TÄMU: Emulating Trusted Applications at the (GlobalPlatform)-API Layer

Philipp Mao[†], Li Shi[‡], Marcel Busch[†*], Mathias Payer[†]
[†]*EPFL,* [‡]*DARKNAVY*

***Abstract*—Mobile devices rely on Trusted Execution Environments (TEEs) to execute security-critical code and protect sensitive assets. This security-critical code is modularized in components known as Trusted Applications (TAs). Vulnerabilities in TAs can compromise the TEE and, thus, the entire system. However, the closed-source nature and fragmentation of mobile TEEs severely hinder dynamic analysis of TAs, limiting testing efforts to mostly static analyses.**

This paper presents TÄMU, a rehosting platform enabling dynamic analysis of TAs, specifically fuzzing and debugging, by interposing their execution at the API layer. To scale to many TAs across different TEEs, TÄMU leverages the standardization of TEE APIs, driven by the GlobalPlatform specifications. For the remaining TEE-specific APIs not shared across different TEEs, TÄMU introduces the notion of *greedy high-level emulation*, a technique that allows prioritizing manual rehosting efforts based on the potential coverage gain during fuzzing. We implement TÄMU and use it to emulate 67 TAs across four TEEs. Our fuzzing campaigns yielded 17 zero-day vulnerabilities across 11 TAs. These results indicate a deficit of dynamic analysis capabilities across the TEE ecosystem, where not even vendors *with source code* unlocked these capabilities for themselves. TÄMU promises to close this gap by bringing effective and practical dynamic analysis to the mobile TEE domain.

1. Introduction

Trusted Applications (TAs) run in the user space of the Trusted Execution Environment (TEE) Operating System (TOS). TAs encapsulate security-critical functionality such as fingerprint verification, digital rights management, or payment services, which is exposed to the untrusted normal world. Because TAs communicate with the normal world and process untrusted input, their security is paramount. Bugs in TAs have repeatedly been used to achieve full-device compromise [7], [8], [9], [12], [11], [21], [22], [5], [25], [33], [17].

Dynamic analysis, such as fuzzing or debugging, helps by proactively analyzing TAs to discover bugs. However, due to TAs running in the TEE, dynamic analysis on-device, with introspection capabilities that go beyond the TEE log output, is restricted as only correctly signed code can run in the TEE. The alternative to unlocking dynamic analysis capabilities for security researchers is emulation. However, emulating TAs is challenging due to the heterogeneity of real-world TEE implementations. TEEs are completely proprietary with vastly different TOSes and runtime environments. For example, Xiaomi's MiTEE uses a Fuchsia-based microkernel, while TEEGris by Samsung is a monolithic kernel with partial POSIX support mixed with custom system calls. These TOSes have drastically different system call interfaces. Emulating a TA requires supporting the underlying interactions with the TOS. Due to the heterogeneity of TOSes, emulation effort spent on one TOS does not translate to support for others.

Samsung's PartEmu [20] uses full-system emulation to boot and run the entire TOS and TAs. While this approach transparently handles TA-to-TOS interactions, the prototype has neither been open-sourced nor has it been reproduced due to the in-house TOS details needed for this approach. Researchers have instead turned to user mode emulation of TAs [30], [18], [26], [6], [21], [5], [22], implementing the specific TOS system calls or hooking API-level interactions. While these approaches have proven to be practical, they often focus on a single TEE or use an ad-hoc combination of API-level and system call hooking.

TÄMU presents a principled approach to TA emulation, applying high-level emulation (HLE) at the API level to TAs, a technique we refer to as API-level interception. TÄMU first leverages the de-facto standardization by means of the GlobalPlatform (GP) APIs. Many TAs are not developed by the TEE vendor but by a third party (such as Google for the Widevine TA to provide DRM capabilities) and thus need to be deployable on various TEEs. Due to the fragmentation of TEEs, the GP specifications were introduced to enable interoperability for TAs between TEEs. The GP specifications define entrypoint functions in the TA exposed to the normal world and a set of APIs that the TEE is expected to implement for a TA, including memory management, cryptography, and storage. The goal of the GP API is to provide an abstraction layer for the concrete underlying TEE implementation. TÄMU interposes TAs at the API level and leverages the standardization of TA APIs to scale its emulation approach to diverse TEEs.

We conduct a study of the TEEs deployed on modern Android smartphones, quantify their level of API standardization, and discuss the feasibility of supporting these TAs on TÄMU. We find that although the GP API is widely adopted, TAs also make use of standard libc APIs and proprietary TEE-specific APIs. Adding support to TÄMU

* Author completed work while at EPFL; now with Google.

for libc APIs is straightforward as these APIs are publicly documented. Unfortunately, a majority of TAs still makes use of proprietary TEE-specific functionality. 94% of all TAs in our dataset use at least one TEE-specific API. Faithfully emulating TEE-specific APIs requires significant manual effort: reverse engineering of the API's functionality, and then implementing that functionality in an emulator shim.

To handle the manual effort incurred by supporting TEE-specific APIs, we introduce greedy HLE, a technique that leverages static analysis to calculate the potentially reachable code as a function of the set of implemented APIs. This approach allows us to prioritize the manual effort of emulating TEE-specific APIs based on the expected coverage yield. Leveraging greedy HLE, we first demonstrate that with just the GP and libc APIs, 39% of code can be reached. For each TEE, there are between one to eight TEE-specific APIs, which, after being implemented in TÄMU, enable execution of up to 90% of basic blocks. We further demonstrate how 70% of the most impactful TEE-specific APIs (in terms of newly reachable basic blocks) can realistically be replaced by a GP or libc API, reducing the implementation effort and paving the way for further standardization.

We implement TÄMU and show that its API interception approach is feasible by correctly running 67 TAs across four TEEs. We demonstrate the fidelity and usefulness of TÄMU by reproducing n-day vulnerabilities. We further fuzz 30 TAs and discover 17 0-day vulnerabilities. We have responsibly disclosed all the vulnerabilities to the affected vendors. TÄMU is open source and available at https://github.com/HexHive/taemu. In summary, we make the following contributions:

- We identify, measure, and leverage the adoption of the GP APIs as a means to bring cost-effective HLE to the TA ecosystem. Our insight reveals the economies of scale unlocked by the increasing adoption of GP APIs.
- We propose a rehosting methodology (greedy HLE) for proprietary TA APIs based on data-driven decisions to prioritize rehosting efforts.
- We design and implement TÄMU, an open-source implementation of our API interception approach, able to run 67 TAs with an integrated fuzzing component that has discovered 17 0-day vulnerabilities.

2. High-Level Emulation and GlobalPlatform API Adoption

The core intuition of TÄMU results from the observation that the TA ecosystem is adopting a common API, the GP TEE API, that unlocks unprecedented economies of scale, thereby mitigating the major challenges faced by previous rehosting works. In this section, we discuss the prior attempts and pitfalls to leverage HLE in other domains, provide the background on the GP APIs, and empirically validate our industry adoption claims regarding the APIs in question.

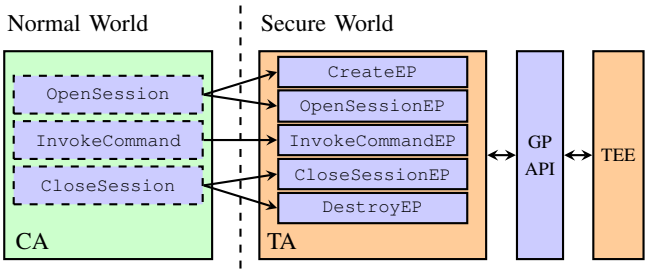

Figure 1: An overview of how GP facilitates communication between the normal world and TA, and between TA and TEE (The TEE includes the user space runtime environment and the TOS kernel). The dashed blue boxes are part of the GP client API, while the solid blue boxes are part of the GP Internal Core API.

2.1. Prior Art

HLE is an approach where, instead of faithfully emulating every hardware peripheral at the register level, the emulator intercepts calls to well-defined software libraries or hardware abstraction layers (HALs) and re-implements their functionality on the host. This allows firmware to run without requiring a fully precise peripheral model and can greatly accelerate re-hosting. For example, when emulating a network card, instead of implementing all hardware intricacies, HLE would interpose the emulation at the send/receive packet API. However, in the deeply embedded world, where this idea was proposed initially by HALucinator [16], it has seen little adoption because the HAL landscape is highly fragmented: each silicon vendor, and often each chip family, ships its own HAL with different APIs, naming conventions, and binary layouts. Covering this diversity requires building and maintaining a large set of HLE shims for every variant, undermining the scalability and portability that make HLE attractive in the first place. HALucinator explores this space but is limited by the lack of a unified HAL standard, limiting its scope to a few targets where HALs are inferred manually, resulting in limited scalability and therefore little adoption.

> HLE emulation is feasible in scenarios where a well-defined, unified abstraction interface exists.

2.2. GlobalPlatform APIs

Unlike embedded HALs, TEEs have begun to converge around a standardized abstraction layer — the GlobalPlatform (GP) API.

The TEE landscape has historically been fragmented, with each vendor implementing its own proprietary APIs and runtime services. TAs built for one TEE often require significant modifications to run on another, making cross-platform deployment expensive and error-prone. This fragmentation is further amplified by device manufacturers shipping products with diverse system-on-chips and incompatible TEE stacks, forcing developers to maintain multiple

```
TEE_Result TA_InvokeCommandEntryPoint(void *sess_ctx,
  int cmd_id, int param_types, TEE_Param params[4]){
  if (param_types != 0x65)
      tee_log("bad parameter types!");
      return TEE_ERROR_BAD_PARAMETERS;

  char* in_buf = params[0].memref.buffer;
  size_t in_len = params[0].memref.size;

  char* copy_buf = TEE_Malloc(in_len, 0);
  memcpy(copy_buf, in_buf, in_len);

  char* key = NULL;
  size_t key_len = 0;
  tee_get_key(&key, &key_len);

  /* setup AES operation op with the key */
  /* cut for brevity */
  res = TEE_CipherDoFinal(op, copy_buf, in_len,
      params[1].memref.buffer, 0x100);
  return TEE_SUCESS;
}
```

Listing 1: An example TA's GP entrypoint function (`TA_InvokeCommandEntryPoint`). The TA encrypts data with a TEE-internal key and returns the ciphertext to the normal world. Lines marked in blue are instances of the TA calling GP APIs. Lines marked in green contain a call to a libc API. Lines in red are calls to TEE-specific APIs.

codebases and weakening the portability of security-critical components.

To address this challenge, GP proposes specifications, which define a unified interface for both the normal and the secure world. GP provides a consistent programming model that abstracts vendor-specific differences. This standardization enables developers to write TAs once and deploy them across any GP-compliant TEE with minimal adaptation, reducing engineering overhead and fostering a more interoperable and secure ecosystem.

The GP specification is split into two complementary parts. The Client API [1] defines how normal-world applications communicate with TAs through standardized session management and parameter passing. In contrast, the TEE Internal Core API [2] specifies the secure-world environment in which TAs run, including entrypoints, which handle requests from the normal world, as well as core services such as memory management, cryptography, secure storage, and inter-TA communication. Together, these two APIs bridge the gap between heterogeneous hardware and portable, security-critical software. Figure 1 gives an overview of how the GP API bridges the normal and the secure world.

By adhering to the GP specification, a TA developer can deploy their TA on different GP-compliant TEEs without heavily modifying the TA for each TEE. With the exception of Google's Trusty, all TEEs deployed on modern Android are at least partially GP-compliant. While the GP APIs are commonly used, many TAs also rely on libc standard

TEE	SOCs	OEMs
TEEGris	Exynos	Samsung
MiTEE	MediaTek	Xiaomi
Beanpod	MediaTek	Xiaomi
T6	MediaTek	various
QSEE	QualComm	various
Kinibi	Exynos, MediaTek	Samsung + various
TrustedCore	HiSilicon	Huawei

Table 1: An overview of the TEEs considered in the GP API adoption measurement study.

API functions. Furthermore, TAs may also include TEE-specific APIs, functions not defined in either the GP or libc specification. Listing 1 shows the source code of an example TA's `TA_InvokeCommandEntryPoint` function (a GP entrypoint function handling requests from the normal world). The TA uses the GP API to allocate memory and the GP cryptographic API to encrypt the user input. It also uses the libc API to make a copy of the input buffer. Furthermore, it uses TEE-specific APIs to retrieve the encryption key and for logging. When porting this TA from one TEE to another TEE supporting GP and libc APIs, the developer only needs to adjust the two TEE-specific API invocations.

> In the TA space, GP and libc APIs define a clear abstraction layer, whose adoption is driven by the need for standardization in the TEE market.

2.3. GP API Adoption Measurement

Before investing significant engineering effort into rehosting TAs for dynamic analysis and fuzzing, it is essential to understand how much code actually shares a set of common APIs. Rehosting is costly: it requires reverse engineering, emulator extension, and often custom peripheral modeling. If the ecosystem is highly fragmented and lacks shared interfaces, each TEE would require its own set of high-level emulation (HLE) shims, making large-scale dynamic testing impractical. Conversely, if most TAs rely on a common standard, rehosting becomes far more scalable, as one emulator backend can support a wide range of devices.

We perform an empirical study combining static inspection of TA binaries with historical snapshots to map adoption trends over time. We download firmware for various system on chip and phone vendors from three points in time (2015, 2020, and 2025). We extract the TA binaries from the firmware and semi-automatically analyze them with a decompiler. To determine whether a TA makes use of GP APIs, we look for the GP entrypoint functions. To search for the GP entrypoint functions, we first use a script that marks TAs for further manual analysis, if at least one potential function (matching function signature) is found for each GP entrypoint function. We then manually analyze the marked TAs. We mark TAs as using the GP API if we confirm the existence of the GP entrypoint functions. This allows us to identify when major vendors began integrating GP APIs and how consistently developers now depend on them.

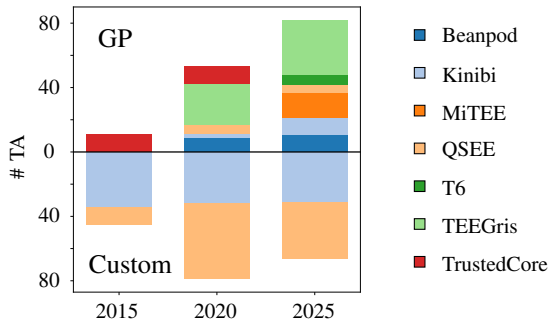

Figure 2: The distribution of TAs using the GP API compared to TAs without across three time snapshots.

TEE	# TAs	# TAs GP-EP	# TAs GP API	# TAs libc API	# TAs TEE API
TEEGris	**34**	34	30	34	34
MiTEE	**16**	16	16	16	12
Beanpod	**11**	11	9	11	11
T6	**6**	6	6	0	6
QSEE	**41**	5	5	0	5
Kinibi	**31**	10	0	0	10

Table 2: Our dataset of TAs from 2025 along with the types of API used by TAs. For TAs that expose the GP entrypoints (EP), we further analyze if the TA uses the GP APIs, libc APIs, or TEE-specific APIs.

We analyze 337 TAs covering seven commercially used and proprietary TEE implementations. To the best of our knowledge, these are *all* the TEEs currently deployed on Android phones, excluding Huawei's iTrustee, whose TAs are encrypted, and Google's Trusty, which is only used on Pixel devices and does not reuse the GP specifications. See Table 1 for additional information on the considered TEEs. Figure 2 shows the trend: only 19% of TAs relied on the GP API in 2015, adoption reached 40% in 2020, and exceeded 55% in 2025. All TEEs introduced after 2020 make use of the GP API. The number of TAs not supporting GP has stagnated and is confined to Kinibi and QSEE with the remaining five TEEs supporting the GP API. Even for Kinibi and QSEE, new TAs are written to be GP-compliant.

As discussed in Section 2.2, a TA using the GP APIs may also rely on libc and TEE-specific APIs. We analyze the 82 GP-compliant TAs from 2025 and whether these TAs make use of libc or TEE-specific APIs. The results are shown in Table 2. TAs from TEEGris, MiTEE, and Beanpod make heavy use of functions from the libc standard library. Supporting libc APIs for HLE is similar to supporting GP APIs, since their functionality is publicly documented. Thus, we treat libc APIs the same as GP APIs, i.e., standard APIs, whose HLE emulation shim scales to various TEEs.

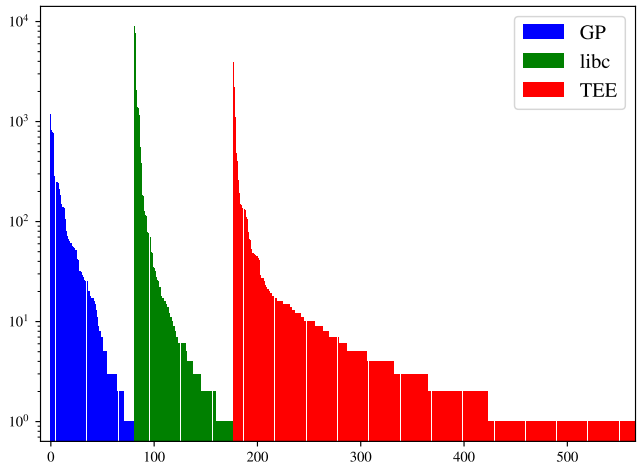

Figure 3: The frequency graph of the types of APIs used across our dataset of GP-compliant TAs. The y-axis denotes the number of invocations of a given API function in log scale.

> The empirically demonstrated convergence towards standardization indicates that the TEE ecosystem is now a prime candidate for HLE-based dynamic analysis, where one GP/libc-aware rehosting framework could scale across most TEEs.

Critically, we find that 94% of TAs use at least one TEE-specific function. Figure 3 shows the frequency graphs of the APIs used by the 82 TAs. Each point on the x-axis is an API function (either GP, libc, or TEE-specific). The y-axis denotes in log scale the number of times the API is called across all TAs. All three categories contain a few high-frequency functions. However, there are more TEE-specific APIs both in terms of the number of unique functions and invocations.

> TEE-specific APIs are still widely used and account for a substantial portion of TA API invocations. Therefore, an effective rehosting approach must address TEE-specific APIs.

3. TÄMU

Figure 4 illustrates TÄMU's design. The left side shows TÄMU's rehosting platform, which allows the execution of GP-compliant TAs on commodity hardware through API-level interception. We explain how the components on the left side work to enable API-level interception in Section 3.1. The right side shows greedy HLE, a technique to prioritize the manual rehosting effort for TEE-specific APIs based on the potential coverage gain. In Section 3.2, we discuss this technique and how it enables TÄMU to scale in the face of TEE-specific APIs. The middle of the figure shows how an analyst interacts with TÄMU. The analyst

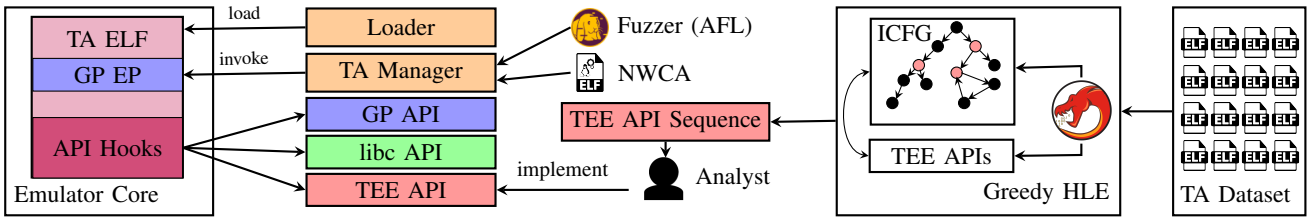

Figure 4: Overview of TÄMU's design.

extends TÄMU's TEE-specific APIs using the TEE APIs obtained with greedy HLE. An analyst interacts with the emulated TA either via commodity fuzzers (AFL) or normal world client applications (NWCA).

3.1. API-Level Interception

At the core of TÄMU's design is the insight that the TA ecosystem is moving towards standardization in terms of the APIs leveraged by TAs. TÄMU's rehosting platform is built around a virtual TEE that is able to load, invoke, and provide the required standard GP and libc APIs to the emulated TA. The TA itself runs in the emulator core, which implements the processor emulation—the semantically correct execution of instructions and the corresponding CPU register and memory updates. The emulator core enables TÄMU to run TAs from any architecture (primarily arm32 or arm64) on a commodity x86 processor.

TÄMU's loader component takes care of correctly mapping the TA into the emulator core. All of TÄMU's supported TEEs ship their TAs as ELF files, allowing TÄMU's loader to reuse existing ELF loading functionality. However, there is no inherent limitation to extend the loader to support TEE-specific executable formats, such as Kinibi's proprietary MCLF format.

To enable API-level interception, TÄMU instruments the TA's API invocations such that whenever the TA invokes an API function, execution is redirected to TÄMU's virtual TEE. From there, the corresponding HLE API implementation in TÄMU is called. Afterwards, execution is resumed in the TA at the expected return address. API invocations are instrumented by installing hooks in the emulator core, which trigger whenever the instruction pointer is equal to the address of an API invocation. The instruction pointer value is used to infer which API the TA was calling.

While these are standard techniques for HLE, TÄMU's main contribution stems from the insight that, thanks to the TEE ecosystem's standardization, heterogeneous TEE interactions go through standard GP or libc APIs. TÄMU's virtual TEE hooks and implements both the GP and libc APIs. Implementing these APIs correctly is feasible due to both GP and libc providing specifications for their APIs. Furthermore, these APIs behave the same regardless of the underlying TEE and implementing them is a one-time effort that scales to any TEE. We differentiate between the following categories of standard APIs:

Stateless APIs. Stateless APIs behave the same irrespective of any prior API invocations by the TA. Examples of such APIs are libc's `memcmp` or GP's `TEE_MemCompare`.

Stateful APIs. These APIs behave differently depending on prior API invocations. TÄMU tracks such internal state to accurately emulate these APIs. Examples of such APIs are memory allocation APIs such as `malloc`/`TEE_Malloc` and GP APIs related to storage and cryptography.

GP Entrypoints. Unlike the previously described APIs, the GP entrypoint APIs in TÄMU are not invoked by the TA itself but by TÄMU's TA manager component. Invocation of an entrypoint triggers the emulation of the TA until the entrypoint function returns. The TA manager calls these entrypoint functions in response to interactions from external programs communicating with the emulated TA. The TA manager supports two modes of interaction: *interactive* and *fuzzing*. In interactive mode, a NWCA can communicate with the emulated TA just as it would on a real device. In fuzzing mode, the TA manager acts as a harness for a fuzzer, deserializing the fuzzing input into the TA's input buffers and invoking the TA's command-handling entrypoint API.

As discussed in Section 2.3, TEE-specific APIs make up a significant amount of the API invocations in our dataset. While the technique to intercept TEE-specific APIs is the same as described above, there is no specification for these TEE-specific APIs, and thus implementing them requires manual effort. We discuss how TÄMU handles TEE-specific APIs leveraging greedy HLE in the following subsection.

3.2. Greedy HLE

With overall 389 unique TEE APIs across 82 TAs, reverse engineering all these proprietary APIs and faithfully implementing them is prohibitively expensive. Instead, we need a way to prioritize TEE-specific APIs.

We introduce the notion of greedy HLE, a generic technique for incrementally rehosting binary components in emulated environments by selectively modeling high-level interfaces. The approach leverages two key properties: 1) the greedy-choice property—where modeling the next interface with the highest expected benefit yields immediate and independent progress—and 2) optimal substructure—where each successful emulation step expands the execution context without invalidating previous work. Greedy HLE operates over a well-defined abstraction interface that captures interactions with the runtime environment (e.g., TEE-specific

TEE	# TAs	# APIs	# GP APIs	# libc APIs	# TEE APIs	# max. BBs	std-cov.*	# greedy 90%†
TEEGris	34	380	58	56	266	152'755	9%	8
MiTEE	16	151	52	64	35	57'227	98%	0
Beanpod	11	153	49	26	78	19'643	70%	2
T6	6	70	56	0	14	17'368	77%	1
All	67	566	81	96	389	246'997	39%	10

Table 3: The results of the greedy HLE study. (*) Percentage of reachable basic blocks after implementing all GP and libc APIs. (†) The number of TEE-specific APIs needed to achieve 90% potentially reachable basic blocks (BBs).

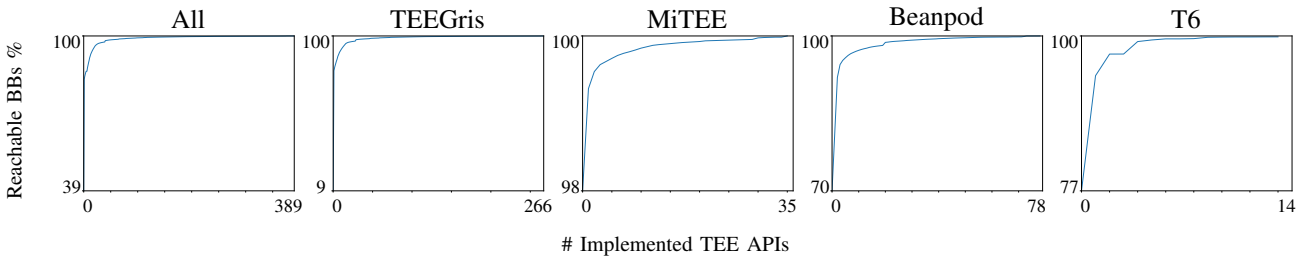

Figure 5: The reachable basic blocks in relation to the number of implemented TEE-specific APIs. The graph starts from all GP and libc APIs implemented, and TEE-specific APIs are added with greedy HLE.

Algorithm 1: Greedy High-Level Emulation (HLE)

Input: ICFG $\mathcal{G}$(root,B,E); List of TEE APIs $\mathcal{A}$;
Output: Greedy API list $\mathcal{I}$

Initialize: $\mathcal{I} \leftarrow \emptyset$;
while $size(\mathcal{I}) < size(\mathcal{A})$ **do**
  foreach $a \in \mathcal{A} \setminus \mathcal{I}$ **do**
    $gain[a] \leftarrow$ ReachableBlocks$(\mathcal{G}, a \cup \mathcal{I}, \mathcal{A})$;
  $a^* \leftarrow \arg\max_a gain[a]$;
  $\mathcal{I} \leftarrow \mathcal{I} \cup \{a^*\}$;
return $\mathcal{I}$

Procedure ReachableBlocks($\mathcal{G}$,$\mathcal{I}$,$\mathcal{A}$):
  foreach $b \in \mathcal{G}.B$ **do**
    foreach $a \in \mathcal{A} \setminus \mathcal{I}$ **do**
      if *b calls a* **then**
        $\mathcal{G}.B \leftarrow \mathcal{G}.B \setminus b$
  return *size(descendants($\mathcal{G}$,root))*

API functions). This abstraction provides a uniform representation for analyzing, ranking, and selecting the next functions to emulate, independent of their underlying implementation details. See Algorithm 1 for the pseudo code of greedy HLE.

We apply greedy HLE to TEEs by targeting the TEE-specific APIs used by TAs. At each step, we select the next API function to implement based on its expected coverage yield—i.e., the number of additional basic blocks it unlocks across the TA corpus. By prioritizing APIs that enable the execution of multiple TAs or uncover deeper code paths, this strategy rapidly expands the dynamic analysis surface while minimizing emulation effort. Although demonstrated in the context of TAs, greedy HLE is a general approach applicable to other domains where execution hinges on incrementally modeling high-level interfaces.

To guide this selection process, greedy HLE leverages an interprocedural control flow graph (icfg) to compute expected coverage as a function of the currently implemented APIs. The icfg encompasses all relevant code of the target, with each basic block annotated by the API calls it makes through the abstraction interface. A basic block calling an unimplemented API terminates emulation and is thus removed from the icfg, along with all its incoming and outgoing edges. Once all APIs called by a basic block are implemented, the block and its edges are reintroduced.

Starting from an empty list of implemented TEE-specific APIs, greedy HLE incrementally appends to this list by ranking and selecting API functions through the abstraction interface according to their coverage contribution—i.e., the number of newly reachable basic blocks in the icfg. This structured ranking process ensures systematic and measurable progress toward full emulation coverage, guiding the manual rehosting effort required to support TEE-specific APIs.

Greedy HLE for TAs. We conduct a study on the TAs of the four fully GP-compliant TEEs, excluding QSEE and Kinibi due to low GP compliance, to demonstrate how greedy HLE enables emulating TAs from a wide variety of TEEs with minimal manual effort.

For TAs, the relevant code is all the code reachable from the GP entrypoint functions since they may be invoked from the normal world. For each TA, we use Ghidra [27] to build the icfg starting from the GP entrypoint APIs, annotating basic blocks that invoke API functions (GP, libc, or TEE-specific). To categorize an API call, we compare its name against the list of APIs defined by the GP Internal Core API and by libc. If neither of the lists contain the API name, we classify the call as TEE-specific.

Since our goal is to have a global ranking of the most

important TEE-specific APIs, we have to reason across our entire dataset of TAs, i.e., we need to apply greedy HLE across multiple TAs. We first generate a TEE-level icfg, which includes all relevant TA code within a given TEE, by merging the individual TA icfgs belonging to that TEE. Specifically, we create a synthetic "TEE root" basic block that connects to the first basic block of each GP entrypoint function from all TA icfgs in that TEE. This TEE-level icfg allows us to apply greedy HLE across all TAs within a single TEE by measuring the basic blocks reachable from the synthetic TEE root. To extend greedy HLE to our entire dataset, we create a global icfg whose synthetic root basic block connects to each TEE root basic block.

We first leverage the icfg to measure the amount of code covered by implementing the standard APIs (GP and libc). Afterwards, we use greedy HLE to add TEE-specific APIs. See Figure 5 for the results for each TEE and merged across all TEEs, and Table 3 for an overview of the numbers.

The plots show that there are a few high-impact TEE-specific APIs, followed by a long tail of TEE-specific APIs that only marginally contribute to the number of potentially reachable basic blocks. These results demonstrate that an HLE approach for TAs is feasible. An emulator supporting the GP APIs and libc APIs achieves around 39% of potentially reachable basic blocks. The reason for this low percentage is TEEGris, where the TEE API `TEES_IsREESharedMemory` blocks most code, resulting in only 9% potentially reachable basic blocks for TEEGris. This API essentially reimplements an existing GP API (used to check permissions on shared memory). After emulating this API, the reachable basic blocks jump to 79% for TEEGris and 82% for all. Across all TEEs, implementing ten TEE-specific APIs, guided by greedy HLE, achieves 90% potentially reachable basic blocks.

> Leveraging greedy HLE, we identify a limited number of high-impact TEE-specific APIs, which enable emulation of up to 90% of basic blocks in TÄMU.

To further motivate TÄMU's HLE approach, we study the manual rehosting effort incurred by TEE-specific APIs. We manually analyze the top ten TEE-specific APIs for each TEE (in terms of newly reachable basic blocks in the icfg after implementation) to try and understand if the TEE-specific API could be replaced by an equivalent GP or libc function. We analyze 40 TEE-specific APIs. For 24 of these, there exists an appropriate GP API. An example is `ut_pf_cp_rd_random` used by Beanpod, which generates random data and can be replaced by `TEE_GenerateRandom`. For a further five APIs, there exists a corresponding libc API. Overall, 70% of analyzed TEE-specific APIs are replaceable by a GP and libc API. This indicates that in the future, these TEE-specific APIs may be replaced by appropriate GP/libc APIs. Furthermore, the rehosting effort for such APIs is minimal, as existing functionality can be reused. The complete table of API functions can be found in Table 4.

> Most high-impact TEE-specific APIs can be feasibly replaced by existing GP/libc functionality.

4. Implementation

We implement TÄMU on top of Qiling [3], which itself is a wrapper around Unicorn [4], in around 6,000 lines of Python.

Annotation of Addresses. TÄMU requires the addresses of API functions to enable API-level interception.

TEEGris and Beanpod TAs are dynamically linked ELF files that export/import the GP, libc, and TEE APIs. Hence, the loader can simply parse the ELF header to determine the addresses of these functions. T6 and MiTEE TAs are statically compiled and, thus, require special handling. For these TAs, the loader uses a TA-specific configuration file, which stores the offsets to the relevant functions. We automate the generation of this file using a Ghidra headless script. The script uses logging strings to map functions to the GP and libc APIs. The script further annotates call sites that have not been marked as an API but contain unhandled system calls. These unhandled system calls require a one-time effort to manually complete the previously generated configuration file to annotate the API making the system call in question. While the automation with Ghidra could be further improved using binary similarity or similar techniques, we consider the manual effort negligible.

API Hooking. To hook the API calls for dynamically linked TAs, TÄMU overwrites the global offset table entries with unique pointers. When the program counter in the emulator core is equal to one of these unique pointers, the corresponding API handler is called. For statically linked TAs, TÄMU parses the configuration file and hooks the inline API function addresses.

The API handlers are implemented in Python and use the Qiling API to update registers and memory. Listing 2 shows the implementation for the `TEE_MemMove` function. The implementation first retrieves the function arguments. It uses TÄMU's address sanitizer (ASAN) implementation to check if memory accesses are valid. Before copying data to the destination, it synchronizes the shared memory. Finally, it copies the data from the source and writes it to the destination. The Qiling API is used to set the return value. To return to the callee position, the address in the `lr` register is restored to the `pc`. This simple API hook is able to detect non-crashing out-of-bounds accesses with ASAN, simulates shared memory accesses, and correctly implements the `TEE_MemMove` functionality.

Fuzzing. The fuzzing mode uses AFL++ [19]'s Unicorn mode in combination with Qiling's `libunicornafl` support to fuzz the target TA. In fuzzing mode, the emulator acts as the target program for AFL++. To start the fuzzing mode, the user specifies the target TA and a Python harness script as arguments to the emulator. The TA manager loads the harness script. The harness specifies two functions. The `init` function, which runs before the

TEE	API	Functionality	GP equivalent	libc equivalent
TEEGris	TEES_IsREESharedMemory	check memory permissions	TEE_CheckMemoryAccessRights	N/A
TEEGris	TEES_GetClientCredentials	retrieve NWCA authentication info	TEE_GetProperty (client)	N/A
TEEGris	sqrf, roundf...	float math operations	N/A	N/A
TEEGris	TA_Communication_mpos_check_iccc	IPC communication	TEE_InvokeTACommand	N/A
TEEGris	hdm_ICCC_check	IPC communication	TEE_InvokeTACommand	N/A
TEEGris	TEES_UnwrapSecureObject	Access encrypted persistent data	TEE_ReadPersistentObject	N/A
TEEGris	OPENSSL_malloc	memory allocation	TEE_Malloc	N/A
TEEGris	TEES_CheckSecureObjectCreator	check integrity of persistent data	TEE_OpenPersistentObject	N/A
TEEGris	TEES_SPIWriteRead	peripheral access	N/A	N/A
TEEGris	TEES_DeriveKeyKDF	generate key	TEE_GenerateKey	N/A
MiTEE	tee_se_open_spi_clk	peripheral access	N/A	N/A
MiTEE	tee_get_enc_rot	retrieve TEE specific value	TEE_GetProperty	N/A
MiTEE	tee_get_cpuid	retrieve device specific value	TEE_GetProperty	N/A
MiTEE	TEE_SEChannelClose	close IPC communication	TEE_CloseTASession	N/A
MiTEE	TEE_SaveTA_Data	write data to persistent storage	TEE_WriteObjectData	N/A
MiTEE	gen_random	generate random data	TEE_GenerateRandom	N/A
MiTEE	tee_se_close_spi_clk	peripheral access	N/A	N/A
MiTEE	TEE_SESessionOpenBasicChannel	IPC communication	TEE_OpenTASession	N/A
MiTEE	TEE_SEChannelGetNumber_ext	IPC communication	N/A	N/A
MiTEE	TEE_SESessionOpenLogicalChannel	IPC communication	TEE_OpenTASession	N/A
Beanpod	TEE_LogPrintf	logging	N/A	printf
Beanpod	msee_ta_printf_va	logging	N/A	printf
Beanpod	ut_pf_log_msg	logging	N/A	printf
Beanpod	ut_pf_km_get_hmac_key	retrieve keymaster key	TEE_InvokeTACommand	N/A
Beanpod	mdrv_open	initialize session with driver	TEE_OpenTASession	N/A
Beanpod	ut_pf_cp_rd_random	generate random data	TEE_GenerateRandom	N/A
Beanpod	base64_decode	base64 decoding	N/A	N/A
Beanpod	TEE_RpmbOpenSession	open object from RPMB	TEE_OpenPersistentObject	N/A
Beanpod	TEE_RpmbReadData	read data from RPMB	TEE_ReadObjectData	N/A
Beanpod	ut_pf_info_get_deviceinfo	get device information	TEE_GetProperty	N/A
T6	debug_log	logging	N/A	printf
T6	debug_log2	logging	N/A	printf
T6	__assert_fail	log and abort	TEE_Panic	N/A
T6	TEE_GetBootSeed	retrieve hw specific data	TEE_GetProperty	N/A
T6	platform_spi_write_read	peripheral access	N/A	N/A
T6	init_ta_session	setup IPC	TEE_OpenTASession	N/A
T6	sf_spi_write_buf	peripheral access	N/A	N/A
T6	sf_spi_write_then_read_buf	peripheral access	N/A	N/A
T6	check_license	get device information	TEE_GetProperty	N/A
T6	platform_open_driver	intitiate IPC	TEE_OpenTASession	N/A

Table 4: The analyzed highest-impact TEE-specific APIs, their reverse-engineered functionality, and the possible replacement candidates from libc or GP.

```
def TEE_MemMove(ql: Qiling, hook_data):
  params = ql.os.resolve_fcall_params(
    {"dest": POINTER, "src": POINTER, "size": POINTER}
  )
  if not asan.is_access_valid(
    ql, hook_data.emu.HEAP, params["dest"], params["size"],
    hook_data.func_name, is_write=True
  ): return
  if not asan.is_access_valid(
    ql, hook_data.emu.HEAP, params["src"], params["size"],
    hook_data.func_name, is_write=False
  ): return
  hook_data.emu.update_shm(params["src"])
  try:
    data = ql.mem.read(params["src"], params["size"])
    ql.mem.write(params["dest"], bytes(data))
  except unicorn.unicorn_py3.unicorn.UcError as e:
    crash(ql, hook_data.func_name)
    return
  hook_data.emu.writeback_shm(params["dest"])
  ql.os.fcall.cc.setReturnValue(params["dest"])
  ql.arch.regs.arch_pc = ql.arch.regs.lr
```

Listing 2: The `TEE_MemMove` API implementation of TÄMU.

fuzzing loop and can be used to set up a specific state in the target TA. The `init` function is called before AFL's forkserver is started, so it only runs once. The second harness function, `place_input_callback`, is invoked for every fuzzing iteration and is responsible for deserializing the fuzzer-provided bytes into the arguments of the `TA_InvokeCommandEntryPoint` function.

Address Sanitization (ASAN). Since TÄMU hooks the memory management APIs (such as `TEE_Malloc` and `TEE_Free`), TÄMU leverages these hooks to track the heap state. When allocating memory, TÄMU sets up a redzone around the returned memory chunk and hooks accesses to this region to detect out-of-bounds accesses. It also tracks the state of chunks to detect invalid frees and double frees. Other APIs that read or write from or to memory use the ASAN sanitizer to ensure memory accesses are valid.

Emulating Common TEE Vulnerabilities.

TÄMU ensures TA-specific vulnerabilities are correctly emulated. Specifically, parameter type-confusion bugs [11] and time-of-check to time-of-use (TOCTTOU) bugs can be faithfully emulated with TÄMU.

Parameter type-confusion bugs are the result of improper sanitization of untrusted input. These inputs can lead to vulnerabilities allowing an attacker to arbitrarily control

pointers in the virtual address space of the affected TA. The TA manager ensures that these type confusions are triggerable by handling the client's inputs faithfully using OP-TEE as the reference implementation. In Section 5.3, we show how we use the emulator to debug and exploit a n-day parameter type-confusion bug.

TOCTTOU bugs in the context of TAs arise from memory being shared between the normal and secure world. If the TA operates directly on the shared memory, concurrent changes by the normal world can lead to inconsistent or stale reads — i.e., the TA checks a value and later uses it after the normal world has modified it, opening a race window that an attacker can exploit. TÄMU models this shared-memory interaction precisely: changes by the NWCA to the TA memory buffer are propagated to the emulator via Linux shared memory, which allows concurrent modifications by the NWCA. In Section 5.1 we demonstrate how we are able to reproduce an n-day TOCTTOU bug with TÄMU.

5. Evaluation

We evaluate TÄMU's emulation capabilities across various TEEs, first assessing its fidelity and then focusing on the use cases: fuzzing and debugging.

Throughout the evaluation, we aim to answer the following research questions:

RQ1 Can TÄMU effectively be used to reproduce known (n-day) vulnerabilities in commercial TAs?

RQ2 Can TÄMU effectively discover new (0-day) vulnerabilities in commercial TAs, reproducible on-device?

RQ3 Can TÄMU unlock fuzzing capabilities for commercial TAs?

5.1. Fidelity (RQ1 & RQ2)

To evaluate the fidelity of TÄMU, we first evaluate if TÄMU can be used to reproduce n-day vulnerabilities. In the second part, we evaluate whether the crashes in TAs found during development and fuzzing can be reproduced on our physical commercial off-the-shelf (COTS) testing devices.

For the first fidelity study, we pick six n-day TA vulnerabilities from TÄMU's supported TEEs. Additionally, we added support to TÄMU for emulating a single TrustedCore Huawei TA to demonstrate the reproducibility of shared memory TOCTTOU vulnerabilities. We chose vulnerabilities for which a proof of concept or a writeup exists, allowing us to write our own proof of concept NWCA. The vulnerabilities we chose include common memory corruptions, i.e., overflows and TEE-specific vulnerabilites such as parameter type-confusions and TOCTTOUs. We compile and run our proof of concept NWCA against the vulnerable TA emulated in TÄMU. We mark this as a success if TÄMU detects the crash. Table 5 shows the n-day vulnerabilities and the result of replaying the proof of concept against the emulator. We successfully reproduced all vulnerabilities in TÄMU.

The second fidelity study aims to reproduce 0-day vulnerabilities found during development of or fuzzing with

```python
def place_input_callback(ql: Qiling, input: bytes, _: int):
  if len(input) < 2:
        return False

  cmd = input[0] % 5
  params = []
  params.append(
    MemRefParam(input[1:], len(input[1:]))
  )
  params.append(
    MemRefParam(bytes(0x608),0x608)
  )
  params.append(NoneParam())
  params.append(NoneParam())
  ptypes= 0x65
  setup_params_fuzz(ql, cmd, ptypes, params)
  return True
```

Listing 3: An example harness for the `14b0aad8-c011-4a3f-b66aca8d0e66f273.ta` TA. The `place_input_callback` function is called for every fuzzing iteration with the fuzzer-generated bytes in the `input` argument. The TA only exposes 5 commands and expects a `0x608`-sized output buffer as the second parameter.

TÄMU on physical COTS devices. For each of these vulnerabilities, we write a proof of concept NWCA, compile them for the corresponding Android device, and check if we can trigger the crash on the device. For Beanpod TAs, we use a Xiaomi Redmi Note 11s. For MiTEE TAs we use a Xiaomi Redmi Note 13 5G and for the T6 TAs we use a Ulephone Power Armor 18. All our testing devices are rooted to allow interaction with the TEE kernel driver. Note that on-device, we do not have any introspection capabilities and can only detect if the TA crashed by observing the return code of `TEEC_InvokeCommand` (we mark a crash as reproduced if we observe a return value of `TEE_ERROR_TARGET_DEAD` and a `returnOrigin` of `TEEC_ORIGIN_TEE`). Table 6 lists the vulnerabilities discovered with TÄMU. The column "Reproduced on Device" marks whether we were able to reproduce the vulnerability on the device. Out of the 17 vulnerabilities we reproduced 11. For 5 vulnerabilities, we could not reproduce them due to our testing devices not supporting that TA. Due to the TEE's enforcement of only running correctly signed code, it is not possible to load TAs from another device onto our testing devices. We were unable to reproduce a 4-byte heap overflow vulnerability due to the limited corruption primitive, and the underlying allocator mechanism.

5.2. Fuzzing TAs (RQ3)

We ran a fuzzing campaign against 30 TAs. We chose these TAs based on the expected input format in their `TA_InvokeCommand` entrypoint function. All of the fuzzed TAs expect a single input buffer. This allows us to apply the same logic to all harnesses, effectively targeting the TAs. The other TAs implement a more complicated input format. Fuzzing these TAs without a bespoke harness will

TEE	TA	CVE-ID	Vuln Type	Reproduced
TEEGris	00000000-0000-0000-0000-000000000046	SVE-2019-14867	parameter type confusion	✓
TEEGris	00000000-0000-0000-0000-000048444350	SVE-2019-14850	parameter type confusion	✓
TEEGris	00000000-0000-0000-0000-000048444350	CVE-2019-20545	stack buffer overflow	✓
Beanpod	d78d338b1ac349e09f65f4efe179739d.ta	CVE-2020–14125	heap buffer overflow	✓
Beanpod	08110000000000000000000000000000.ta	CVE-2023-32835	parameter type confusion	✓
TrustedCore	task_storage	N/A	TOCTTOU	✓

Table 5: The results of reproducing n-day vulnerabilities with TÄMU.

TEE	TA	Found During	Vuln Type	Reproduced on Device
Beanpod	08110000000000000000000000000000.ta	development	4 byte heap overflow	✗
Beanpod	df1edda8627911e980ae507b9d9a7e7d.ta	development	parameter type confusion	✓
MiTEE	377ee4e8-af0e-474f-a9d636a9268fe85c.ta	development	TOCTTOU	✓
MiTEE	377ee4e8-af0e-474f-a9d636a9268fe85c.ta	fuzzing	stack overflow	✓
MiTEE	377ee4e8-af0e-474f-a9d636a9268fe85c.ta	fuzzing	heap overflow	✓
MiTEE	f13010e0-2ae1-11e5-896a0002a5d5c51d.ta	fuzzing	buffer underflow	✓
MiTEE	a734eed9-d6a1-4244-aa507c99719e7b7f.ta	development	parameter type confusion	✗*
MiTEE	a734eed9-d6a1-4244-aa507c99719e7b7f.ta	fuzzing	null pointer dereference	✗*
MiTEE	a734eed9-d6a1-4244-aa507c99719e7b7f.ta	fuzzing	oob. read	✗*
MiTEE	9811c1f6-47e3-5cea-ae6ef62ba433c4fd.ta	fuzzing	oob. read	✗*
MiTEE	3d08821c-33a6-11e6-a1fa089e01c83aa2.ta	fuzzing	double free	✓
MiTEE	59a4867c-9fe5-f7c2-b409a46bae6ff73e.ta	fuzzing	oob. read	✗*
MiTEE	8aaaf201-2460-0000-7143fe4f7c823c80.ta	fuzzing	arb. read	✓
MiTEE	8aaaf201-2460-0000-7143fe4f7c823c80.ta	fuzzing	stack overflow	✓
MiTEE	86f623f6-a299-4dfd-b560ffd3e5a62c29.ta	fuzzing	arb. read	✓
T6	9459b61a-02d3-4d1e-b68be94397e7ca8c.ta	development	parameter type confusion	✓
T6	9459b61a-02d3-4d1e-b68be94397e7ca8c.ta	fuzzing	data section overflow	✓

Table 6: The vulnerabilities found with TÄMU either during development or fuzzing. (*)These vulnerabilities could not be reproduced due to our testing devices not supporting the TA.

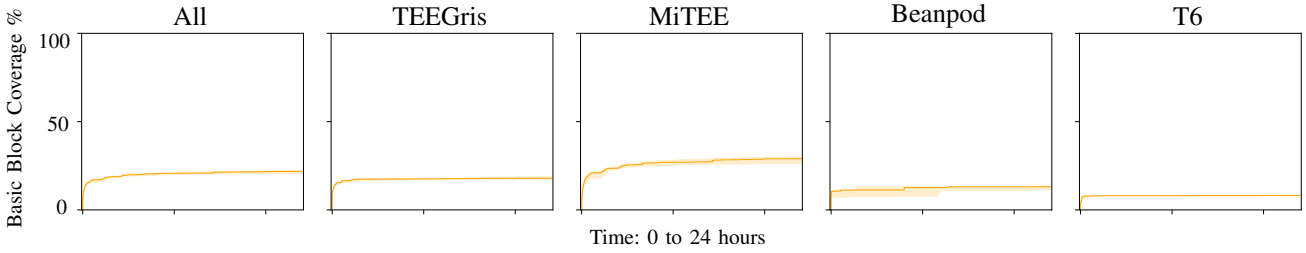

Figure 6: Basic block coverage observed after fuzzing 30 TAs five times for 24 hours as a percentage of the reachable basic blocks.

TEE	# Fuzzed TAs	# Crash	# Bug Crash	# Not impl. API Crash	# max. BBs	# cov. BBs
TEEGris	16	23	0	23	24'740	4643
MiTEE	10	98	84	14	36'128	10'820
Beanpod	2	2	0	2	2485	325
T6	2	31	28	3	10'147	848
All	30	154	112	42	73'500	16'473

Table 7: The number of fuzzed TAs, uncovered crashes, and covered basic blocks of our fuzzing campaign.

lead to few meaningful findings, as all inputs will be rejected early in the entrypoint function.

The fuzzing logic of the harness uses the first byte of the fuzzing data to choose the command and then writes the rest of the fuzzing bytes to the input buffer. For each of the 30 TAs, the harness ensures that the parameter types and buffer sizes corresponding to a given command are as expected by the TA, such that the fuzzer can reach code where the TA starts processing the fuzzing data. Listing 3 shows an example of such a harness.

We fuzz each TA five times for 24 hours. We compare the achieved coverage to all potentially reachable basic blocks from the entrypoints. Figure 6 shows the coverage for the various TEEs. In our fuzzing campaign, we modify the default API function, called for non-implemented APIs, to cause a crash registered with AFL. We replay all crashes, deduplicate, and then triage whether the crashes are true positives or false positives caused by missing API handling.

Table 7 shows the results. Out of the overall 154 crashes, 112 (72%) are true positive crashes. The remaining 42 crashes are due to the TA calling a non-implemented API. On average, we achieve 100 executions per second, which is a 5-10 times speedup compared to Partemu's reported fuzzing speed [20].

This fuzzing campaign uncovered 12 0-day vulnerabilities, see Table 6. As pointed out by related work [10], [24], fuzzing TAs is challenging due to state, complex input format, and large input sizes. In our fuzzing campaign, we address none of these challenges as we fuzz a single invocation of `TA_InvokeCommandEntryPoint` in each fuzzing iteration with raw bytes mutated by AFL. Instead, TÄMU enables setting up a fuzzing campaign with coverage for a large number of TAs from different TEEs.

5.3. Case Study: Debugging

Another important use case for TÄMU is debugging. We demonstrate how we used TÄMU to improve an exploit for the parameter type confusion bug (CVE-2023-32835) in the 08110000000000000000000000000000 Beanpod TA. Exploitation of this vulnerability has already been presented in [11]. As the details of the vulnerability are known, we instead focus on porting the existing proof of concept to TÄMU and adjusting the exploit from a stack-based PC hijack to a more reliable GOT overwrite-based PC hijack to build an arbitrary read primitive.

CVE-2023-32835 is a parameter type confusion bug, which causes the TA to write an integer at an attacker-controlled address. Listing 4 shows the relevant code. Due to the `paramTypes` argument not getting checked, an attacker can supply arbitrary values, which will be interpreted as pointers by the TA. By supplying arbitrary pointers as the `out` buffer, controlled data from the `in` buffer can be written to chosen addresses. The exploit from GlobalConfusion [11] triggers the arbitrary write by sending a request to the TA with `keycount` equal to one, `in_buf[22]` containing the chosen arbitrary write value, and `out` being the chosen arbitrary write location. It then repeatedly triggers this arbitrary write to overwrite the return address stored on the stack to jump to shellcode. We demonstrate how using TÄMU we can exploit this arbitrary write and transform it to an arbitrary memory read. Note that directly using the parameter type confusion for an arbitrary read is not feasible due to the magic number check in the `in` buffer.

We download the existing proof of concept from the artifact of GlobalConfusion [11][1] and recompile it for the emulator. Due to TÄMU's NWCA compilation pipeline, we only need minimal changes to the existing proof of concept to run it against the emulator and reproduce the arbitrary write. The TA uses the TEE-specific `msee_ta_printf_va` API before executing the vulnerable code. Thanks to greedy HLE, this API is supported in TÄMU, and all the code relevant to the proof of concept can be executed. The existing proof

1. https://github.com/HexHive/GlobalConfusion/blob/public/pocs/ta_keyinstall/jni/poc.c

```
TEE_STATUS TA_InvokeCommandEntryPoint(void* session, int cmd,
  int paramTypes, TEE_PARAM params[4]){
  if(cmd == 1){
    int* in = params[0].memref.buffer;
    int in_size = params[0].memref.size;
    int* out = params[1].memref.buffer;
    int out_size = params[1].memref.size;
    if(!TEE_CheckMemoryAccessRights(5, in, inSize) &&
      !TEE_CheckMemoryAccessRights(6, out, outSize)){
      query_drmkey(in, out);
    }
  }
}
void query_drmkey(int* in_buf, int* out_buf){
  msee_ta_printf_va("[KI TA] query_dmr_key_impl start");
  if(in_buf[0] != 0x4d50424b) return;
  int keycount = in_buf[17];
  for(int i=0; i<keycount; i++){
    out_buf[i] = in_buf[22]
    in_buf = &in_buf[24];
  }
}
```

Listing 4: The simplified code of the parameter type confusion in the 08110000000000000000000000000000 TA. Since the parameters are not checked, an attacker can write data from the `in` buffer to an arbitrary pointer supplied with a value in `out`.

```
Breakpoint 1, 0x00009b74 in TA_InvokeCommandEntryPoint ()
──────────────────────────────────────────────────
$r0 : 0x00000005
$r1 : 0xbbbbe000 –> 0x4d50424b 'KBPM'
$r2 : 0x00000370
$r3 : 0x00000004
...
$pc : 0x00009b74
──────────────────────────────────────────────────
–> 0x9b75 fff756e8 blx 0x8c04 <TEE_CheckMemoryAccessRights>
```

Listing 5: The output of gdb, attached to TÄMU's gdb server at the first invocation of `TEE_CheckMemoryAccessRights`.

of concept overwrites a hard coded stack address to hijack the PC. This way of achieving code execution relies on knowledge of the stack address where the return address is stored. Instead, we will use TÄMU's gdb integration to build an exploit that overwrites the GOT to achieve a reliable memory leak. Note that the exploit works because all the involved binaries are not position independent, and the TA's GOT is writable.

We run the proof of concept with gdb attached and set a breakpoint on the first call to `TEE_CheckMemoryAccessRights`. Listing 5 shows the relevant output of the gdb context at the breakpoint. Register `r1` holds the pointer to the `in` buffer. More importantly, `r3` holds `out_size`. The proof of concept uses an `out_size` of 4. Since the TA does not check the parameter types, we can fully control both `r1` and `r3`. Furthermore, by overwriting the GOT entry of

TEE_CheckMemoryAccessRights, we can jump to a chosen address.

We analyze the TA and the TA's loader libld-l4.so for useful gadgets. In the loader we find the following gadget: ldr r3, [r3] ; str r3, [r1] ; bx lr, reading four bytes from r3 and writing them to r1. Since we control both r3 and r1, we can use this gadget for arbitrary read. Our exploit first overwrites the GOT entry of TEE_CheckMemoryAccessRights with the address of this gadget in the loader. Since TÄMU itself overwrites GOT entries to point to hooked memory, this GOT overwrite works transparently in TÄMU. Afterwards, when invoking the TA with out_size set to the target arbitrary read address and a legitimate in buffer, we can leak memory 4 bytes at a time.

With TÄMU's gdb integration and NWCA compilation pipeline, we were able to easily port an existing exploit to TÄMU, debug it, and adjust the exploit to build a 100% reliable memory leak primitive.

6. Discussion

Extending TÄMU to Non-Compliant TEEs. Extending TÄMU to non-compliant TEEs requires first understanding how the TAs are loaded and invoked by the TEE. Further, it requires implementing the TEE-specific APIs.

The two major TEEs currently not supported by TÄMU are QSEE and Kinibi. QSEE TAs wait for requests in a loop in their entry function. Upon request reception, the TA handles the request in its tz_app_command_handler function. Implementing this dispatching logic in TÄMU requires loading and running the TA until it enters the waiting loop. Then, TÄMU can set up the input and output buffers and call the command-handling function. QSEE TAs are dynamically linked and the TEE-specific APIs all start with qsee_. Extending TÄMU with the most used QSEE APIs should be straightforward using greedy HLE.

Kinibi TAs similarly run from their entry point until a main loop, waiting for incoming requests. These requests are then handled in tlApiWaitNotification. Similar to QSEE, TÄMU needs to load the Kinibi TAs and run them until the main loop, then invoke the command handling function. Unlike QSEE, Kinibi TAs are statically compiled and thus identifying the APIs requires a one-time effort as described in Section 4.

Manual Effort to Support TEE-specific APIs. The reverse engineering and implementation effort to support a new TEE-specific API depends on the nature of the API. For some TEE-specific APIs, only the TA's code is required to emulate the API. These include API invocations that do not modify any state in the TA, such as logging functionality or API invocations in which the TA only inspects the return code. For such cases, a simple stub allowing the TA to continue execution is sufficient. For more complex API invocations, the analyst has to reverse engineer the implementation of the API function. This may involve reverse engineering the library, the kernel, or other TAs, in the case of inter-process communication (IPC). For the implementation of TÄMU used in the evaluation, the 40 TEE-specific APIs are implemented in 788 lines of Python. It took one expert analyst, who already implemented the GlobalPlatform and libc API stubs, around one week of work to reverse engineer and implement them.

While greedy HLE is useful to prioritize manual rehosting efforts, as can be seen in Figure 5, after around 90% of reachable basic blocks, greedy HLE yields diminishing returns due to the long tail of TEE-specific APIs. Each additional rehosted API only exposes a few new basic blocks. Beyond this 90% threshold, small numbers of newly potentially reachable basic blocks may be less important to the analyst than the accessibility of specific TA functionality and, greedy HLE may be replaced by a more pragmatic approach of simply implementing unimplemented TEE-specific APIs when these get executed by the TA under test.

Emulating TA to TA IPC. One class of TEE-specific APIs is wrappers around TA to TA IPC. In its current implementation, TÄMU implements this TA to TA IPC by hooking the GP API invocations responsible for IPC, inspecting the destination TA, and returning the expected data. Future work could extend TÄMU to connect multiple instances of TÄMU to transparently handle IPC.

Visibility of Library Code. A significant limitation of API-level interception is the fact that library code is not emulated. There could be bugs in the libraries used by the TAs, or the TA could be misusing the library. Extending TÄMU to load TEE libraries along with the TA is possible. As long as the loaded libraries use the same APIs as the TA, no additional work is needed. However, for libraries designed to interact directly with the TOS, TÄMU needs to be extended to handle these TOS interactions.

TÄMU's API handlers track API-internal state either with handler-internal objects or with TA-visible memory. While TÄMU can detect bugs that lead to invalid accesses to these internal structures, the actual security impact depends on the library's implementation, not visible in TÄMU. An example of this is heap-related bugs. Since TÄMU implements its own allocator, it abstracts away the TEE allocator's internals, such as how heap chunks are laid out and how inline metadata is parsed, which are highly relevant to determine the impact of a heap-related bug.

TA Harnessing. Fuzzing is one of TÄMU's main use cases. For our fuzzing evaluation, we implemented harnesses for 30 TAs. These 30 TAs expect a straightforward input format with a single input buffer. This translates very easily to fuzzing, as the fuzzing input can simply be copied to the input buffer. The remaining TAs use a more complicated input format, with the TA expecting the input in multiple input buffers or as input values. Furthermore, many TAs require a specific state to trigger relevant functionality. Automatically generating harnesses for more involved input formats or to build up state is an open research problem; TÄMU provides a platform to evaluate automatic harness generation for TAs.

One important consideration is the interplay between higher-quality harnesses and greedy HLE. Higher-quality harnesses will execute more unique basic blocks, which in

turn makes it more likely that a TEE-specific API, not implemented by TÄMU, is called. This is a fundamental limitation of greedy HLE, where additional engineering effort invested in harnessing may not result in significantly higher coverage in TÄMU, without investing an equal amount of effort to implement TEE-specific APIs.

7. Related Work

Both academia and industry demonstrated the pervasiveness and criticality of TA vulnerabilities [7], [8], [9], [12], [11], [21], [22], [5], [25], [33], [28], [15], [17], [35], [23], [14], [32], [13]. Motivated by these findings, the research community proposed various analysis techniques for TAs.

The most popular approach is dynamic analysis, as it enables fuzzing. PartEmu [20] executes TAs by partially emulating hardware and software dependencies of the TOS. This powerful approach enables dynamic analysis for TAs, but comes with questionable feasibility in real-world scenarios where datasheets for hardware and source code for closed-source components are missing. SyncEmu [24] manually rehosts a single TOS (Huawei's TrustedCore) and fuzzes TAs by using a NWCA-in-the-loop approach, forwarding TA requests made by the COTS device to the rehosted TA.

As full TOS emulation is infeasible in real-world situations, the focus shifts to emulating TAs. LightEMU [30] emulates TAs at the system call layer. LightEmu reports successfully emulating eight TAs across four TOSes, but lacks details concerning the reverse engineering overhead of implementing the system call stubs for the various TOSes. Fan et al. [18] propose an approach similar to TÄMU—hooking at the API level to emulate QSEE TAs. They emulate one QSEE TA from the Pixel 4. Their work focuses on the internals of the Widevine TA to reproduce an n-day bug with their emulator and does not discuss the feasibility of API-level hooking. A number of ad-hoc industry projects leveraged syscall or API-level hooking to emulate one or multiple TAs [26], [6], [21], [5], [22].

Another research thrust for dynamic analysis of TAs investigates the feasibility of on-device analysis. Crowbar [29] presents an approach to TA fuzzing on a development device using ARM Coresight to extract coverage. This approach is limited by an analyst's ability to obtain such debugging capabilities (e.g., Coresight access). Similarly, DTA [31] executes TAs in the normal world user space and forwards system calls to a custom proxy TA running in the secure world. Prior to DTA, Makkaveev [25] demonstrated this approach to fuzz QSEE TAs on the Google Nexus 6 device. This approach requires a vulnerability to compromise the TEE to allow loading arbitrary TAs. TEEzz [10] fuzzes TAs without coverage feedback on-device, leveraging dynamic analysis on NWCAs to generate high-quality seeds. This approach to high-quality seed generation is complementary to TÄMU's fuzzing mode. SyzTrust [34] targets TEE OS kernels powering deeply embedded systems (e.g., systems without MMU) on-device, whereas TÄMU focuses on user-space hosted TAs of MMU-enabled systems. SyzTrust uses the GP API as the interface through which its fuzz driver targets the TEE OS kernels. While leveraging the same interface, TÄMU explores another angle of the GP API, namely using it as an abstraction to scale emulation of TAs, abstracting away the underlying TEE OS.

An alternative to dynamic analysis of TAs is static analysis. Busch et al. use static analysis to uncover both rollback [12] and parameter type confusion vulnerabilites [11] in TAs.

TÄMU leverages the economies of scale for manual rehosting effort unlocked by the convergence of the TA ecosystem towards a common API—the GP TEE Internal Core API. In contrast to previous approaches to dynamic analysis of TAs, TÄMU unites a promising direction for generic and cross-vendor TA emulation by providing a generic TA emulator in combination with a systematic approach to handle TEE-specific APIs that are not yet part of the emulator.

8. Conclusion

TÄMU is the first principled HLE approach for TAs. TÄMU leverages the well-defined abstraction interface provided by the GP and libc APIs, which allows TÄMU to emulate 39% of all potentially reachable basic blocks. To handle TEE-specific functionality, we introduce greedy high-level emulation (HLE), a technique to prioritize manual rehosting effort. By supporting standard GP and libc APIs, as well as adding support for ten TEE-specific APIs via greedy HLE, 90% of all basic blocks are reachable in TÄMU. Most of these ten TEE-specific APIs can feasibly be replaced by GP/libc APIs and are thus straightforward to implement.

TÄMU is able to emulate 67 TAs from four different TEEs. We demonstrate TÄMU's practicality by using it to fuzz TAs and uncover 17 0-day vulnerabilities, which we have responsibly disclosed to the affected vendors. TÄMU's source code and artifacts are publicly available.

9. Ethics Considerations

We evaluated the ethical implications, considering the interests of vendors and of end users.

For vendors, all discovered vulnerabilities were responsibly disclosed in accordance with coordinated vulnerability disclosure guidelines. Each affected vendor received a private report, submitted either through the vendor's bug bounty platform or via PGP-encrypted email. To prevent premature exposure, we will not publish any technical details of the vulnerabilities prior to the release of this manuscript. Consequently, each vendor will have had over 90 days to investigate and address the reported issues.

For end users, we took care to avoid actions that could create security risks. Neither the paper nor the artifact includes ready-to-use exploit code, and the exploitation case study focuses exclusively on an n-day vulnerability patched in 2023. Our analysis was limited to TAs extracted from publicly available firmware images, and all experiments on COTS devices were conducted on our own testing devices.

10. LLM usage considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

Acknowledgments

This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2 186974, and SNSF 200021-236559.

References

[1] GlobalPlatform TEE Client API Specification, Version 1.0. Technical report, GlobalPlatform, 2010. https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf.

[2] GlobalPlatform TEE Internal Core API Specification, Version 1.3.1. Technical report, GlobalPlatform, 2021. https://globalplatform.org/wp-content/uploads/2021/03/GPD_TEE_Internal_Core_API_Specification_v1.3.1_PublicRelease_CC.pdf.

[3] Qiling: A True Instrumentable Binary Emulation Framework. https://github.com/qilingframework/qiling, 2025.

[4] Unicorn: Multi-platform CPU Emulator Framework. https://github.com/unicorn-engine/unicorn, 2025.

[5] Alexandre Adamski, Joffrey Guilbon, and Maxime Peterlin. A Deep Dive Into Samsung's TrustZone. Quarkslab Blog, 2019. Available at: https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-1.html.

[6] Anonymous. ARM TrustZone: pivoting to the secure world. Thalium Blog, 2023. Available at: https://blog.thalium.re/posts/pivoting_to_the_secure_world/.

[7] Gal Beniamini. Trust Issues: Exploiting TrustZone TEEs. Blog post, Google Project Zero, 2017. Available at: https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html.

[8] David Berard. Kinibi TEE: Trusted Application Exploitation. Blog post, Synacktiv, 2018. Available at: https://www.synacktiv.com/en/publications/kinibi-tee-trusted-application-exploitation.html.

[9] Marcel Busch and Kalle 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*, pages 23–26, 2020.

[10] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. TEEzz: Fuzzing Trusted Applications on COTS Android Devices. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1204–1219, 2023.

[11] Marcel Busch, Philipp Mao, and Mathias Payer. GlobalConfusion: TrustZone Trusted Application 0-Days by Design. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5537–5554, Philadelphia, PA, August 2024. USENIX Association.

[12] Marcel Busch, Philipp Mao, and Mathias Payer. Spill the TeA: An Empirical Study of Trusted Application Rollback Prevention on Android Smartphones. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5071–5088, Philadelphia, PA, August 2024. USENIX Association.

[13] Marcel Busch, Florian Nicolai, Fabian Fleischer, Christsian Rückert, Christoph Safferling, and Felix Freiling. Make Remote Forensic Investigations Forensic Again: Increasing the Evidential Value of Remote Forensic Investigations. In *International Conference on Digital Forensics and Cyber Crime (EAI ICDF2C)*, 2020.

[14] Marcel Busch, Ralph Schlenk, and Hans Heckel. TEEMo: Trusted Peripheral Monitoring for Optical Networks and Beyond. In *Proceedings of the 4th Workshop on System Software for Trusted Execution (SysTEX) co-located with the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[15] Marcel Busch, Johannes Westphal, and Tilo Mueller. Unearthing the TrustedCore: A Critical Review on Huawei's Trusted Execution Environment. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[16] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218. USENIX Association, August 2020.

[17] Adam Crenshaw. On the Nose Bypassing Huaweis Fingerprint authentication by exploiting the TrustZone. Derbycon, 2018. Available at: https://www.youtube.com/watch?v=QFFhdqP7Dxg/.

[18] Chun-I Fan, Li-En Chang, and Cheng-Han Shie. Qualcomm Trusted Application Emulation for Fuzzing Testing, 2025.

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[20] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 789–806. USENIX Association, August 2020.

[21] Daniel Komaromy. Unbox Your Phone – Exploring and Breaking Samsung's TrustZone Sandboxes. Ecoparty, 2021. Available at: https://www.youtube.com/watch?v=L2Mo8WcmmZo.

[22] Zhongquan Li. Dive into Android TA Bug Hunting And Fuzzing. Zhongquan Li's Blog, 2024. Available at: https://imlzq.com/android/fuzzing/unicorn/tee/2024/05/29/Dive-Into-Android-TA-BugHunting-And-Fuzzing.html.

[23] Christian Lindenmeier, Mathias Payer, and Marcel Busch. EL3XIR: Fuzzing COTS Secure Monitors. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[24] Christian Lindenmeier, Matti Schulze, Jonas Röckl, and Marcel Busch. SyncEmu: Enabling Dynamic Analysis of Stateful Trusted Applications. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 177–185, 2024.

[25] Slava Makkaveev. The Road to Qualcomm TrustZone Apps Fuzzing. Check Point Research Blog, 2019. Available at: https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing/.

[26] Hector Marco and Vano Fernando. Auditing Closed Source Trusted Applications for Qualcomm Secure Execution Environment (QSEE). CYBER Intelligence Blog, 2022. Available at: https://cyberintel.es/publications/2022-11-17_DeepSec_pub/.

[27] National Security Agency. Ghidra: A Software Reverse Engineering (SRE) Framework. https://github.com/NationalSecurityAgency/ghidra, 2025.

[28] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 251–268, Boston, MA, August 2022. USENIX Association.

[29] Haoqi Shan, Moyao Huang, Yujia Liu, Sravani Nissankararao, Yier Jin, Shuo Wang, and Dean Sullivan. Crowbar: Natively fuzzing trusted applications using arm coresight. *Journal of Hardware and Systems Security*, 7(2):44–54, 2023.

[30] Haoqi Shan, Sravani Nissankararao, Yujia Liu, Moyao Huang, Shuo Wang, Yier Jin, and Dean Sullivan. LightEMU: Hardware Assisted Fuzzing of Trusted Applications. In *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 01–11, 2024.

[31] Juhyun Song, Eunji Jo, and Jaehyu Kim. DTA: Run TrustZone TAs Outside the Secure World for Security Testing. *IEEE Access*, 12:16715–16727, 2024.

[32] Chad Spensky, Aravind Machiry, Marcel Busch, Kevin Leach, Rick Housley, Christopher Kruegel, and Giovanni Vigna. TRUST. IO: Protecting Physical Interfaces on Cyber-physical Systems. In *Proceedings of the 8th IEEE Conference on Communications and Network Security (IEEE CNS)*, 2020.

[33] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. Horizontal Privilege Escalation in Trusted Applications. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.

[34] Qinying Wang, Boyu Chang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Gaoning Pan, Chenyang Lyu, Mathias Payer, Wenhai Wang, et al. Syztrust: State-aware fuzzing on trusted os designed for iot devices. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2310–2387. IEEE, 2024.

[35] Qi Zhao. Wideshears: Investigating and Breaking Widevine on QTEE. Black Hat Asia, 2021. Available at: https://i.blackhat.com/asia-21/Thursday-Handouts/as-21-Zhao-Wideshears-Investigating-And-Breaking-Widevine-On-QTEE.pdf.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper proposes a rehosting platform named TÄMU that enables dynamic analysis, such as fuzzing and debugging, of Trusted Applications (TAs) by interposing their execution at the API layer. To scale across fragmented Trusted Execution Environments (TEEs), the framework leverages standard GlobalPlatform APIs and introduces greedy high-level emulation to prioritize the manual implementation of TEE-specific APIs based on potential coverage gain.

A.2. Scientific Contributions

- 3. Creates a New Tool to Enable Future Science
- 4. Addresses a Long-Known Issue
- 5. Identifies an Impactful Vulnerability
- 6. Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. The fragmentation and closed-source nature of mobile TEEs severely hinder the dynamic analysis of TAs. By shifting from full-system emulation to API-level emulation based on the widely adopted GlobalPlatform standard, the authors provide a scalable approach that reduces the effort of emulating each TEE one by one.
2) The paper identifies an impactful vulnerability. Through extensive fuzzing campaigns, the authors successfully uncovered 17 zero-day vulnerabilities across 11 real-world TAs. This demonstrates the framework's real-world effectiveness and practical utility.
3) The paper creates a new tool to enable future science. The authors successfully emulated a large dataset of 67 TAs across four distinct TEE implementations, including TEEGris, MiTEE, Beanpod, and T6. Furthermore, the authors have made the source code and artifacts publicly available, allowing for reproduction and serving as a platform for further research.

A.4. Noteworthy Concerns

1) The effectiveness of the greedy high-level emulation prioritization is not evaluated dynamically. The paper does not empirically validate whether implementing the top-ranked TEE-specific APIs leads to substantially higher dynamic coverage gains during a fuzzing campaign compared to implementing a randomly selected set of APIs.

Appendix B.
Response to the Meta-Review

We considered further experiments on greedy HLE, implementing randomly selected TEE-specific APIs. However, implementing TEE-specific APIs that are not reachable in the greedy HLE CFG (due to an unimplemented API being called on its path) is futile: such APIs cannot be executed until their dependencies are implemented.