GP TEE Internal Core API specification.

4.3. Selective Instruction Trace Tracking

This section introduces how to obtain the code coverage feedback in SYZTRUST when testing the Trusted OSes. To handle the inability of instrumentation in Challenge I, we present a selective instruction trace track, which is implemented in the trace collector in the fuzzing engine. The trace collector controls the debug probe to collect traces synchronously when the test cases are being executed. After completing a test case, it calculates the code coverage and delivers it to the manager as feedback.

In the selective instruction trace track, we utilize the ETM component to enable non-invasive monitoring of the execution context of target Trusted OSes. The overall workflow is as follows. (1) Before each payload is sent to the MCU, the hardware-assisted controller resets the target MCU via the debug probe. (2) Once the execution engine reads a valid payload from the specific memory space and starts to execute it, the hardware-assisted controller starts the ETM component to record the instruction trace for each syscall from the payload. Specifically, in the meanwhile of executing the syscalls, the generated instruction trace is synchronously recorded and delivered to the fuzzing engine via the debug probe. (3) After completing a payload, the hardware-assisted controller computes the coverage based on the instruction traces for the manager.

```
extern int32_t start_event;
extern int32_t stop_event;
void TA_ProcessEachPayload(){
  RecvPayloadFromCA();
  DecodePayload();
  //Start invoking syscalls one by one
  do {
    // Data access event is triggered, start tracing
    start_event++;
    InvokeOneSyscall();
    // Data access event is triggered, stop tracing
    stop_event++;
    if (AllSyscallsExecuted()){
      break;
    }
  } while(1);
}

```

Listing 1: A code snippet containing the main execution logic from our designed TA.

However, the IoT Trusted OSes are highly resource-constrained, making locally storing ETM traces infeasible and limiting the speed of local fuzzing. Therefore, SYZTRUST utilizes the debug probe (see Section 4.2) to stream all ETM trace data to the host PC in real time while the target system is running. Moreover, SYZTRUST enables parallel execution of test case generation, transmission, and execution, as well as coverage calculation, thereby boosting the speed of fuzzing.

For code coverage, we cannot directly use the raw instruction trace packets generated by the ETM as a replacement for branch coverage due to two issues. First, there is a gap between the raw ETM trace packets and the instruction traces generated by the Trusted OS. The TEE internals are complicated and the ETM component records instruction traces generated by the software running on the MCU, including the CA, rich OS, the TA, and the Trusted OS. Thus, we design a selective instruction trace collection strategy to generate fine-grained traces. ARM ETM allows enabling/disabling trace collection when corresponding events occur. We configure the different events via the Data Watchpoint and Trace Unit (DWT) hardware feature to filter out noisy packets. In SYZTRUST, we aim to calculate the code coverage triggered by every syscall in a test case and generated by Trusted OS. Thus, as shown in Listing 1, we configure the event-based filters by adding two data write access event conditions. The two event conditions start ETM tracing before invoking a syscall and stop ETM tracing after completing the syscall, respectively. In addition, we specify the address range of the secure world shall be included in the trace stream to filter noisy trace packets generated from the normal world.

Second, there is a gap between the raw ETM trace packets to the quantitive coverage results. To precisely recover the branch coverage information, we have to decode the raw trace packets and map them to disassembled binary instruction address. After that, we can recover the instruction traces and construct the branch coverage information [29] [25]. However, disassembling code introduces significant run-time overhead as it incurs high computation cost [30]. Thus, we calculate the branch coverage directly using the raw trace packets [26]. At a high level, this calculation mechanism utilizes a special basic block generated with Linear Code Sequence and Jump (LCSAJ) [31] to reflect any change in basic block transitions. LCSAJ basic blocks consist of the basic address of a raw ETM branch packet and a sequence of branch conditions. The branch conditions indicate whether the instructions followed by the basic address are executed.

This mechanism performs several hash operations on the LCSAJ basic blocks to transform them into random IDs, which we utilize as branch coverage feedback.

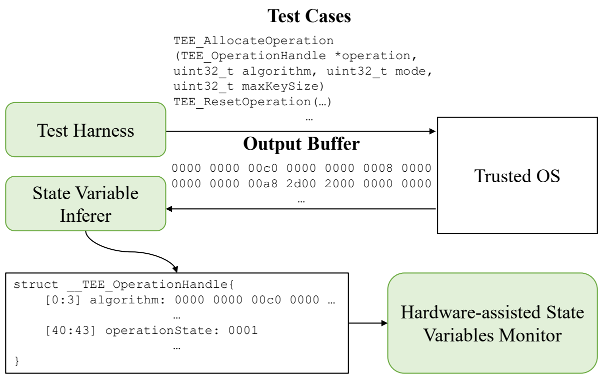
4.4. State Variable Inference and Monitoring

Here we introduce how to identify the internal state of the Trusted OS and how to obtain the state coverage feedback through the state variable monitor component. In particular, the state variable inference provides the state variable monitor with the address ranges of the inferred state variables. Then the state variable monitor tracks the values of these state variables synchronously when the execution engine executes test cases. After completing a test case, the state variable monitor calculates the state coverage and delivers it to the manager as feedback. Below are the details about state variable inference and monitoring.

According to the GP TEE Core API specification, Trusted OSes have to maintain several complex state machines to achieve the cryptographic algorithm in a highly secure way. To explore all states of Trusted OS, the fuzzer needs to feed syscall sequences in several specific orders with different specific state-related argument values. Coverage-based fuzzers are unlikely to explore further states, causing the executions to miss some vulnerabilities hidden in a deep state. For instance, a syscall sequence achieves a new DES cryptographic operation configuration by filling different arguments, whose code coverage may be the same as the syscall sequence to achieve a new AES cryptographic operation configuration. Preserving such syscall sequences as a seed and further exploring them will achieve new cryptographic operations and gain new code coverages. However, a coverage-based fuzzer may discard such syscall sequences that seem to have no code coverage contribution but trigger new internal states. Thus, SyzTrust additionally adopts state coverage as feedback to handle the statefulness of Trusted OSes.

State variable inference. By referring to the GP TEE Internal Core API specification and several open-source TEE implementations from Github, we find that Trusted OSes maintain two important structures, including the `struct` named `TEE_OperationHandle` and the `struct` named `TEE_ObjectHandle`, which present the internal states and control the execution context. We further find that several vital variables (with names such as `operationState` and `flags`) in the two complex `structs` determine the Trusted OS' internal state. Thus, we utilize the value combinations of state variables to present the states of Trusted OS and track all state-related variables to collect their values. Then, we consider a new value of the state variable combination as a new state. However, the `TEE_OperationHandle` and `TEE_ObjectHandle` implementations are customized and close source, making recognizing the state variables and their addresses challenging.

To handle it, we come up with an active inference method to recognize the state variables in the `TEE_OperationHandle` and `TEE_ObjectHandle` implementations. This method is based on the assumption that the state variables in the above two handles will have different values according to different cryptographic operation configurations. For instance, in Samsung's Trusted OS implementation, after executing several syscalls to set cryptographic arguments, the value of a state variable from `TEE_OperationHandle` changes from 0 to 1, which means a cryptographic operation is initialized.



```
TEE_AllocateOperation
(TEE_OperationHandle *operation,
uint32_t algorithm, uint32_t mode,
uint32_t maxKeySize)
TEE_ResetOperation(…)
…
```

```
0000 0000 00c0 0000 0000 0008 0000
0000 0000 00a8 2d00 2000 0000 0000
…
```

```
struct __TEE_OperationHandle{
    [0:3] algorithm: 0000 0000 00c0 0000 …
                    …
    [40:43] operationState: 0001
                    …
}
```

Figure 3: State variable inference.

Based on the assumption, SyzTrust uses a test harness to generate and execute test cases carefully. Then SyzTrust records the buffers of the two handles and applies state variable inference to detect the address ranges of state variables in the recorded buffers, as shown in Figure 3. SyzTrust first filters randomly changeable byte sequences that store pointers, encryption keys, and cipher text. Specifically, SyzTrust conducts a 24-hour basic fuzzing procedure on the initial seeds from Section 4.1 and collects the buffers of the two handles. SyzTrust then parses the buffer into four-byte sequences to recognize changeable values. After that, SyzTrust collects all different values for each byte sequence in the fuzzing procedure and calculates the number of times that these different values occur. SyzTrust considers the byte sequences that occur over 80 times as buffers and excludes them from the state variables. For the remaining byte sequences, SyzTrust applies the following inference to identify state variables. In our observations, the cryptographic operation configurations are determined by the operation-related arguments, including the operation mode (encryption, decryption, or verification) and cryptographic algorithm (AES, DES, or MAC). The cryptographic operation configurations are also determined by the operation-related syscall sequences. We identify such arguments and syscalls by referring to GP TEE Internal Core API specification and conclude the operation-related syscalls include the syscalls that accept the two handles as an input argument and are specified to allocate, init, update, and finish a cryptographic operation, such as `TEE_AllocateOperation` and `TEE_AllocateTriansientObject`. SyzTrust performs mutated seeds that include the operation-related syscalls and records the buffers of the two handles. These byte sequences that vary with certain syscall sequences with certain arguments are considered state variables. Finally,

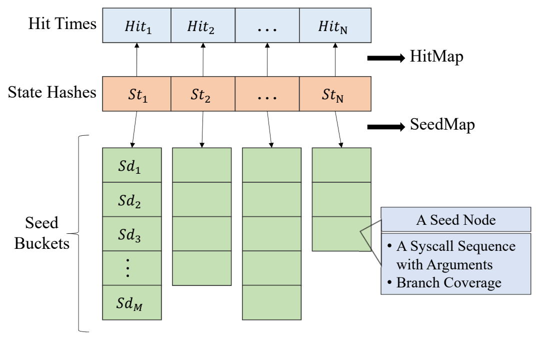


Figure 4: Seed corpus in SYZTRUST.

SYZTRUST outputs the address ranges of these byte sequences and feeds them to the hardware-assisted controller.

State variable monitor. The identified state variables and their address ranges are then used as configurations of the hardware-assisted controller. SYZTRUST utilizes the debug probe to monitor the state variables. When a new `TEE_OperationHandle` or `TEE_ObjectHandle` is allocated, SYZTRUST records its start address. Given the start address of the two handles and address ranges of state variables, SYZTRUST calculates the memory ranges of state variables and directly reads the memory via the debug probe. In the fuzzing loop, for each syscall, beside the branch coverage, SYZTRUST records the hash of combination values of the state variables as the state coverage.

4.5. Fuzzing Loop

After collecting code and state coverage feedback, the fuzzer enters the main fuzzing loop implemented in the manager. SYZTRUST has a similar basic fuzzing loop to SYZKALLER. It schedules two tasks for test case generation: generating a new test case from syscall templates or mutating an existing one based on selected seeds. As for the generation task, SYZTRUST faithfully borrows the generation strategy from SYZKALLER. As for the mutation task, SYZTRUST utilizes a composite feedback mechanism to explore rarely visited states while increasing code coverage. Notably, SYZTRUST does not adopt the **Triage** scheduling tasks from SYZKALLER due to two key reasons. First, SYZTRUST fuzzes directly on the MCU, enabling swift MCU resets after each test case, thus mitigating false positives in branch coverage. Branch coverage validation through triaging is, therefore, unnecessary. Second, executing test cases for Trusted OSes (averaging 30 syscalls) is time-consuming, incurring a significant overhead for SYZTRUST to minimize a syscall sequence by removing syscalls one by one. Appendix A further experimentally demonstrates our intuition for the new scheduling tasks.

4.6. Composite Feedback Mechanism

SYZTRUST adopts a novel composite feedback mechanism, leveraging both code and state coverage to guide mutation tasks within the fuzzing loop. Specifically, SYZTRUST preserves and prioritizes seeds that trigger new code or states. We design two maps as the seed corpus to preserve seeds according to code coverage and state feedback. With such seed corpus, SYZTRUST then periodically selects seeds from the hash table to explore new codes and new states. This section introduces how SYZTRUST fine-tunes the evolution through a novel composite feedback mechanism, including seed preservation and seed selection strategy.

Seed preservation. Given the composite feedback mechanism, we thus provide two maps as the seed corpus to store seeds that discover new state values and new code, respectively. As shown in Figure 4, in two maps, SYZTRUST calculates the hash of combination values of state variables as the keys. SYZTRUST maps the state hashes to their hit times in the HitMap and maps the state hashes to seed buckets in the SeedMap. In the SeedMap, for a certain state hash, the mapping seed bucket contains one or several seeds that can produce the matching state variable values. To construct the SeedMap, SYZTRUST handles the following two situations. First, if a syscall from a test case triggers a new value combination of state variables, SYZTRUST adds a new state hash in the SeedMap. Then, SYZTRUST maps the new state hash to a new seed bucket. To construct the seed bucket, SYZTRUST utilizes the test case with its branch coverage to construct a seed node and appends the seed bucket with the newly constructed seed node. Second, if a syscall from a test case triggers a new code coverage, SYZTRUST adds a new seed node in a seed bucket. Specifically, SYZTRUST calculates the hash of combination values of state variables produced by the syscall. Then, SYZTRUST looks up the seed bucket that is mapped to the state hash and appends a new seed node that contains the test case and its branch coverage to the seed bucket. Noted, a test case could be stored in multiple seed buckets if it triggers multiple feedbacks at the same time. As for the HitMap, each time after completing a test case, SYZTRUST calculates all the state hashes it triggers and updates the hit times for these state hashes according to their hit times.

Seed selection. Given the preserved seed corpus, SYZTRUST applies a special seed selection strategy to improve the fuzzing efficiency. Algorithm 1 shows how SYZTRUST selects seeds from the corpus. First, SYZTRUST chooses a state and then chooses a seed from the mapping seed buckets according to the state. In state selection, SYZTRUST is more likely to choose a rarely visited state by a weighted random selection algorithm. The probability of choosing a seed is negatively correlated with its hit times. Thus, we assign each seed a weight value, which is the reciprocal of its hit times. Finally, The probability of choosing a seed is equal to the proportion of its weight in the sum of all weights. In seed selection, SYZTRUST is more likely to choose a seed with high branch coverage. To this end, SYZTRUST chooses a seed based on a weighted random selection algorithm, and the probability of choosing a certain seed is equal to the proportion of its branch coverage in all coverages. Notably, the probabilities of choosing a state and seed are dynamically updated since the hit times and the sum of coverage is updated in the fuzzing procedure.

Algorithm 1: Seed Selection Algorithm

```
Input: $C$, SeedMap that maps state hashes to a set of seeds
Input: $H$, HitMap that maps state hashes to hit times
Output: $s$, the selected seed
1  $Sum\_W \leftarrow 0$;
2  $Map\_W =$ initMap() //maps state hashes to their weights ;
3  for $each\ key \in H$ do
4      $t \leftarrow$ getValue($H, key$);
5      $Sum\_W \leftarrow Sum\_W + t^{-1}$;
6  end
7  for $each\ key \in H$ do
8      $t \leftarrow$ getValue($H, key$);
9      $Map\_W \leftarrow Map\_W \cup (key, t^{-1}/Sum\_W)$;
10 end
11 $St \leftarrow$ WeightedRandom_StateSelection($Map\_W$);
12 $seedSets \leftarrow$ getValue($C, St$);
13 $s \leftarrow$ WeightedRandom_SeedSelection($seedSets$);
```

4.7. Scope and Scalability of SyzTrust.

SyzTrust targets Trusted OSes provided by IoT vendors and assumes that (i) a TA can be installed in the Trusted OS, and (ii) target devices have ETM enabled. These assumptions align with typical IoT Trusted OS scenarios. First, given that IoT device manufacturers often need to implement device-specific TAs, Trusted OS binaries supplied by IoT vendors generally allow TA installation. Second, SyzTrust tests IoT Trusted OSes by deploying them on development boards where ETM is enabled by default. Moreover, SyzTrust can directly test the Trusted OSes following the GP TEE Internal Core API specification with MCU-specific configurations. It has built-in support for testing on alternative Trusted OSes, including proprietary ones. Appendix B extends discussion with concrete data.

SyzTrust can support other Trusted OSes. To test a new Trusted OSes on a different MCU, SyzTrust requires MCU configurations, including the address ranges of specific memory for storing payloads, the addresses of the data events for the event-based filter, and the address ranges of secure memory for the address-based filter. We developed tooling in the CA and TA to automatically help the analyst obtain all required addresses. In addition, by following the development documents from IoT TEE vendors, the CA and TA may require slight adjustments to meet the format required by the new Trusted OSes and loaded into the Rich OS and Trusted OS.

To extend SyzTrust to proprietary Trusted OSes, we augment the syscall templates and the API declarations in our designed TA and test harness with the new version of these customized APIs. This can be done by referring to the API documents provided by IoT vendors, which is simple and requires minimum effort. To enable the state-aware feature, we need expert analysis of the state-related structures in the Trusted OS and the use of our state variable inference to collect address ranges. These structures can be extracted from the documents and header files. We rely on two heuristics to help extract them. First, state-related data structures usually have common names, e.g., related to $context$ or $state$. Second, the state structures will be the inputs and outputs of several crypto operation-related syscalls. For example, on Link TEE Air, a pointer named $context$ is used among cryptographic syscalls such as tee_aes_init and tee_aes_update, and can be further utilized to infer state variables. This information can be obtained from the $crypto.h$ header file.

5. Implementation

We have implemented a prototype of SyzTrust on top of Syzkaller. We replaced Syzkaller's execution engine with our custom CA and TA pair, integrating our extended syscall templates, eliminating Syzkaller's triage scheduling task, and implementing our own seed preservation and selection strategy. Sections 4.2, 4.5, and 4.6 detail these adaptations.

Below are details about our implementations. (1) As for the overall fuzzing framework, we use the SEGGER J-Trace Pro debug probe to control the communication between the fuzzing engine and the execution engine, as shown in Figure 5. The pair of CA and TA is developed following the GP TEE Internal Core API specification and is loaded into the MCU following the instructions provided by the IoT vendors. To control the debug probe, we developed a hardware-assisted controller based on the SEGGER J-Link SDK. The hardware-assisted controller receives commands from the manager and sends feedback collected on the MCU to the manager via socket communications. (2) For the selective instruction trace track, SyzTrust integrates the ETM tracing component of the SEGGER J-Trace Pro and records the instruction traces from Trusted OSes non-invasively. The raw ETM packets decoder and branch coverage calculation are accomplished in the hardware-assisted controller. (3) For the state variable inference and monitoring, SyzTrust follows the testing strategy in Section 4.4 and utilizes an RTT component [32] from the SEGGER J-Trace Pro to record related state variable values and deliver them to the fuzzing engine. The RTT component accesses memory in the background with high-speed transmission.

Several tools help us analyze the root cause of detected crashes. We utilize CmBacktrace, a customized backtrace tool [33], to track and locate error codes automatically. Additionally, we develop TEEKASAN based on KASAN [34] and MCUASAN [35] to help identify out-of-bound and use-after-free vulnerabilities. We integrate TEEKASAN with lightweight compiler instrumentation and develop shadow memory checking for bug triaging on the open source Trusted OSes. Since TEEKASAN only analyzed a small number of vulnerabilities due to the limitation of the instrumentation tool and most Trusted OSes are closed-source, we manually triage the remaining vulnerabilities.

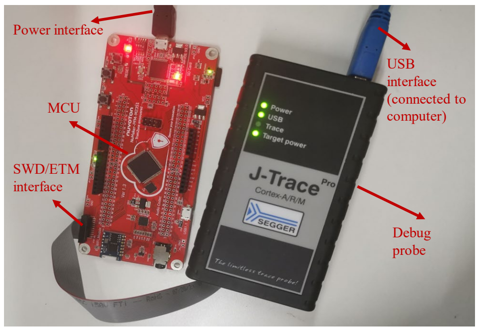


Figure 5: SyzTrust setup in fuzzing a target TEE implementation on the Nuvoton M2351 board. The debug probe accesses memory and tracks instruction traces on the MCU via the SWD/ETM interface. It also delivers data to the PC and receives commands from the PC via a USB interface.

6. Evaluation

In this section, we comprehensively evaluate the SyzTrust, demonstrating its effectiveness in fuzzing IoT Trusted OSes. First, we evaluate the overhead breakdown of SyzTrust. Second, we conduct experiments exploring the effectiveness of our designs. Third, we examine SyzTrust's state variable inference capabilities. Finally, we apply SyzTrust on fuzzing three real-world IoT Trusted OSes and introduce their vulnerabilities. In summary, we aim to answer the following research questions:
RQ1: What is the overhead breakdown of SyzTrust? (Section 6.2)
RQ2: Is SyzTrust effective for fuzzing IoT Trusted OSes? (Section 6.3)
RQ3: Is the state variable inference method effective, and are the inferred state variables expressive? (Section 6.4)
RQ4: How vulnerable are the real-world Trusted OSes from different IoT vendors from SyzTrust's results? (Section 6.5)

6.1. Experimental Setup

Target Trusted OSes. We evaluate SyzTrust on three Trusted OSes designed for IoT devices, mTower, TinyTEE, and Link TEE Air. The reason for selecting these targets is as follows. mTower and TinyTEE both provide the standard APIs following the GP TEE Internal Core API specification. In addition, they are developed by two leading IoT vendors, Samsung and Tsinglink Cloud (which serve more than 30 downstream IoT manufacturers, including China TELECON and Panasonic), respectively. Link TEE Air is a proprietary Trusted OSes developed by Ali Cloud for SyzTrust to evaluate its built-in support of closed-source and proprietary Trusted OSes. Moreover, the three targets have been adopted in a number of IoT devices and their security vulnerabilities have a practical impact. In this paper, we evaluate mTower v0.3.0, TinyTEE v3.0.1, Link TEE Air v2.0.0, and each of them is the latest version during our experiments.

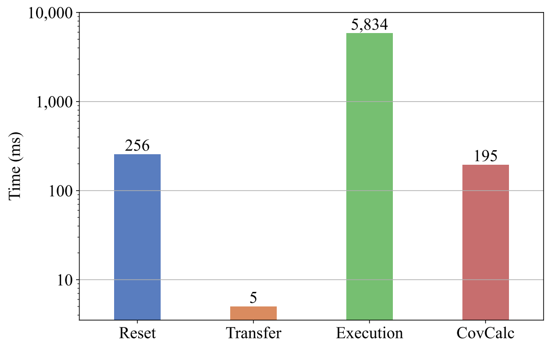


Figure 6: Overhead breakdown of SyzTrust.

Experiment settings. We perform our evaluation under the same experiment settings: a personal computer with 3.20GHz i7-8700 CPU, 32GB RAM, Python 3.8.2, Go version 1.14, and Windows 10.
Evaluation metrics. We evaluate SyzTrust with the following three aspects. (1) We measure the branch coverage to evaluate the capability of exploring codes, which is widely used in recent research [9] [10]. The branch calculation is introduced in Section 4.3. (2) To evaluate the capability of exploring the deep state, we measure the state coverage and the syscall sequence length of each fuzzer. Specifically, we consider the value combinations of state variables as a state and measure the number of different states. (3) We count the number of unique vulnerabilities. SyzTrust relies on the built-in exception handling mechanism to detect abnormal behaviors of Trusted OSes. We explore the dedicated fault status registers [36] to identify the `HardFault` exception in concerns. These exceptions indicate critical system errors and thus can be used as a crash signal [21]. For the crashes, we reproduce them and report their stack traces by CmBackTrace to track and locate error codes. We filter stack traces into unique function call sequences to collect the explored unique bugs on target programs, which are widely used for deduplication in the CVE dataset [37] and debugging for vendors. Following best practices, we extract the top three function calls in the stack traces to de-duplicate bugs [38], [39]. We then analyze the root cause of the bugs manually.
State transition analysis. We further develop a script to automatically construct a state transition tree, which helps visually understand the practical meaning of the state variables. Specifically, we utilize the state hashes calculated based on state variables to present the states of Trusted OSes and take the syscall sequences as state transition labels.

6.2. Overhead Breakdown (RQ1)

We measure the execution time of sub-processes of SyzTrust to assess its impact on the overall overhead. For each fuzzing round, SyzTrust performs the following sub-processes: (1) `Reset` means the time spent on the MCU resetting. (2) `Transfer` means the time spent on the manager engine sending the test case. (3) `Execution` means the time spent on the execution engine executing the test case. In the meanwhile, the debug probe records the raw instruction trace packets and state variable values and transfers them to the PC via RTT. (4) `CovCalc` means the time spent on the hardware-assisted controller decoding the collected raw trace packets and calculating the branch coverage. In the meanwhile, the hardware-assisted controller calculates the state hashes based on the state variable values. We only evaluate the sub-processes that are related to interacting with the resource-limited MCU, which mainly determines the speed of SYZTRUST. As introduced in Section 4.3, the subprocesses of `Transfer`, `Execution`, and `CovCalc` are designed to run in parallel.

The results are shown in Figure 6, from which we have the following conclusions. First, the overheads of `Reset`, `Transfer`, and `CovCalc` in SYZTRUST are relatively low. Thus, using `Reset` to mitigate the false positive coverage and vulnerabilities is acceptable. Second, `Execution` takes the most time. Test cases for IoT Trusted OSes include 14.4 syscalls on average and are complex. In addition, for `Execution`, the Nuvoton M2351 board used in our manuscript has no Embedded Trace Buffer (ETB), SYZTRUST utilizes the debug probe to stream all the ETM trace data to the host PC in real time. We conducted a 48-hour fuzzing and found the peak tracing speed is 168 KB/s, and 92% of the trace files for single syscalls are smaller than 2KB. In conclusion, the ETM tracing takes less time than the syscall execution.

RQ1: It takes SYZTRUST 6,290 ms on average to complete a test case and to collect its feedback. The subprocess of executing a test case on the MCU takes the most time, while the orchestration and analysis take only roughly 1% of the overall time.

6.3. Effectiveness of SYZTRUST (RQ2)

In this section, we evaluate SYZKALLER, SYZTRUST_BASIC, SYZTRUST_STATE, and SYZTRUST_FSTATE on mTower to explore the effectiveness of our designs, with each experiment running for 48 hours. To measure our new scheduling tasks and composite feedback mechanism, we construct two prototypes of SYZTRUST only with the new scheduling tasks and only with the composite feedback mechanism, which are named SYZTRUST_BASIC and SYZTRUST_FSTATE, respectively. In addition, to measure the necessity of our state variable inference, we construct a prototype that considers the complete buffer values of two state handles as state variables named SYZTRUST_STATE. We evaluate three prototypes of SYZTRUST against the state-of-the-art fuzzer SYZKALLER, which has found a large number of vulnerabilities on several kernels and is actively maintained. To reduce the randomness, we repeat all experiments ten times. The results are shown in Figure 7 and Table 1.

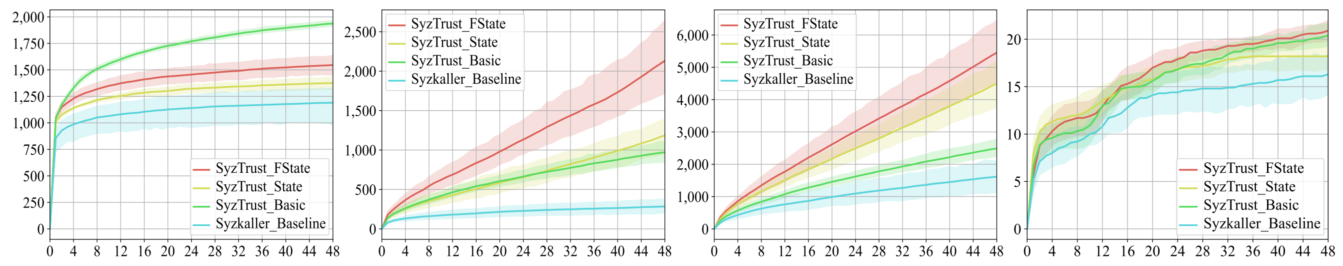
Branch coverage is calculated in terms of LCSAJ basic block number introduced in Section 4.3. SYZKALLER triggers 1,191 branches on average and is exceeded by all three versions of SYZTRUST shown in Figure 7. Among

TABLE 1: The unique vulnerabilities found by SYZKALLER, SYZTRUST_BASIC, SYZTRUST_STATE, and SYZTRUST_FSTATE in 48 hours. The vulnerabilities whose IDs are marked with * have already received CVEs.

Vul. ID	Syzkaller-Baseline	SyzTrust-Basic	SyzTrust-State	SyzTrust-FState
1*	●	●	●	●
2*	●	●	●	●
3	○	●	○	●
4	○	○	●	○
5*	●	●	●	●
6	○	○	●	○
7*	○	○	●	●
8*	●	●	●	●
9*	●	●	●	●
10	○	○	●	●
11	●	○	○	●
12*	●	○	●	●
13*	●	●	●	●
14*	●	●	●	●
15*	●	●	●	●
16	●	●	●	●
17	○	○	○	●
18	○	○	○	●
19	○	○	○	●
20	●	●	●	●
21	●	●	●	●
22	●	●	●	●
23	●	●	●	●
24	●	●	●	●
25	●	●	●	●
26	●	●	●	●
27	●	●	●	●
28	●	●	●	●
29	●	●	●	●
30	●	●	●	●
31	●	●	●	●
32	●	●	●	●
33	○	●	○	●
34	○	●	●	●
35	●	●	●	●
36	○	○	●	●
37	○	●	●	●
38	●	●	○	○
Total	**26**	**28**	**31**	**35**

them, SYZTRUST_BASIC archives the highest code coverage among the four fuzzers with 1,983 branches, which shows the effectiveness of our new scheduling tasks and the extended syscall templates. When state coverage feedback is integrated into SYZTRUST, the branch coverage explored by the SYZTRUST_STATE and SYZTRUST_FSTATE is lower than SYZTRUST_BASIC. It is because their adopted composite feedback mechanisms drive them to preserve more seeds that trigger new states, which might have no contribution to the code coverage. Given the same evaluation time, a certain percentage of time is assigned to mutating and testing such seeds that trigger new states, and the code coverage growth will result in slower growth. Thus, we additionally perform a long-time fuzzing experiment, and the result shows that SYZTRUST_FSTATE can achieve 1,984 branches in 74 hours.

Among the four fuzzers, SYZTRUST_FSTATE triggers the most states with 2,132 states on average, and SYZKALLER triggers the least states with 284 states on average shown in Figure 7b. SYZTRUST_STATE has similar performances on the state growth with SYZTRUST_BASIC. It is because SYZTRUST_STATE spends lots of time exploring the test cases that trigger new values of non-state variables since SYZTRUST_STATE utilizes the two whole handles' values to present state. To further illustrate this



(a) Branch coverage. (b) States (based on variables). (c) States (based on handles). (d) Unique vulnerabilities.

Figure 7: The branch coverage growth, state numbers growth, and unique vulnerability growth discovered by SYZKALLER, SYZTRUST_BASIC, SYZTRUST_STATE, and SYZTRUST_FSTATE.

deduction, we present Figure 7c, where we calculate the growth of the states based on the whole handle values. In such settings, SYZTRUST_STATE triggers more states than SYZTRUST_BASIC and SYZKALLER. However, these states are not expressive and effective for guiding the fuzzer to trigger more branch coverage and vulnerabilities.

The exploration space of unique vulnerabilities triggered by SYZKALLER with 16 vulnerabilities on average is fully covered and exceeded by the three versions of SYZTRUST shown in Figure 7. Among them, SYZTRUST_FSTATE detects 21 vulnerabilities on average, which achieves the best vulnerability-finding capability. SYZTRUST_STATE has similar performances on vulnerability detection with SYZTRUST_BASIC and they detect 20 vulnerabilities on average.

In addition, Table 1 shows the unique vulnerabilities detected by SYZKALLER and the three versions of SYZTRUST in ten trials during 48 hours. The vulnerability ID in Table 1 is consistent with Table 6 in Appendix C. First, SYZTRUST finds all the unique vulnerabilities that SYZKALLER finds. Using the currently assigned CVE as ground truth, SYZTRUST detected more CVEs than SYZKALLER. Second, SYZTRUST_FSTATE finds the most vulnerabilities and finds eight vulnerabilities that SYZKALLER and SYZTRUST_BASIC cannot find. The eight vulnerabilities are all triggered by syscall sequences whose lengths are more than 10, which indicates they are triggered in a deep state. For instance, the vulnerability of ID 7 occurs when the Trusted OS enters the $key\_set\&initialized$ state after a MAC function is configured and the function $TEE\_MACUpdate$ is invoked with an excessive size value of "chunkSize". SYZTRUST_FSTATE can detect more vulnerabilities, which is benefited from our composite feedback mechanism. Specifically, the fuzzer preserves the seeds that trigger new states and then can detect more vulnerabilities by exploring these seeds. In summary, this evaluation reveals two observations. First, although coverage-based fuzzer achieves high coverage effectively, their vulnerability detection capability will be limited when testing stateful systems. Second, for stateful systems, understanding their internal states and utilizing the state feedback to guide fuzzing will be effective in finding more vulnerabilities.

TABLE 2: The number of state variables inferred by SYZTRUST. False postive is denoted as FP.

Target	Handle	Number	FP	Precision
mTower	$TEE\_ObjectHandle$	11	1	87.5%
	$TEE\_OperationHandle$	13	2	
TinyTEE	$TEE\_ObjectHandle$	13	3	82.6%
	$TEE\_OperationHandle$	10	1	
OP-TEE	$TEE\_ObjectHandle$	10	1	87.0%
	$TEE\_OperationHandle$	13	2	
Link TEE Air	$context(AES)$	6	2	71.4%
	$context(Hash)$	8	2	

RQ2: The design of SYZTRUST is effective as the three versions of SYZTRUST outperform SYZKALLER in terms of code and state coverage and number of detected vulnerabilities. New task scheduling with extended syscall templates significantly improves the fuzzer's code exploration, and the composite feedback mechanism helps trigger more states and detect more vulnerabilities.

6.4. State Variable Inference (RQ3)

As for state variable inference evaluation, we first evaluate the precision of our state variable inference method. Second, we utilize the executed syscall sequences and their state hashes to construct a state transition tree and present an example to show the expressiveness of our state variables.

For the precision evaluation, we manually analyze the usage of the inferred state variables in the Trusted OSes. We check if a state variable is used in condition statements to control the execution context. As for mTower and OP-TEE, we obtain their source codes and manually read the state variable-related codes. As for TinyTEE and Link TEE Air, we invite five experts with software reverse engineering experiences to manually analyze their binary codes.

Table 2 shows the results. For Trusted OSes, including a proprietary one, our active state variable inference method is effective and achieves 83.3% precision on average. These validated state variables are expressive and meaningful, including `algorithm`, `operationClass` (description identifier of operation types, e.g., CIPHER, MAC), `mode` (description identifier of operation, e.g., ENCRYPT, SIGN), and `handleState` (describing the current state of the

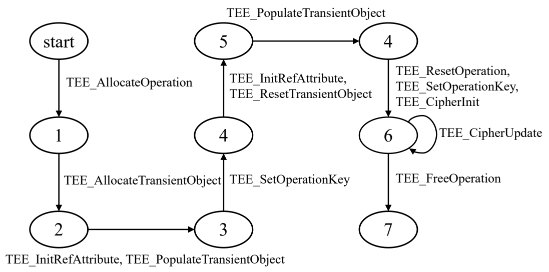

start
TEE_AllocateOperation
1
TEE_AllocateTransientObject
2
TEE_InitRefAttribute, TEE_PopulateTransientObject
3
TEE_SetOperationKey
4
TEE_InitRefAttribute, TEE_ResetTransientObject
5
TEE_PopulateTransientObject
4
TEE_ResetOperation, TEE_SetOperationKey, TEE_CipherInit
6
TEE_CipherUpdate
TEE_FreeOperation
7

Figure 8: An example of state transition path from the constructed state transition tree in mTower.

operation, e.g., an operation key has been set). The false positive state variables are of two types. One is some variables that indicate the length of several specific buffers, e.g., digest length, and have specific values. Another type is several buffers that do not likely have changeable values. Both of them generate a few false positive new states and have little impact on the fuzzing procedure.

To evaluate state variables' expressiveness, we construct a state transition tree to help visualize their practical meanings. As shown in Figure 8, we present an example state transition path from our constructed state transition tree. We replace the state hashes with numerical values in the node label to enhance the readability. Below is the meaning of the example state transition tree. First, `TEE_OperationHandle` and `TEE_ObjectHandle` are allocated by `TEE_AllocateOperation` and `TEE_AllocateTriansientObject`, respectively. Once a handle is allocated, the state transition is triggered. Second, `TEE_ObjectHandle` loads a dummy key by executing `TEE_InitRefAttribute` and `TEE_PopulateTransientObject`. The dummy key is then loaded into the `TEE_OperationHandle` by `TEE_SetOperationKey`. Then state node 4 is triggered, indicating mTower is in the `key_set` state, which is consistent with the GP TEE Internal Core API specification. Third, `TEE_ObjectHandle` is reset and `TEE_OperationHandle` loads an encryption key by executing the same syscalls of the second procedure with a valid key. Fourth, mTower turns to a new state after `TEE_CipherInit`, which is specified as `key_set & initialized` state in the specification. This state means that mTower loads the key and the initialization vector for encryption. When mTower is in the `key_set & initialized`, mTower processes the ciphering operation on provided buffers by `TEE_CipherUpdate`. Finally, mTower turns to a new state after it frees all the encryption configurations by `TEE_FreeOperation`. As a result, our state presentation mechanism truthfully reflects the workflow of symmetric encryption.

RQ3: On average, our active state variable inference method is effective and achieves 83.3% precision on average. In addition, from the state transition tree, the inferred state variables are meaningful.

TABLE 3: The number of unique vulnerabilities, branches and states found by SyzTrust in 90 hours.

Target	Unique bugs	Branches	States
mTower	38	2,105	3,994
TinyTEE	13	1,072	2,908
Link TEE Air	19	10,710	182,324

6.5. Real World Trusted OSes (RQ4)

We apply SyzTrust on mTower from Samsung, TinyTEE from Tsinglink Cloud, and Link TEE Air for 90 hours. The results are shown in Table 3. The branch and state coverage explored by SyzTrust on Link TEE Air is relatively low because Link TEE Air has a more complicated and large code base. As for vulnerability detection, a total of 70 vulnerabilities are found by SyzTrust, and 28 of them are confirmed, while vendors are still investigating the remaining bug reports. We have reported confirmed vulnerabilities to MITRE, and 10 of them have already been assigned CVEs.

We categorize vulnerabilities into seven types following *the Common Weakness Enumeration (CWE) List* [40] and we have the following conclusions. First, the Trusted OSes suffer from frequent null/untrusted pointer dereference vulnerabilities. They lack code for validating supplied input pointers. When a TA tries to read or write a malformed pointer by invoking Trusted OS syscalls, a crash will be triggered, resulting in a DoS attack. Even worse, carefully designed pointers provided by a TA to the syscall can compromise the integrity of the execution context and lead to arbitrary code execution, posing a significant risk. The problem is severe since Trusted OSes frequently rely on pointers to transfer complex data structures, which are used in cryptographic operations. Second, the Trusted OSes allocate resources without checking bounds. They allow a TA to achieve excessive memory allocation via a large len value. Since the IoT TEEs are resource-limited, these vulnerabilities may easily cause denial of service. Third, the Trusted OSes suffer from buffer overflow vulnerabilities. They missed checks for buffer accesses with incorrect length values and allowed a TA to trigger a memory overwriting, DoS, and information disclosure. We discuss the mitigation in Section 7 and list the vulnerabilities with their root cause analysis found on mTower and TinyTEE in Table 6 in Appendix C.

Case study 1: buffer overflow (CVE-2022-35858). SyzTrust identifies a stack-based buffer overflow vulnerability in `TEE_PopulateTransientObject` syscall in mTower, which has 7.8 CVSS Score according to *CVE Details* [41]. Specifically, the `TEE_PopulateTransientObject` syscall creates a local array directly using the parameter `attrCount` without checking its size. Once `TEE_PopulateTransientObject` is invoked with a large number in `attrCount`, a memory overwrite will be triggered, resulting in Denial-of-Service (DoS), information leakage, and arbitrary code execution.

Case study 2: null pointer dereference (CVE-2022-40759). SyzTrust uncovers a null pointer dereference in `TEE_MACCompareFinal` in mTower. A DoS attack can be triggered by invoking `TEE_MACCompareFinal` with a null pointer for the parameter operation. This bug is hard to trigger by traditional fuzzing, as it requires testing on a specific syscall sequence to enter a specific state. SyzTrust can effectively identify such syscall sequences that trigger new states and prioritize testing them. In this way, this syscall sequence can be fuzzed to cause a null pointer dereference vulnerability; otherwise, it will be discarded.

RQ4: mTower, TinyTEE and Link TEE Air are all vulnerable. SyzTrust identifies 38 vulnerabilities on mTower, 13 vulnerabilities on TinyTEE, and 19 vulnerabilities on Link TEE Air, resulting in 10 CVEs.

7. Discussion

Ethic. We pay special attention to the potential ethical issues in this work. First, we obtain all the tested Trusted OSes from legitimate sources. Second, we have made responsible disclosure to report all vulnerabilities to IoT vendors.

Lessons. Based on our evaluations, we provide several security insights about the existing popular IoT Trusted OSes. First, mTower and TinyTEE are similar to OP-TEE, an open-source TEE implementation designed for Cortex-A series MCUs, and they have several similar vulnerabilities. For instance, they both implement vulnerable syscalls `TEE_Malloc` and `TEE_Realloc`, which allow an excessive memory allocation via a large value. In a resource-constrained MCU, such implementations can cause a crash and result in a DoS attack. Thus, the security principles should be rethought to meet the requirements of the IoT scenario. As a mitigation, we suggest adding checks when allocating memory; this suggestion is adopted by Samsung and TsingLink Cloud. Second, for the null/untrusted pointer dereference and buffer overflow vulnerabilities, the input pointers and critical parameters should be especially carefully checked. For the untrusted pointer dereferences on handlers, in addition to checking if the pointer address is valid, we suggested adding a handler list to mark if they are allocated or released. Third, we found mTower and TinyTEE implement several critical syscalls provided by Trusted OS in non-privileged codes, which exposes a number of null/untrusted pointer dereference vulnerabilities. Thus, a TA can easily reach these syscalls and damage the Trusted OSes. Even worse, mTower and TinyTEE do not have memory protection mechanisms, such as ASLR (Address Space Layout Randomization). Trusted OSes and TAs are all loaded into the same fixed address in the virtual address space. The above two problems make the exploitation of Trusted OSes easier. However, implementing privileged codes and memory protection mechanisms requires additional overhead, which may be unacceptable for IoT devices. We suggest that the downstream TA developers should be aware of it and carefully design their TAs to mitigate this security risk, or lightweight Trusted OS implementations should be designed to minimize the overhead brought by security mechanisms.

Limitations and future work. While SyzTrust provides an effective way to fuzz IoT Trusted OSes, it also exposes some opportunities for future research. First, our current prototype of SyzTrust primarily targets Trusted OSes following GP TEE Internal Core API specification. It has built-in support for alternative Trusted OSes, including proprietary ones, requiring certain modifications and configurations. We demonstrated this flexibility by extending to a proprietary Trusted OS and plan to extend SyzTrust for broader applicability. Second, SyzTrust assumes that a TA can be installed in the Trusted OS for assisting fuzzing and ARM ETM is enabled for collecting the traces. In the case of certain Trusted OSes, such as ISEE-M, which are developed and used within a relatively tight supply chain, we will need to engage with the providers of these Trusted OSes to help assess the security of their respective Trusted OSes. Finally, SyzTrust targets the Trusted OS of TEE, leaving several security aspects of TEEs to be studied, e.g., TAs and the interaction mechanism between peripherals and the Trusted OSes.

8. Related Work

TEE vulnerability detection. Several researchers have studied and exploited vulnerabilities in TrustZone-based TEEs. Marcel *et al.* reverse-engineer HUAWEI's TEE components and present a critical security review [42]. Some researchers studied the design vulnerabilities of TEE components [43], [44], such as Samsung's TrustZone keymaster [45] and the interaction between the secure world and the normal world [46]. Recently, Cerdeira *et al.* analyzed more than 200 bugs in TrustZone-assisted TEEs and presented a systematic view [16]. Since those works require manual efforts for vulnerability detection or only provide an automatic tool to target a specific vulnerability, some literature works on automatically testing the TEEs. Some works try to apply other analysis tools, e.g., utilizing concolic execution [47] and fuzzing. A number of TA fuzzing tools are developed, such as TEEzz [9], PartEmu [10], Andrey's work [48] and Slava's work [49]. On the contrary, Trusted OS fuzzing receives little attention. To the best of our knowledge, the only tool OP-TEE Fuzzer [50] is for open-source TEEs, which is not applicable to closed-source IoT TEEs.

ETM-based fuzzing. Firmware analysis primarily adopts two approaches: on-device testing [51], [52] and rehosting [53], [54]. For on-device testing, a few ETM-assisted analysis methods have been proposed for ARM platforms [55], [56]. For instance, Ninjia [24] utilizes ETM to analyze malware transparently. HApper [25] and NScope [29] utilize ETM to unpack Android applications and analyze the Android native code. Recently, two studies have integrated ETM features into fuzzing projects. One is AFL++ CoreSight mode [57], which targets the applications running on ARM Cortex-A platforms. Another is μAFL [26] to fuzz Linux peripheral drivers. However, these works have different purposes. AFL++ CoreSight mode and μAFL focus on the application, whereas SyzTrust focuses on Trusted OSes for IoT devices and incurs many challenges due to the constrained resource and inability to instrumentation. Moreover, SyzTrust proposes a novel fuzzing framework and the state-aware feature to effectively test the stateful Trusted OSes. Consequently, SyzTrust is a novel hardware-assisted fuzzing approach proposed in this paper.

State-aware fuzzing. Recently, state-aware fuzzing has emerged and gained the attention of the research community. To understand the internal states of target systems, existing studies utilize the response code of protocol servers [58], [59] or apply model learning [60], [61] to identify the server's states. StateInspector [11] utilizes explicit protocol packet sequences and run-time memory to infer the server's state machine. However, they are not applicable to software and OSes since software and OSes do not have such response codes or packet sequences. IJON [62] proposes an annotation mechanism that allows the user to infer the states during the fuzzing procedure manually. After that, numbers of studies work on automatically inferring the state of software and OSes [63], [64], [65], such as StateFuzz [13], SGFUZZ [66], and FUZZUSB [12]. However, these studies either require the source codes of targets or precise dynamic instrumentation tools. Even worse, their targets are Linux kernels, protocols, and drivers, and their intuitions and observations are not suitable for IoT Trusted OSes.

9. Conclusion

We present SyzTrust, the first automated and practical fuzzing system to fuzz Trusted OSes for IoT devices. We evaluate the effectiveness of the SyzTrust on the Nuvoton M2351 board with a Cortex M23, and the results show SyzTrust outperforms the baseline fuzzer Syzkaller with 66% higher code coverage, 651% higher state coverage, and 31% improved vulnerability-finding capability. Furthermore, we apply SyzTrust to evaluate real-world IoT Trusted OSes from three leading IoT vendors and detect 70 previously unknown vulnerabilities with security impacts. In addition, we present the understanding of Trusted OS vulnerabilities and discuss the limitation and future work. We believe SyzTrust provides developers with a powerful tool to thwart TEE-related vulnerabilities within modern IoT devices and complete the current TEE fuzzing scope.

Acknowledgments

We sincerely appreciate our shepherd and all the anonymous reviewers for their insightful comments. This work was partly supported by NSFC under No. U1936215, the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCHA202001, the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), SNSF PCEGP2_186974, ERC StG 850868, Google Scholar Award, Meta Faculty Award, and China Scholarship Council.

References

[1] GlobalPlatform, "GlobalPlatform TEE spec adoption to reach 10 billion," https://globalplatform.org/latest-news/globalplatform-tee-spec-adoption-to-reach-10-billion.

[2] Samsung, "mTower," https://github.com/Samsung/mTower.

[3] TsinglinkCloud, "Qinglian Cloud's TinyTEE," https://www.qinglianyun.com/Front/Secure/safe.

[4] A. Cloud, "About Alibaba Cloud Link TEE," https://iot.aliyun.com/products/tee.

[5] L. Luo, Y. Zhang, C. Zou, X. Shao, Z. Ling, and X. Fu, "On runtime software security of trustzone-m based iot devices," in *Proceedings of GLOBECOM 2020-2020 IEEE Global Communications Conference*, 2020.

[6] G. Beniamini, "TrustZone kernel privilege escalation," http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html.

[7] G. Beniamini, "Hijacking the linux kernel from QSEE," https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html.

[8] G. Beniamini, "Extracting Qualcomm's keymaster keys - breaking android full disk encryption," https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html.

[9] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer, "TEEzz: Fuzzing Trusted Applications on COTS Android devices," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2023.

[10] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "PARTEMU: Enabling dynamic analysis of real-world TrustZone software using emulation," in *Proceedings of USENIX Security Symposium (SEC)*, 2020.

[11] C. McMahon Stone, S. L. Thomas, M. Vanhoef, J. Henderson, N. Bailluet, and T. Chothia, "The closer you look, the more you learn: A grey-box approach to protocol state machine learning," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2265–2278.

[12] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. Butler, A. Bianchi, and D. J. Tian, "FUZZUSB: Hybrid stateful fuzzing of USB gadget stacks," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2022.

[13] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "StateFuzz: System call-based state-aware linux driver fuzzing," in *Proceedings of USENIX Security Symposium (SEC)*, 2022.

[14] Wikipedia, "Trusted execution environment," https://en.wikipedia.org/wiki/Trusted_execution_environment.

[15] GlobalPlatform, "TEE Internal Core API Specification v1.3.1," https://globalplatform.org/specs-library/tee-internal-core-api-specification.

[16] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2020.

[17] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *Proceedings of USENIX Security Symposium (SEC)*, 2017.

[18] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "ASSURED: Architecture for secure software update of realistic embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2290–2300, 2018.

[19] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares, "IIoTEED: An enhanced, trusted execution environment for industrial IoT edge devices," *IEEE Internet Computing*, vol. 21, no. 1, pp. 40–47, 2017.

[20] A. Fitzek, F. Achleitner, J. Winter, and D. Hein, "The ANDIX research OS—ARM TrustZone meets industrial control systems security," in *Proceedings of IEEE 13th International Conference on Industrial Informatics (INDIN)*, 2015.

[21] A. Community, "Cortex-M23 and Cortex-M33 - security foundation for billions of devices," https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/cortex-m23-and-cortex-m33---security-foundation-for-billions-of-devices.

[22] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM computing surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[23] A. Developer, "ARM coresight architecture specification version 3.0," https://developer.arm.com/documentation/ihi0029/e.

[24] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *Proceedings of USENIX Security Symposium (SEC)*, 2017.

[25] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, and M. H. Au, "Happer: Unpacking android apps via a hardware-assisted approach," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2021.

[26] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "μAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[27] Google, "syzkaller - kernel fuzzer," https://github.com/google/syzkaller.

[28] Linaro, "Open portable trusted execution environment," https://www.op-tee.org.

[29] H. Zhou, S. Wu, X. Luo, T. Wang, Y. Zhou, C. Zhang, and H. Cai, "NCScope: hardware-assisted analyzer for native code in android apps," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 629–641.

[30] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 633–645.

[31] D. F. Yates and N. Malevris, "The effort required by LCSAJ testing: an assessment via a new path generation strategy," *Software Quality Journal*, vol. 4, no. 3, pp. 227–242, 1995.

[32] "J-Link RTT - real time transfer," https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer.

[33] T. Zhu, "CmBacktrace: ARM Cortex-M series MCU error tracking library," https://github.com/armink/CmBacktrace.

[34] T. L. Kernel, "The kernel address sanitizer (KASAN)," https://www.kernel.org/doc/html/v5.0/dev-tools/kasan.html.

[35] E. Styger, "Finding memory bugs with Google address sanitizer (ASAN) on microcontrollers," https://mcuoneclipse.com/2021/05/31/finding-memory-bugs-with-google-address-sanitizer-asan-on-microcontrollers.

[36] ARM, "Arm Cortex-M23 Devices Generic User Guide r1p0 - Fault Handling," https://developer.arm.com/documentation/dui1095/a/The-Cortex-M23-Processor/Fault-handling.

[37] "CVE details," https://www.cvedetails.com.

[38] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[39] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers." in *USENIX Security Symposium*, 2021, pp. 2777–2794.

[40] CWE, "CWE list version 4.9," https://cwe.mitre.org/data/index.html.

[41] "NVD, CVE: Common vulnerabilities and exposures," https://cve.mitre.org/.

[42] M. Busch, J. Westphal, and T. Mueller, "Unearthing the TrustedCore: A critical review on Huawei's trusted execution environment," in *Proceedings of 14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[43] K. Ryan, "Hardware-backed heist: Extracting ECDSA keys from qualcomm's trustzone," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[44] B. Zhao, S. Ji, X. Zhang, Y. Tian, Q. Wang, Y. Pu, C. Lyv, and R. Beyah, "UVSCAN: Detecting third-party component usage violations in IoT firmware," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[45] A. Shakevsky, E. Ronen, and A. Wool, "Trust dies in darkness: Shedding light on samsung's TrustZone keymaster design," in *Proceedings of USENIX Security Symposium (SEC)*, 2022.

[46] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "BOOMERANG: Exploiting the semantic gap in trusted execution environments." in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2017.

[47] M. Busch and K. Dirsch, "Finding 1-day vulnerabilities in trusted applications using selective symbolic execution," in *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA*, 2020.

[48] A. Andrey, "Launching feedback-driven fuzzing on trustzone tee," https://zeronights.ru/wp-content/themes/zeronights-2019/public/materials/5_ZN2019_andrej_akimovLaunching_feedbackdriven_fuzzing_on_TrustZone_TEE.pdf.

[49] S. Makkaveev, "The road to Qualcomm TrustZone apps fuzzing," https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing.

[50] Riscure, "OP-TEE fuzzer," https://github.com/Riscure/optee_fuzzer.

[51] P. Liu, S. Ji, X. Zhang, Q. Dai, K. Lu, L. Fu, W. Chen, P. Cheng, W. Wang, and R. Beyah, "IFIZZ: Deep-state and efficient fault-scenario generation to test IoT firmware," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 805–816.

[52] P. Liu, S. Ji, L. Fu, K. Lu, X. Zhang, J. Qin, W. Wang, and W. Chen, "How iot re-using threatens your sensitive data: Exploring the user-data disposal in used IoT devices," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 3365–3381.

[53] H. Peng and M. Payer, "{USBFuzz}: A framework for fuzzing {USB} drivers by device emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2559–2575.

[54] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "{HALucinator}: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1201–1218.

[55] Z. Ning and F. Zhang, "Understanding the security of arm debugging features," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2019.

[56] Z. Ning, C. Wang, Y. Chen, F. Zhang, and J. Cao, "Revisiting arm debugging features: Nailgun and its defense," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[57] A. Moroo and S. Yuichi, "ARMored coresight: Towards efficient binary-only fuzzing," https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html, November 2021.

[58] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[59] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*, 2015.

[60] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2009.

[61] Q. Wang, S. Ji, Y. Tian, X. Zhang, B. Zhao, Y. Kan, Z. Lin, C. Lin, S. Deng, A. X. Liu *et al.*, "MPInspector: A systematic and automatic approach for evaluating the security of IoT messaging protocols," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 4205–4222.

[62] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2020.

[63] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.

[64] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *Proceedings of USENIX Security Symposium (SEC)*, 2021.

[65] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, pp. 1–31, 2022.

[66] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *Proceedings of USENIX Security Symposium (SEC)*, 2022.

[67] P. Cohen, "Winbond partners with Nuvoton, Qinglianyun to release fully integrated reference design for OTA firmware updating)," https://embeddedcomputing.com/technology/iot/winbond-partners-with-nuvoton-qinglianyun-to-release-fully-integrated-reference-design-for-ota-firmware-updating.

[68] Trustonic, "Mircochip first to use Turstonic revolutionary Kinibi-M platform for microcontrollers," https://www.sourcesecurity.com/news/mircochip-turstonic-kinibi-m-microcontrollers-co-1530084457-ga-co-1530085342-ga-npr.1530086842.html.

[69] ARM, "TF-M platforms," https://tf-m-user-guide.trustedfirmware.org/platform/index.html.

[70] Beanpod, "ISEE trusted execution environment platform (Secure OS)," https://www.beanpodtech.com/en/products/.

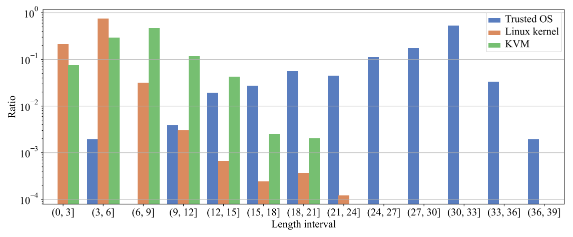


Figure 9: The seed length ratio.

Appendix A. The Motivation of the New Scheduling

To justify our motivation for the new scheduling tasks design for fuzzing the IoT Trusted OS implementation, we compare the seed length ratio when testing Trusted OSes, Linux kernel and KVM. In the evaluation, we utilize SYZKALLER to test mTower, Linux kernel (git checkout 356d82172), and KVM v5.19 for 24 hours, and the results are shown in Figure 9. As shown in Figure 9, the average seed length when fuzzing Trusted OSes is 27.8 while the others are less than 7. In the triage scheduling task, SYZKALLER removes syscalls one by one in a syscall sequence. Then SYZKALLER tests the modified syscall sequences to get the smallest syscall sequence that maintains the same code coverage. For those syscall sequences whose length is more than 30, the triage scheduling tasks probably spend lots of time on removing syscalls and testing the modified syscall sequences. In addition, we count the number of minimized syscalls when fuzzing mTower. In a 48-hours fuzzing, only 56 syscall sequences are minimized, and among them, 18 syscall sequences only are removed with one syscall in the triaging tasks. Thus, SYZTRUST doesn't have to perform triaging tasks since most test cases will not be minimized.

Appendix B. Scope and Scalability of SYZTRUST

We first provide an overview of the major Trusted OSes from leading IoT vendors and use the objective data to validate our assumptions. Then, we justify how to extend SYZTRUST for Cortex-A TEE OSes.

TABLE 4: An overview of the major Trusted OS implementations provided by leading IoT vendors.

Vendor	Trusted OS	Standards	Support (installing TA)	Some of supported devices
Samsung	mTower	GP Standards	●	NuMaker-PFM-M2351
Alibaba	Link TEE Air	Proprietary	●	NuMaker-PFM-M2351
TsingLink Cloud	TinyTEE	GP Standards	●	NuMaker-PFM-M2351/LPC55S69/STM32L562
Beanpod	ISEE-M	GP Standards	●	LPC55S series/GD32W515/STM32L5 series
Trustonic	Kinibi-M	PSA Certified APIs	●	MicroChip SAML11
ARM	TF-M	PSA Certified APIs	●	NuMaker-PFM-M2351, STM32L5, ...

As shown in Table 4, there are six major Trusted OSes for IoT devices, which are widely adopted by the major IoT MCUs [2], [3], [4], [67], [68], [69], [70]. Three of them follow the GP standards, and they all allow TA installation.

TABLE 5: ETM feature on IoT devices.

Manufacturer	Device	Privilige Secure Debug (including ETM)	Debug Authentication Managerment
Nuvoton	NuMaker-PFM-M2351	Enable in default	ICP programming tool
NXP Semiconductors	LPC55S69	Enable in default	Debug credential certificate
STMicroelectronics	STM32L562	Enable in default	STM32CubeProgrammer
GigaDevice	GD32W515	Enable in default	Efuse
MicroChip	SAML11	Enable in default	Extern debugger

In addition, as shown in Table 5, though the devices have multiple debug authentication to disable the privilege secure debug (debug in secure privileged modes), they enable privilege secure debug by default, thereby enabling the ETM feature to collect execution traces. Thus, SYZTRUST can be directly deployed on half of the major Trusted OSes. As for other Trusted OSes, SYZTRUST can be applied with modification introduced in Section 5.

Though our paper focuses on Cortex-M Trusted OS for IoT devices, SYZTRUST can be applied to those Trusted OSes for Cortex-A that allows installing a TA and having their ETM feature enabled. For instance, OP-TEE from Linaro, Link TEE Pro from Ali Cloud, and iTrustee from Huawei meet these assumptions. However, for the Cortex-A Trusted OSes that are developed and employed in a relatively private supply chain, such as QSEE from Qualcomm, we can collaborate with those mobile vendors and help them vet the security of their Trusted OSes.

Appendix C. Vulnerabilites Found by SYZTRUST

We present all the vulnerabilities found by SYZTRUST on mTower, TinyTEE and Link TEE Air with their root cause in Table 6. We update the vulnerabilities disclosure progress in the GitHub link https://github.com/SyzTrust.

Appendix D. Meta-Review

D.1. Summary

This paper presents a fuzzing framework for TEEs on IoT devices. It leverages hardware-assisted features such as Arm ETM to collect traces. It uses the state and code coverage as composite feedback to guide the fuzzer to effectively explore more states.

D.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

D.3. Reasons for Acceptance

1) Creates a New Tool to Enable Future Science. This paper presents a new fuzzing tool that targets IoT Trusted OSes.
2) Identifies an Impactful Vulnerability. Several vulnerabilities were identified, and CVEs are disclosed.
3) Provides a Valuable Step Forward in an Established Field. This paper leverages hardware-assisted features such as Arm ETM to further improve the effectiveness of TEE OS fuzzing on IoT devices.

D.4. Noteworthy Concerns

The proposed fuzzing framework targets Trusted OSes following GP TEE internal API specification. It is assumed that a TA can be installed in the trusted OS for assisting fuzzing. Arm ETM needs to be enabled for collecting the traces.

Appendix E. Response to the Meta-Review

The meta-review notes that our proposed fuzzing framework, SyzTrust, targets trusted OSes following GP TEE Internal API specification. To clarify, SyzTrust also has built-in support for testing alternative Trusted OSes, including proprietary ones, as demonstrated by our extension of SyzTrust to the proprietary "Link TEE Air". We are also extending our SyzTrust prototype to other OSes.

The meta-review notes that SyzTrust assumed that a TA can be installed in the trusted OS for assisting fuzzing and ARM ETM needs to be enabled for collecting the traces. We agree and note that these assumptions align with typical IoT Trusted OS scenarios. SyzTrust focuses on the Trusted OS binaries provided by IoT vendors, delivering security insights for device manufacturers and end users. First, given that IoT device manufacturers often need to implement device-specific TAs, Trusted OS binaries supplied by IoT vendors generally allow TA installation (similar to smartphones where manufacturers can similarly install TAs in the respective TEEs). Second, SyzTrust tests IoT Trusted OSes by deploying them on development boards where ETM is enabled by default. For certain Trusted OSes that are developed and used within a relatively private supply chain, we will need to engage with the providers of these Trusted OSes to help assess the security of their respective Trusted OSes. Appendix B provides a detailed discussion along with supporting data.

TABLE 6: Vulnerabilities detected by SyzTrust.

Vul. ID	Target	Description	Status	Root Cause & Impact
1	mTower	Allocation of resources without limits or throttling	CVE-2022-38155 (7.5 HIGH)	TEE_Malloc allows a TA to achieve excessive memory allocation via a large len value
2	mTower	Allocation of resources without limits or throttling	CVE-2022-40762 (7.5 HIGH)	TEE_Realloc allows a TA to achieve excessive memory allocation via a large len value
3	mTower	Allocation of resources without limits or throttling	Confirmed	TEE_AllocateOperation allows a TA to achieve excessive memory allocation via a large len value
4	mTower	Allocation of resources without limits or throttling	Confirmed	TEE_AllocateTransientObject allows a TA to achieve excessive memory allocation via a large len value
5	mTower	Improper input validation	CVE-2022-40761 (7.5 HIGH)	The function tee_obj_free allows a TA to trigger a denial of service by invoking the function TEE_AllocateOperation with a disturbed heap layout, related to utee_cryp_obj_alloc
6	mTower	Buffer overflow	Reported	A Buffer Access with Incorrect Length Value vulnerability in the TEE_MemMove function allows a TA to trigger a denial of service by invoking the function TEE_MemMove with a ”size” parameter that exceeds the size of ”dest”.
7	mTower	Buffer overflow	CVE-2022-40760 (7.5 HIGH)	A Buffer Access with Incorrect Length Value vulnerability in the TEE_MACUpdate function allows a TA to trigger a denial of service by invoking the function TEE_MACUpdate with an excessive size value of chunkSize.
8	mTower	Buffer overflow	CVE-2022-40757 (7.5 HIGH)	A Buffer Access with Incorrect Length Value vulnerability in the TEE_MACComputeFinal function allows a TA to trigger a denial of service by invoking the function TEE_MACComputeFinal with an excessive size value of messageLen.
9	mTower	Buffer overflow	CVE-2022-40758 (7.5 HIGH)	A Buffer Access with Incorrect Length Value vulnerability in the TEE_CipherUpdate function allows a TA to trigger a denial of service by invoking the function TEE_CipherUpdate with an excessive size value of srcLen.
10	mTower	Buffer overflow	Reported	A Buffer Access with Incorrect Length Value vulnerability in the TEE_DigestDoFinal function allows a TA to trigger a denial of service by invoking the function TEE_DigestDoFinal with an excessive size value of chunkLen.
11	mTower	Buffer overflow	Reported	A Buffer Access with Incorrect Length Value vulnerability in the TEE_DigestUpdate function allows a TA to trigger a denial of service by invoking the function TEE_DigestUpdate with an excessive size value of chunkLen.
12	mTower	Missing release of memory after effective lifetime	CVE-2022-35858 (7.8 HIGH)	The TEE_PopulateTransientObject and __utee_from_attr functions allow a TA to trigger a memory overwrite, denial of service, and information disclosure by invoking the function TEE_PopulateTransientObject with a large number in the parameter attrCount.
13	mTower	NULL pointer dereference	CVE-2022-40759 (7.5 HIGH)	TEE_MACCompareFinal contains a NULL pointer dereference on the parameter operation
14	mTower	NULL pointer dereference	CVE-2022-36621 (7.5 HIGH)	TEE_AllocateTransientObject contains a NULL pointer dereference on the parameter object
15	mTower	NULL pointer dereference	CVE-2022-36622 (7.5 HIGH)	TEE_GetObjectInfo1 contains a NULL pointer dereference on the parameter objectInfo
16	mTower	NULL pointer dereference	Confirmed	TEE_GetObjectInfo contains a NULL pointer dereference on the parameter objectInfo
17	mTower	Untrusted pointer dereference	Reported	Uncertain (provided the PoC to the vendor)
18	mTower	Untrusted pointer dereference	Reported	Uncertain (provided the PoC to the vendor)
19	mTower	Untrusted pointer dereference	Reported	TEE_GetObjectInfo and utee_cryp_obj_get_info functions allow a corruption on the link field of object handle and then a Denial of Service (DoS) will be triggered by invoking the function tee_obj_get
20	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_MACComputeFinal function
21	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_SetOperationKey2 function
22	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_MACUpdate function
23	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_GetOperationInfoMultiple function
24	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_AEEncryptFinal function
25	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_MACInit function
26	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_SetOperationKey function
27	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_ResetOperation function
28	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_DigestUpdate function
29	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_AEDecryptFinal function
30	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_CipherInit function
31	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_FreeOperation function
32	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_DigestDoFinal function
33	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_AllocateOperation function
34	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_FreeTransientObject function
35	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_AEUpdate function
36	mTower	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_ResetOperation function
37	mTower	Untrusted pointer dereference	Reported	Uncertain (provided the PoC to the vendor)
38	mTower	Untrusted pointer dereference	Reported	Uncertain (provided the PoC to the vendor)
39	TinyTEE	Allocation of resources without limits or throttling	Confirmed	TEE_Malloc allows a trusted application to achieve Excessive Memory Allocation via a large len value
40	TinyTEE	Allocation of resources without limits or throttling	Confirmed	TEE_Realloc allows a trusted application to achieve Excessive Memory Allocation via a large len value
41	TinyTEE	NULL pointer dereference	Confirmed	TEE_AllocateTransientObject contains a NULL pointer dereference on the parameter object
42	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_DigestUpdate function
43	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_SetOperationKey function
44	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_SetOperationKey function
45	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_ResetOperation function
46	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_FreeOperation function
47	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_CipherDoFinal function
48	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_CipherInit function
49	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_AllocateOperation function
50	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_AsymmetricSignDigest function
51	TinyTEE	Untrusted pointer dereference	Confirmed	An invalid pointer dereference can be triggered when a TA tries to read a malformed TEE_OperationHandle by the TEE_CipherUpdate function
52	Link TEE Air	NULL pointer dereference	Reported	tee_memcpy calls tee_osa_memcpy, which contains a NULL pointer dereference on the result object
53	Link TEE Air	NULL pointer dereference	Reported	tee_strcpy calls tee_osa_strcpy, which contains a NULL pointer dereference on the result object
54	Link TEE Air	NULL pointer dereference	Reported	tee_memset calls tee_osa_strcpy, which contains a NULL pointer dereference on the result object
55	Link TEE Air	Buffer overflow	Reported	tee_hash_update does not check the size of its second parameter ”size” and calls dev_ioctl, which triggers an invalid memory access with large ”size” value when consecutively copying 64 bytes in tee_osa_memcpy
56	Link TEE Air	NULL pointer dereference	Reported	tee_memmove calls tee_osa_memmove, which contains a NULL pointer dereference on the result object
57	Link TEE Air	NULL pointer dereference	Reported	tee_strcat calls tee_osa_strcat, which contains a NULL pointer dereference on the result object
58	Link TEE Air	Buffer overflow	Reported	tee_hash_digest does not check the size of its third parameter ”size” and calls dev_ioctl, which triggers an heap overflow with large ”size” value and causes an invalid pointer dereference in tee_osa_free
59	Link TEE Air	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a trusted application tries to access an invalid address in the pool_free function called by tee_osa_free and tee_free
60	Link TEE Air	NULL pointer dereference	Reported	tee_strncpy calls tee_osa_strncpy, which contains a NULL pointer dereference on the result object
61	Link TEE Air	NULL pointer dereference	Reported	tee_strcasecmp calls tee_osa_strcasecmp, which contains a NULL pointer dereference on the s1 and s2 object
62	Link TEE Air	NULL pointer dereference	Reported	tee_memcmp calls tee_osa_memcmp, which contains a NULL pointer dereference on the s1 and s2 object
63	Link TEE Air	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a trusted application tries to access an invalid address in the tee_hash_init function
64	Link TEE Air	NULL pointer dereference	Reported	tee_strncat calls tee_osa_strncat, which contains a NULL pointer dereference on the result object
65	Link TEE Air	NULL pointer dereference	Reported	tee_strnlen calls tee_osa_strnlen, which contains a NULL pointer dereference on the s object
66	Link TEE Air	Buffer overflow	Reported	tee_base64_encode does not check the size of its parameters ”src_len” and ”dst”, which triggers a buffer overflow if ”src_len” is larger than the size of ”dst” and ruins the metadata of the next chunk
67	Link TEE Air	NULL pointer dereference	Reported	tee_strlen calls tee_osa_strlen, which contains a NULL pointer dereference on the s object
68	Link TEE Air	NULL pointer dereference	Reported	tee_strcmp calls tee_osa_strcmp, which contains a NULL pointer dereference on the s1 and s2 object
69	Link TEE Air	Buffer overflow	Reported	tee_hash_final does not check the size of its first parameter ”dgst” and calls dev_ioctl, which triggers an heap overflow if the size of ”dgst” is smaller than 48 bytes and causes an invalid pointer dereference in tee_osa_free
70	Link TEE Air	Untrusted pointer dereference	Reported	An invalid pointer dereference can be triggered when a trusted application tries to access an invalid address in the tee_osa_memcpy function called by tee_aes_init