Artifact Evaluated
NDSS
Available
Functional
Reproduced

Oversharing: Exposing Double Fetch Vulnerabilities in Trusted Applications

Philipp Mao*, Yiwen Xu*, Mathias Payer, Marcel Busch†
EPFL

***Abstract*—Android smartphones leverage a Trusted Execution Environment (TEE) to protect security-critical services. Trusted Applications (TAs) run in the TEE and communicate with the untrusted normal world. To meet performance requirements, the normal world may share physical memory pages with TAs to pass input data. This design exposes the risk of double fetches, in which a malicious normal world attacker can modify shared memory between multiple accesses by a TA, potentially leading to time-of-check-to-time-of-use (TOCTTOU) vulnerabilities.**

To automatically detect attacker-triggerable TOCTTOU vulnerabilities, we present ScHMuzz, a three-stage fuzzer which leverages fuzzing to detect memory safety violations triggered by shared double fetches. We present the first study of cross-domain shared memory usage and TOCTTOU vulnerabilities in TAs deployed on up-to-date mobile phones. We analyze 66 unique TAs from all major commercial TEEs. We find that 30 TAs operate directly on shared memory. We apply ScHMuzz to these 30 TAs, uncovering overlapped fetches from shared memory in 23 TAs and six 0-day TOCTTOU vulnerabilities in five TAs. We reproduce our findings on commercial phones, uncovering the wide use of zero-copy shared memory in commercial TEEs. Looking forward, we find that even TAs written in memory-safe languages are affected by shared memory double fetches. Given that both existing and future TAs are affected by shared memory double fetches, we propose a mitigation that makes shared memory, a performance optimization, explicitly opt-in. Our mitigation prevents all discovered double fetch vulnerabilities from being exploited without changes to the existing TAs. This work addresses a critical and previously overlooked fault line, shared memory, affecting TAs deployed on billions of devices.

I. Introduction

Trusted Execution Environments (TEEs) form the last line of defense on Android devices, protecting sensitive user data even against a compromised normal world. The normal world runs the Android user space and the Linux kernel. To interact with the TEE, the normal world communicates with Trusted Applications (TAs). Each TA implements a subset of the functionality required to interact with TEE-protected services, such as biometrics, DRM, or digital payments. Vulnerabilities in TAs are critical as they allow normal world-based attackers to compromise the security guarantees of the TEE, gaining access to sensitive data and potentially gaining complete control over the targeted device [10], [11], [14], [17], [16], [25], [26], [9], [30], [20].

The GlobalPlatform API defines a standardized interface enabling TAs to be deployed on different GlobalPlatform-compliant TEEs. The specification is now adopted by all relevant TEE implementations [16]. GlobalPlatform specifies how a normal world client application (app[1]) exchanges data with a TA. The app can send input either as integers or a memory buffer. This memory buffer may be shared between the normal and secure worlds. *Shared* in this case means the app and the TA are both accessing the same physical memory page. Such zero-copy shared memory is crucial for performance-critical TA use-cases such as DRM.

Unfortunately, while shared memory is great for performance, handling shared memory securely is difficult due to double fetches. Double fetches arise when a value is fetched from the same physical address in the shared memory twice by the TA. Since the memory may be shared with the untrusted app, a malicious app may intentionally modify the value between the fetches. In the worst case, this allows the malicious app to trigger time-of-check-to-time-of-use (TOCTTOU) vulnerabilities, which may be exploited to compromise the TA.

Prior work on TA security has extensively examined vulnerabilities arising from the absence of rollback protection [17], flawed API design [16], and improper handling of untrusted data [9], [15], [34]. Separately, prior work on automatic TOCTTOU vulnerability detection has focused on kernels or enclaves with source-code access [33], [43], [41], [19], and does not transfer to proprietary, binary-only TAs. Thus, shared memory handling in TAs remains unexplored.

This gap persists due to two fundamental challenges. **First**, there is no system to automatically detect double fetch vulnerabilities in proprietary, binary-only TAs. **Second**, the lack of such a detection capability has prevented a systematic study of double fetches in TAs. Furthermore, due to their proprietary nature, it is unknown which commercial TEEs actually support zero-copy shared memory.

To address the first challenge, we present ScHMuzz, a three-stage fuzzer to automatically detect crashes that are due to double fetches from shared memory in proprietary binary-only TAs. ScHMuzz leverages coverage-guided fuzzing in an emulator to explore the target and discover inputs that trigger overlapped fetches from shared memory. ScHMuzz creates

* These authors contributed equally to this work.
† Author completed work while at EPFL; now with Google.

[1]For brevity, we refer to normal-world client applications as apps. In practice, Android apps typically do not communicate with TAs directly; instead, privileged user-space services proxy communication between Android apps and the TA.

Network and Distributed System Security (NDSS) Symposium 2027
22 – 26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.231920
www.ndss-symposium.org

a snapshot just before the second fetch, which, after deduplication, results in a snapshot for each uniquely discovered overlapped fetch location in the target. Leveraging Fetch-Anchored Fuzzing, ScHMuzz determines if a given overlapped fetch leads to a vulnerability by fuzzing the double-fetched value. Finally, in the Distillation stage, ScHMuzz determines whether a crash is due to a shared memory double fetch, or may be triggerable without shared memory.

To address the second challenge, we conduct the first ecosystem-wide study of double fetches in GlobalPlatform-compliant TAs. To this end we create a dataset consisting of 66 unique up-to-date TAs from five TEEs. This dataset is a representative snapshot of TAs currently deployed on current Commercial-Off-The-Shelf (COTS) devices. We manually analyze all TAs to determine which TAs operate on shared memory directly, instead of a local copy. We find that 30 TAs do not copy the shared memory buffer to a local copy and are thus potentially affected by shared memory double fetches. We then apply ScHMuzz to these 30 TAs. ScHMuzz detected overlapped fetches in 23 TAs. More importantly, ScHMuzz discovered 62 crashes due to shared memory double fetches. After deduplication and triage, these crashes result in six 0-day TOCTTOU vulnerabilities in five TAs. These vulnerabilities lead to memory corruptions such as stack-based overflows or double frees and affect all TEEs in our dataset. We reproduce these vulnerabilities on COTS devices and demonstrate that four out of five major TEEs support zero-copy shared memory. We have responsibly disclosed all discovered vulnerabilities to the affected vendors, which have acknowledged our reports.

All TAs in our dataset are written in memory-unsafe languages (C/C++). However, there is a push in the TEE space for TAs written in memory-safe languages such as Rust [40], [7]. Applying ScHMuzz to these TAs, we find that even TAs written in Rust operate on shared memory and discuss how shared memory double fetches break Rust's safety guarantees.

Our findings demonstrate that double fetches due to shared memory are an important security issue for TAs (both commercially deployed TAs and future TAs written in Rust). We hypothesize that the root cause of the vulnerabilities we discovered is the TA developers' unawareness of zero-copy shared memory. We propose an extension to the GlobalPlatform specification that requires explicit opt-in for zero-copy shared memory. Our mitigation successfully fixes the vulnerabilities found by ScHMuzz, without requiring modifications of the affected TAs. We demonstrate the feasibility of our mitigation by extending OP-TEE, the de facto TEE reference implementation. We are in discussion with the GlobalPlatform consortium to amend the specification with our extension. In summary, we make the following contributions:

- **ScHMuzz**: a fuzzer to automatically detect crashes due to shared memory double fetches in binary-only TAs.
- **Ecosystem-wide Analysis**: Conducting an ecosystem-wide study on a representative corpus of COTS TAs with ScHMuzz and demonstrating the prevalence of double fetches and discovering 0-day TOCTTOU vulnerabilities.
- **On-device Validation**: Reproducing the TOCTTOU findings on COTS devices and revealing widespread zero-copy shared memory support among all major TEEs.
- **Long-term Security Implications**: Demonstrating that even TAs written in a memory-safe language are affected by shared memory double fetches.
- **Specification-level Mitigation**: Proposing an extension to the GlobalPlatform specification, which makes shared memory opt-in, mitigating all detected vulnerabilities.

Our implementation of ScHMuzz and our mitigation are open source and available at https://github.com/HexHive/taemu.

II. Background

TEEs deployed on modern Android smartphones are based on ARM TrustZone. TrustZone enables hardware-enforced partitioning of the execution context into the secure world and the normal world. Code running in the normal world is isolated from the secure world and does not have access to the memory of the secure world. Only code signed by the phone vendor runs in the secure world. Code running in the secure world is referred to as the TEE, comprised of TAs and the TEE OS. TAs are user space applications, each of which encapsulates a specific TEE use-case, such as DRM or credentials.

Apps communicate with TAs to access TEE functionality. Nowadays, most apps and TAs leverage the GlobalPlatform APIs, a set of standardized API specifications that enable interoperability for apps/TAs between different TEE implementations. The logical channel defined by the GlobalPlatform API allows an app to call a command handling function (`TA_InvokeCommandEntryPoint`) in the TA. The arguments passed by the app to this function include the `commandID`, specifying which command is invoked, `param_types`, specifying the parameter types, and an array of up to four parameters, `params`. Each parameter may either be a `value` (two 32-bit integers) or a `memref` (a pointer to a memory buffer and its size). Listing 1 shows a TA's `TA_InvokeCommandEntryPoint` function. The TA retrieves the input buffer from the app on Line 13.

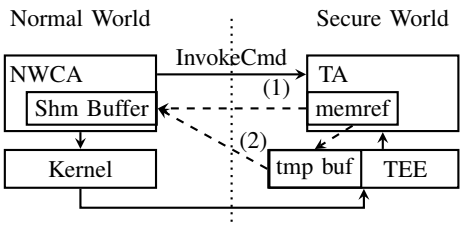


Figure 1: Shared memory facilitates communication between an app and a TA. In scenario (1), double fetch vulnerabilities are possible, as the `memref` buffer pointer of the TA points to the physical page of the buffer in the app. In scenario (2), the TEE makes a local copy of the buffer, and the TA's `memref` buffer points to that local copy.

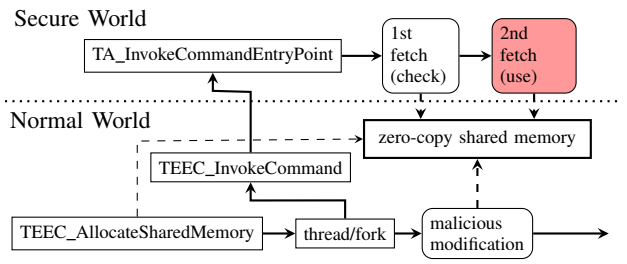


Figure 2: TOCTTOU attack on zero-copy shared memory between the normal and secure world. `TEEC_AllocateSharedMemory` and `TEEC_InvokeCommand` are part of the GlobalPlatform API used by the app to communicate with the TEE/TA.

To pass data from the normal to the secure world, physical pages are shared between the normal and secure world, henceforth referred to as shared memory. When the app invokes the command handler of a TA with a `memref` parameter, the kernel driver forwards the physical addresses of the app-allocated memory buffer to the TEE, which itself maps these physical pages to access the data from the normal world. The TEE implementation may map the physical pages directly into the TA's address space, or it may first copy the contents of the app's memory buffer to a temporary buffer. In the first case, the pointer in any given `memref` parameter points to memory that is shared with the app. As a consequence, modifications of the buffer contents by the app, during the execution of `TA_InvokeCommandEntryPoint`, are also visible to the TA in the secure world. Figure 1 gives an overview of how the TEE passes buffers from the app to the TA's `memref` buffer.

We refer to the definitions of DEADLINE [43] for double fetches. The precondition for **double fetch vulnerabilities** is an **overlapped fetch**, where data is read from the same shared memory range after the range has been previously read/written to. An overlapped fetch itself may not lead to a security violation. There needs to be a **relation** between the shared memory regions accessed by the two fetches. If the value of the second fetch destroys this relation due to changes between fetches, it leads to **double fetch bugs**. These double fetch bugs may turn into security vulnerabilities depending on the context of the double fetch. A canonical example is a TOCTTOU vulnerability, where the target program uses the first fetch to validate a value in shared memory and the second fetch to subsequently use that value. Figure 2 shows how such a TOCTTOU vulnerability is triggered.

The TA in Listing 1 has three overlapped fetches (we assume the value of `in->size` is fetched from memory on every access). The overlapped fetch when handling command 1, on Line 21, while technically a double fetch bug due to breaking the relation between the check and print, is not a memory safety violation, as the second value is simply used for logging. However, the overlapped fetch when handling command 2 is a TOCTTOU vulnerability, as the second fetched value is used as the size in the `memcpy` operation.

```
1  typedef struct in_data{
2      size_t size;
3      char data[0];
4  } in_data;
5
6  int* global = NULL;
7
8  TEE_Result TA_InvokeCommandEntryPoint(
9    int commandID, int param_types, TEE_Param params[4])
10 {
11   if (param_types != 0x7) return -1;
12
13   in_data* in = (in_data*)params[0].memref.buffer;
14   size_t in_buf_sz = params[0].memref.size;
15
16   size_t in_size = in->size;
17   if(in_size > in_buf_sz || in_size > 0x2000) return 0;
18
19   char* copy_buf = TEE_Malloc(in_size, 0);
20   if(commandID == 1){
21     printf("[debug] called with size %zu\n", in->size);
22   } else if (commandID == 2){
23     memcpy(copy_buf, in->data, in->size);
24   } else if (commandID == 3 && in->size == 1234){
25     *global = 1;
26    }
27   return 0;
28 }
```

Listing 1: A TA's `TA_InvokeCommandEntryPoint` function. The TA expects a single argument buffer, containing an `in_data` structure. While executing this function, the TA operates directly on the shared memory (`params[0].memref.buffer`) and ends up having three overlapped fetches. The first fetch is marked in blue and the second fetches are marked in red. While the double fetch on line 21 is harmless, the double fetch on line 23 may lead to a heap buffer overflow if a malicious app modifies the value of `in->size` between lines 16 and 23.

The size checks and allocations are done on the stale value retrieved with the first fetch, while the `memcpy` operation uses the second fetched value, which may have been changed between Lines 16 and 23 to a value larger than the first fetch, leading to a heap buffer overflow. The overlapped fetch on Line 24 is not a double fetch bug, but still leads to a crash. We discuss how ScHMuzz distinguishes between these three cases in Section III.

As illustrated by the TA in Listing 1, TOCTTOU vulnerabilities may lead to memory corruption. Memory corruption vulnerabilities in TAs are extremely interesting to attackers as they may be exploited from the normal world to achieve code execution in the vulnerable TA [3], [10], [11], [14], [17], [16], [25], [26], [9], [30], [20], allowing the attacker to pivot to the secure world. Furthermore, while in other contexts, access to shared memory is made explicit to the developer, such as `copy_from_user` in the Linux kernel, in the TA context, there is no such mechanism, and it is up to the TA developer to securely handle the `memref` buffer pointer.

Threat Model, Our threat model assumes that the attacker can interact with the TEE kernel driver. This encompasses the

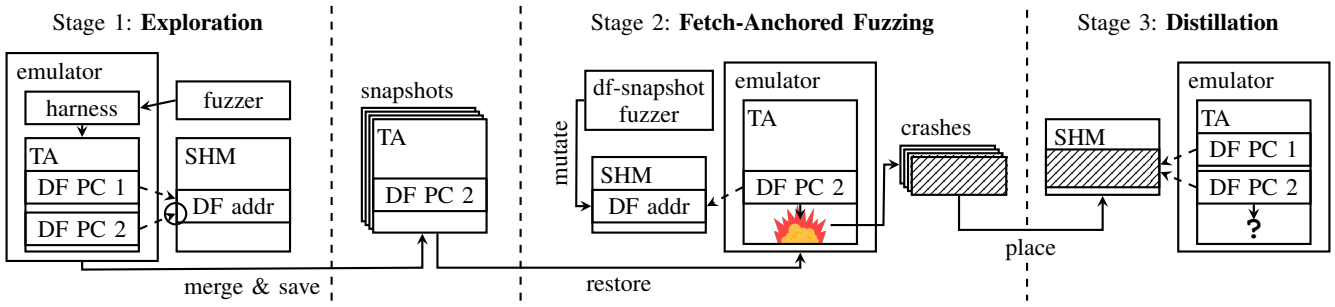


Figure 3: Overview of ScHMuzz and its three phases. To analyze a TA, ScHMuzz requires the TA binary itself and a fuzzing harness. In the *Exploration* phase, the TA is fuzzed with the harness. Snapshots are created after observing an overlapped fetch. These snapshots are passed to *Fetch-Anchored Fuzzing*, where the TA state is restored just before the second fetch and the value fetched from memory is fuzzed. All resulting crashes are then passed to *Distillation*, where ScHMuzz determines if a crash is only triggerable due to shared memory double fetches.

official ARM TrustZone threat model (compromised normal world OS) but also user space root or any high-privileged service, for example gatekeeperd, with an SELinux context that gives access to the TEE kernel driver. The attacker has achieved this level of control after escalating privileges through a local or remote exploit chain.

III. Automatic Detection of Double Fetches in TAs

We observe the absence of an automated approach for detection of shared memory double fetch vulnerabilities in TAs. Without such a capability, conducting a large scale study of double fetches in TAs is not feasible. To enable our large-scale study, we design ScHMuzz, a three-stage fuzzer for automatic double fetch vulnerability detection in binary-only TAs. We will leverage ScHMuzz to conduct the first ecosystem-wide study of TOCTTOU vulnerabilities in TAs.

ScHMuzz is based on dynamic analysis. While static analysis can, in principle, cover an entire program, it produces a large number of false positives: detected double fetches may not be triggerable in practice due to complex path constraints, limited attacker control, or execution contexts that are unreachable from the normal world. Such findings are of little security relevance, particularly for TAs, where attacker interaction is restricted to a small, well-defined attack surface. Instead, we deliberately choose a tight under-approximation by leveraging dynamic analysis starting from TA entrypoints that are reachable by an app. Furthermore, static analysis for closed-source targets is more complex due to decompilation and might suffer from incomplete/incorrect results.

ScHMuzz works in three stages. 1. **Exploration**: Identify overlapped fetches from shared memory in attacker-reachable TA code paths. 2. **Fetch-Anchored Fuzzing**: Determine whether mutating an identified double-fetched value exposes a vulnerability. 3. **Distillation**: Validate whether a given crash is caused by shared-memory double fetching.

To enable the introspection and runtime environment, needed for coverage-guided fuzzing and shared memory tracing of TAs, we build ScHMuzz on top of TÄMU [31], an emulator for GlobalPlatform-compliant TAs, which itself is built on top of Qiling [4], and Unicorn [6]. TÄMU uses high-level emulation to rehost TAs, hooking API invocations of the emulated TA, stubbing or implementing the API's functionality. In this manner, the underlying TEE OS is abstracted away and allows running proprietary arm64 TAs on x86 machines.

A. Exploration

In the Exploration phase, the target TA is fuzzed, and accesses to shared memory are traced to discover overlapped fetches. Bochspwn [24] demonstrated that detecting overlapped fetches in binary-only targets (Windows kernel) dynamically is possible by running the target in an emulated environment. ScHMuzz's Exploration phase extends Bochspwn's approach to target GlobalPlatform-compliant TAs and enables ScHMuzz's second and third stages. Fuzzing allows ScHMuzz to explore all attacker-exposed functionality regardless of whether or not the functionality is used during benign usage. We reuse TÄMU's grey-box fuzzing component to drive exploration.

As defined by DEADLINE [43], overlapped fetches can only lead to vulnerabilities if the two fetches are related. To avoid false positive overlapped fetches, dynamic approaches need a way to determine if two fetches from the same address may be related. Bochspwn marks overlapped fetches as potentially related either heuristically, if they take place in a short time frame of each other, or if they take place while handling the same system call. Other systems for dynamic double fetch detection, such as DECAF [33], do not attempt to establish related overlapped fetches. ScHMuzz leverages the TA's normal-world-facing API, `TA_InvokeCommandEntryPoint`, whose execution naturally operates on the same logical `memref` buffer, allowing potentially related overlapped fetches to be identified without relying on heuristics.

To implement Exploration, we use Unicorn's memory access hooks to monitor accesses to the input `memref` buffers. Whenever ScHMuzz observes a potentially related overlapped fetch, a snapshot of the program is taken just before the second fetch. Contiguous second fetch accesses are merged into one

logical overlapped fetch. The resulting set of snapshots is further deduplicated based on the program trace, resulting in a snapshot for each uniquely discovered overlapped fetch.

The Exploration phase applied to the TA from Listing 1 results in three snapshots, one for each double fetch. Snapshots are taken on Lines 21, 23, and 24, just before `in->size` is fetched from the shared memory for a second time. These overlapped fetches are detected since the second fetch of `in->size` (fetching 8 bytes from offset 0) overlaps with the first shared memory read on Line 16.

B. Fetch-Anchored Fuzzing

In the Fetch-Anchored Fuzzing phase, ScHMuzz determines if a given overlapped fetch leads to a memory corruption vulnerability. Static approaches such as DEADLINE [43] can determine if a given overlapped fetch results in a bug; however, all proposed approaches require source code. Dynamic approaches either leave triage of overlapped fetches to the analyst [24] or are not designed to target a specific double fetch [33], [19], [13].

ScHMuzz addresses both challenges dynamically by fuzzing the values read during the second fetch. By fuzzing the second value, ScHMuzz determines whether tampering leads to memory corruption. Snapshots from the previous phase are used to restore the program state immediately before the second fetch, and fuzzing inputs are written into shared memory to replace the double-fetched region. This models a shared-memory attack in which an attacker wins the race and substitutes the data. The fuzzer then explores program states reachable after the race. Any crashes observed during fuzzing indicate potentially vulnerable double fetches. To detect memory corruptions, ScHMuzz checks if the TA exited due to a Unicorn MemoryAccess exception. Furthermore, ScHMuzz implements a binary-only ASAN for heap memory, similar to BaseSAFE [29], by hooking memory allocation and deallocation APIs. When a memory corruption is detected, the crashing seed is stored alongside the corresponding snapshot.

For the TA in Listing 1, the double fetch snapshot on Line 23 effectively ends up fuzzing the size of the `memcpy` operation, which quickly leads to a crash due to a heap overflow/invalid memory access.

C. Distillation

It is important to note that not all crashes found during Fetch-Anchored Fuzzing are necessarily due to shared memory double fetches. They may be triggerable without replacing the shared memory value in between fetches, i.e., an overlapped fetch resulting in an unrelated vulnerability. None of the existing dynamic double fetch detection systems addresses this issue. An example of such an overlapped fetch is shown on Line 24 in Listing 1. The null pointer dereference on Line 25 may be triggered during Fetch-Anchored Fuzzing, since the fuzzer is effectively only fuzzing the comparison to `1234`. However, the crash can be triggered without shared memory. The Distillation stage filters out crashes that are triggerable without shared memory. This step is crucial for our large-scale study: it systematically eliminates unrelated crashes and allows the analyst to focus on crashes related to shared memory.

In the Distillation stage, we model each execution as a two-fetch trace v_x, v_y, where v_x and v_y denote the values observed by the TA at the first and second fetches, respectively. Let $V_x, V_y \subseteq V$ be the sets of crash-relevant values at the two fetch sites, where V denotes the complete value domain of the fetched object. The crashes exposed after ScHMuzz are classified into five scenarios. For each scenario $i \in \{A, B, C, D, E\}$, a crash over the observed value pair $(v_x, v_y) \in V \times V$ is characterized by an indicator predicate $\mathrm{Crash}_i : V \times V \to \{0, 1\}$, where $\mathrm{Crash}_i(v_x, v_y) = 1$ iff the pair triggers a crash under scenario i.

Thus the five scenarios are characterized as follows:

$$\mathrm{Crash}_A(v_x, v_y) \;=\; \mathbb{I}[\, V_x = V \;\wedge V_y = V \,], \tag{A}$$

$$\mathrm{Crash}_B(v_x, v_y) \;=\; \mathbb{I}[\, v_x \in V_x \;\wedge V_y = V \,], \tag{B}$$

$$\mathrm{Crash}_C(v_x, v_y) \;=\; \mathbb{I}[\, v_y \in V_y \;\wedge V_x = V \,], \tag{C}$$

$$\mathrm{Crash}_D(v_x, v_y) \;=\; \mathbb{I}[\, v_x \in V_x \;\wedge\; v_y \in V_y \;\wedge\; V_x \cap V_y \neq \emptyset \,], \tag{D}$$

$$\mathrm{Crash}_E(v_x, v_y) \;=\; \mathbb{I}[\, v_x \in V_x \;\wedge\; v_y \in V_y \;\wedge\; V_x \cap V_y = \emptyset \,]. \tag{E}$$

In Scenario (A) and Scenario (B), v_y does not influence whether the TA crashes. Fetch-Anchored Fuzzing only fuzzes v_y and thus can not trigger the crash. The crash is only triggerable in Exploration. However, ScHMuzz discards snapshots for crashing seeds with overlapped fetches in Exploration and thus these crashes have already been filtered and do not reach Distillation. In Scenario (C), only v_y influences whether the TA crashes. This crash may be found either in Exploration or Fetch-Anchored Fuzzing. As discussed, crashes found during Exploration are not propagated further in ScHMuzz's pipeline. Crashes arising from this scenario found during Fetch-Anchored Fuzzing are passed to Distillation. Finally, in scenarios (D) and (E), both v_x and v_y influence whether the TA crashes. ScHMuzz can find these crashes by first discovering v_x during Exploration and v_y during Fetch-Anchored Fuzzing, which are then passed to Distillation. Scenario (E) only crashes if v_x and v_y are different and is thus only triggerable with shared memory double fetching. Scenario (D) on the other hand has at least one value pair where v_x is equal to v_y and can be triggered without shared memory double fetches. The double fetch on line 24 in Listing 1 is an example of scenario (D). All crashes passed to Distillation are from either scenario (C), (D), or (E).

Distillation determines if a crash is in (D) or (E) by replacing v_x with v_y on the first fetch. If the TA still crashes, the crash can not be in (E) and is thus filtered out. If the TA does not crash, the crash is either from (D) or (E) since in (C) v_x should not influence the crash. All crashes marked by Distillation to be in (D) or (E) require shared memory double fetching to be triggered. Determining whether a crash is from scenario (D) or (E) after Distillation requires manual triage. Crashes from scenario (D) are false positives of ScHMuzz

Dataset			Exploration			Fetch-Anchored Fuzzing	Distillation
TEE	# TAs	# TAs w/o Local Copy	# TAs with Overl. Fetches	# Overl. Fetches	# Overl. Fetches merged	# Crashes	# Crashes Distilled
TEEGris	33	9	7	33	33	32	32
QSEE	4	3	2	866,145	1,670	6	6
Kinibi	6	3	2	3,301	3,256	123	7
MiTEE	12	10	8	41,918	10,469	44	8
Beanpod	11	5	4	1,852	1,804	125	9
all	66	30	23	913,249	17,232	330	62

Table I: Overview of the results of our large-scale study of shared memory usage and vulnerabilities in TAs. Each merged overlapped fetch corresponds to a snapshot that is fuzzed in Fetch-Anchored Fuzzing.

since there exists a value pair that triggers the crash without shared memory. However, applying ScHMuzz to our dataset in Section IV we did not observe such a false positive.

To implement Distillation, the crashes from Fetch-Anchored Fuzzing are replayed. For each crash, we reuse the original seed discovered during Exploration that led to the double-fetch scenario, and inject the crashing input from Fetch-Anchored Fuzzing into the corresponding shared-memory region prior to execution of `TA_InvokeCommandEntryPoint`. The program is then executed without further modifications to shared memory. If the crash reoccurs under these conditions, it indicates that the vulnerability does not depend on concurrent shared-memory modification.

For the overlapped fetch on Line 24 in Listing 1, the seed which executed `commandID 3` is replayed, and at the start of `TA_InvokeCommandEntryPoint`, `in->size` is set to `1234`. This will also trigger the null pointer dereference, allowing ScHMuzz to automatically determine that this crash is not due to a double fetch.

IV. Ecosystem-wide TA Double-Fetch Study

To understand whether up-to-date TAs have overlapped fetches and suffer from TOCTTOU vulnerabilities, we conduct an ecosystem-wide study of shared memory usage in TAs. We first build a dataset that captures a snapshot of the current GlobalPlatform-compliant TAs across all major TEEs. We then manually analyze all TAs to identify TAs operating on shared memory. Finally, we apply ScHMuzz to these TAs to detect overlapped fetches and TOCTTOU vulnerabilities.

a) Dataset: We build a representative dataset of up-to-date COTS TAs, which we will analyze. Our dataset includes *all* major TEEs used on modern Android devices except for Huawei's iTrustee and Google's Trusty. We exclude iTrustee due to the TAs being shipped encrypted in the firmware and Google's Trusty, as it does not use the GlobalPlatform APIs. For each remaining TEE, we download the latest firmware images from five devices released between 2024 and 2026, spanning a diverse set of system-on-chips and vendors. We extract the included TAs and deduplicate them based on the TA filename, resulting in 66 unique GlobalPlatform-compliant TAs. Prior work on ecosystem TA security [16] found 131 unique TAs across 2016–2024; our 2024–2026 dataset already contains 66 unique TAs, indicating a comparable scale and a representative snapshot of current deployments (see Table I).

b) Local Copy of Shared Memory Buffer: One simple solution to prevent shared memory double fetch vulnerabilities is to copy the data from the shared memory to a local buffer. Applying ScHMuzz only makes sense if there is a reasonable chance that the TA operates on shared memory. For each TA in our dataset, we manually check if the TA makes a local copy of the entire shared `memref` input buffer in the `TA_InvokeCommandEntryPoint` function. If this is the case, we exclude this TA from further analysis as it can not be affected by double fetches.

We find that 30 TAs do not immediately make a local copy of the `memref` buffer and thus are potentially affected by double fetches, see column three in Table I.

c) Applying ScHMuzz to COTS TAs: For this campaign, we apply ScHMuzz to the 30 TAs operating on shared memory. For ScHMuzz's Exploration stage, we need a harness to fuzz the TA (ScHMuzz uses TÄMU's fuzzing framework that requires a harness to deserialize fuzzer bytes into arguments for `TA_InvokeCommandEntryPoint`). We manually write a fuzzing harness for these 30 TAs that can invoke each command of the TA with the required parameters. Our fuzzer uses the first bytes of the fuzzer input to determine the `commandID`. Based on the `commandID`, the harness sets up the `params` array, filling each required member with fuzzer data. For `memref` parameters, the fuzzing bytes are written to `memref.buffer` and `memref.size` is set appropriately.

We fuzz each TA five times for 24 hours in the Exploration phase to account for the randomness of coverage-guided fuzzing and to ensure stable coverage and snapshot discovery. We then merge the obtained snapshots and pass them to Fetch-Anchored Fuzzing and Distillation. In Fetch-Anchored Fuzzing, we fuzz each snapshot for 15 minutes each, which we found sufficient to either trigger a memory corruption or reach coverage saturation. We run our experiments on an Ubuntu Linux server with a Xeon E5-2680 CPU and 256 GB of RAM.

Table I shows the results of the campaign. In total, ScHMuzz detected overlapped fetches in 23 TAs (75% of the fuzzed TAs). After deduplication and merging the 913,249 overlapped fetches, 17,232 snapshots are passed to Fetch-Anchored Fuzzing. Figure 4 shows the coverage graph from Exploration. We estimate the maximally reachable number of basic blocks by building a CFG from each TA's `TA_InvokeCommandEntryPoint` function and counting the unique entrypoint-reachable basic blocks. After Explo-

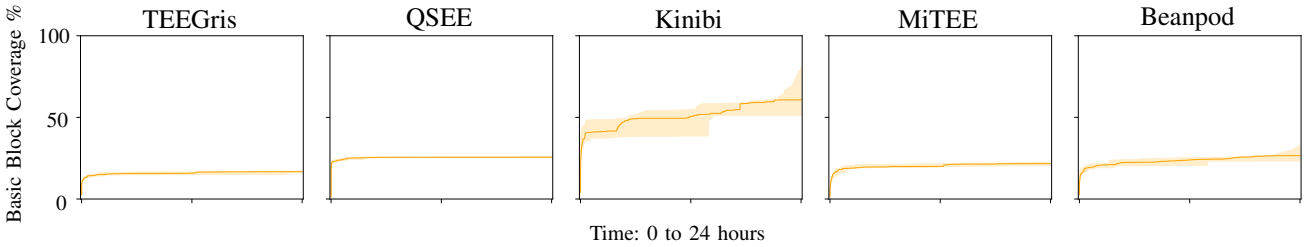


Figure 4: Basic block coverage observed in Exploration. The x-axis denotes the time, and the y-axis is the percentage of covered basic blocks reachable from `TA_InvokeCommandEntryPoint`.

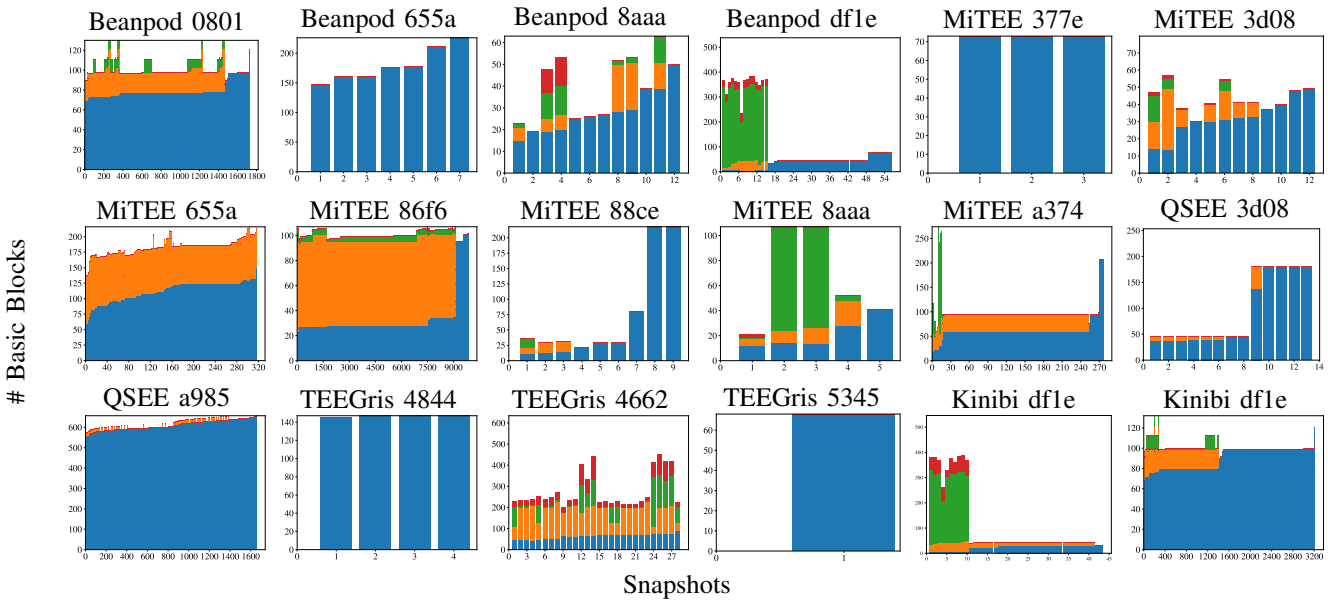


Figure 5: The basic blocks discovered during Fetch-Anchored Fuzzing. Each graph shows a given fuzzed TA. Each bar corresponds to the basic blocks discovered during Fetch-Anchored Fuzzing for a given snapshot. The y-axis denotes the number of covered basic blocks. are the basic blocks executed until the second fetch. are basic blocks also covered by the seed that originally triggered the double fetch. are basic blocks also covered during Exploration. are basic blocks *only* covered during Fetch-Anchored Fuzzing.

ration, we achieve between 15 - 30% coverage of basic blocks reachable by an app. We discuss how this coverage may be further improved in Section VIII.

In Fetch-Anchored Fuzzing, we fuzz the 17,232 snapshots each for 15 minutes for a total of 4,330 hours. Figure 5 shows the basic blocks covered during Fetch-Anchored Fuzzing for each snapshot. We distinguish between four types of covered basic blocks. Especially interesting are the harnesses which covered basic blocks in Fetch-Anchored Fuzzing, which were not covered during Exploration, i.e., ScHMuzz discovered code only reachable due to double fetches. Fetch-Anchored Fuzzing resulted in 330 crashes. With Distillation, we narrow these crashes down to 62 crashes only triggerable with shared memory. We manually triage these distilled crashes and discover six TOCTTOU vulnerabilities in five TAs. All distilled crashes result from a TOCTTOU vulnerability. Multiple distilled crashes are due to the same TOCTTOU vulnerability, see Table II for an overview of the discovered vulnerabilities. We further analyzed the remaining 268 non-distilled crashes and found that all of them can be triggered without shared memory, arising from conventional bugs in handling input from an untrusted source, confirming that Distillation correctly filters non-shared memory related bugs. Note that ScHMuzz may discover non-double fetch vulnerabilities both during Exploration and Fetch-Anchored Fuzzing. We have reported *all* discovered crashes to the affected vendors.

V. Zero-Copy Shared Memory Support in Commercial TEEs

Our implementation of ScHMuzz assumes that `memref` buffers are zero-copy shared memory. However, as discussed in Section II, the TEE implementation may make a local copy of the buffer. To demonstrate that TEEs deployed on COTS devices support zero-copy shared memory, and consequently that shared memory double fetches are a real security threat

TEE	TA UUID	TA Name	DF Vulnerability	Reproduced on Device
MiTEE	377ee4e8-af0e-474f-a9d636a9268fe85c	SoterApp	stack overflow	✓
TEEGris	00000000-0000-0000-0000-4662436b6d52	FbSkmR	oob read	✓
MiTEE,QSEE	3d08821c-33a6-11e6-a1fa089e01c83aa2	VSIMApp	oob write	✓*
MiTEE,QSEE	3d08821c-33a6-11e6-a1fa089e01c83aa2	VSIMApp	double free	✓*
MiTEE	88ce8e6b-8646-4092-bb78faf5b55ff4df	Mlipay	oob read	✓
Kinibi,Beanpod	08010203000000000000000000000000	ifaa-key	oob read	✗†

Table II: The vulnerabilities in commercial TAs found with ScHMuzz. (*) Only on MiTEE, because we did not have a QSEE testing device supporting this TA. (†) None of our testing devices supports the TA.

TEE	Phone	Zero-Copy Shm
TEEGris	Samsung A56 5G	✓
TEEGris	Samsung A16	✓
TEEGris	Samsung S22	✓
TEEGris	Samsung S10	✓
QSEE	OnePlus 12R	✓
QSEE	OnePlus Nord 5	✓
Kinibi	Transsion Pova Pro 5	✓
Kinibi	Transsion Infinix Note 12	✓
MiTEE	Xiaomi 15T	✓
MiTEE	Xiaomi Redmi Note 13 5G	✓
Beanpod	Xiaomi Redmi Note 11s 5G	✗
Beanpod	Xiaomi Redmi 9A	✗

Table III: The devices we used to reproduce double fetches. All devices were updated to the newest firmware version.

to commercial TEEs, we need to reproduce the discovered TOCTTOU vulnerabilities on-device. See Table III for the devices we used in this evaluation. All devices are rooted and updated. Our Proof-Of-Concept (POC) app runs on the testing device and attempts to trigger the TOCTTOU vulnerabilities. It implements three functionalities: obtaining a pointer to shared memory, invoking the vulnerable TA functionality, and racing the shared memory access.

To communicate with the TA, our POC uses the GlobalPlatform client library shipped on the device. The GlobalPlatform client library is developed by the TEE vendor to correctly interface with the TEE-specific kernel driver. With this library, GlobalPlatform-compliant apps can use the GlobalPlatform APIs to communicate with the TAs. By leveraging the client library, we can reuse most parts of the POC for different TEEs. To obtain a pointer to shared memory, our POC leverages the `TEEC_AllocateSharedMemory` API to allocate a memory buffer shared with the TA.

We find that both the GlobalPlatform client library for MiTEE and TEEGris do not return a pointer to the actual shared memory in this API. Instead, a pointer to an intermediate buffer is returned, whose content is copied to the actual shared memory (shared with the TA) by the client library when invoking the TA functionality. This way of hiding the true shared memory buffer is not secure against a malicious app and removes the option of zero-copy shared memory for benign use cases. Listing 2, Lines 13 to 19, show how our POC retrieves the pointer to shared memory.

After obtaining the shared memory pointer, our POC spawns

```
1  void* mod_thread(void* shm){
2    while(1){
3      mod1(shm);
4      mod2(shm);
5    }
6  }
7  typedef struct imp_shm{
8    uint64_t _ignore[3];
9    void* shm;
10 }
11 void trigger_df(TEEC_Context *context,
12       TEEC_Session *session, cmd){
13   TEEC_SharedMemory in_mem;
14   in_mem.size = buf_size;
15   in_mem.flags = TEEC_MEM_INPUT;
16   res = TEEC_AllocateSharedMemory(context, &in_mem);
17   if (res != TEEC_SUCCESS) exit(-1);
18   imp_shm* imp = in_mem->imp;
19   void* shm = imp->shm;
20
21   pthread_t tid;
22   pthread_create(&tid, NULL, mod_thread, shm);
23
24   setup_params(&op, &in_mem);
25   TEEC_InvokeCommand_impl(session, cmd, &op, &err_origin);
26 }
27
28 int main(){
29   TEEC_Context context; TEEC_Session session;
30   open_session(&context, &session);
31   trigger_df(&context, &session);
32 }
```

Listing 2: The relevant components of our POC to trigger double fetch vulnerabilities.

a separate thread that asynchronously modifies the shared memory while the TA is operating on it. This way, we can race the double fetch accesses and eventually trigger the TOCTTOU vulnerability. Lines 1 to 6 in Listing 2 show the racing thread. `Mod1` sets up the shared memory such that the TA's check passes, while `mod2` modifies shared memory to cause a crash during use. If `mod2` happens between the fetches of the TA, the TA reads two different values, thus triggering the TOCTTOU vulnerability.

Finally, the POC invokes the TA's vulnerable functionality. To correctly set up the expected arguments, we leverage the fuzzer-generated seed that triggered the double fetch (Lines 24 to 25 in Listing 2). We write our POC such that triggering the double fetch vulnerability will cause the TA to crash. To observe if we caused the TA to crash, we check the return

code of the `TEEC_InvokeCommand` function.[2] We run the POC in a loop until we observe a crash. We reproduce five out of six discovered double fetch vulnerabilities on-device, see Table II.

```
1 void combine_file_name(char* part1, char* shm){
2   char out_path[0x80];
3   size_t p1_len = strlen(part1);
4   size_t p2_len = strlen(shm);
5   if(p1_len + p2_len > 0x80) return;
6   strcpy(out_path, p1);
7   strcpy(out_path + p1_len, shm);
8   ...
9 }
```

Listing 3: The simplified code of the double fetch stack overflow vulnerability in the SoterApp TA. The `shm` argument of the function points to the shared `memref` buffer.

a) Case Study: Stack-based Overflow MiTEE: The affected TA has a TOCTTOU vulnerability leading to a stack-based buffer overflow. Listing 3 shows the vulnerable code. The TA first obtains the length of the null-terminated string in shared memory during the first fetch. As part of the second fetch, the TA copies a string from shared memory into a stack buffer. To trigger the stack-based buffer overflow, our POC's `mod_thread` sets up the shared memory with a string larger than 0x80. It then repeatedly sets the first byte in the shared memory to 0 and back to the original string value. If the fetches line up, `strlen` is calculated on a zero-length string, but `strcpy` will copy a string larger than 0x80, triggering the stack-based buffer overflow.

```
1 TEE_Result TA_InvokeCommandEntryPoint(.., params){
2   void* buf0 = params[0].memref.buffer;
3   size_t buf0_size = params[0].memref.size;
4   if(*(size_t*)buf0 > buf0_size) return -1;
5   size_t ser_size = *(size_t*)buf0;
6   void* local = TEE_Malloc(ser_size);
7   memcpy(local, buf0, ser_size);
8   ...
9 }
```

Listing 4: The simplified code of the double fetch out-of-bounds read vulnerability in the FbSkmR TA.

b) Case Study: OOB read TEEGris: The affected TEEGris TA has a TOCTTOU vulnerability leading to an out-of-bounds read on the input shared memory buffer. The TA expects the input buffer to contain length-value serialized data. It first checks if the serialized length is not larger than the size of the shared memory buffer at Line 4 (Listing 4). The TA then fetches the length again at Line 5 and copies from the shared memory buffer to a local buffer. The `mod_thread` for this TA's POC repeatedly sets the first `size_t` value in the shared memory to zero or a very large value, which will crash in the `memcpy` function when reading out of bounds of the buffer.

[2]A return value of `0xFFFF3024 (TEE_ERROR_TARGET_DEAD)` indicates the TA has crashed. [2]

c) Double Fetches in other TEEs: For the TEEs for which we did not reproduce a crash on-device (Kinibi, Beanpod, and QSEE), due to the vulnerable TA not being supported on our testing devices, we still want to test if the TEE supports zero-copy shared memory. To this end, we manually analyze the overlapped fetches found in TAs from these TEEs to find fetches that, if triggered, lead to an observable outcome. We find that both Kinibi and QSEE support zero-copy shared memory, see Appendix A for the details of the double fetches we used. However, Beanpod does not support shared memory. We leverage the exploit from [16] to achieve code execution in a Beanpod TA. Our shellcode repeatedly reads from the `memref` buffer and logs the value. Although we are modifying the value concurrently from our app, we are unable to observe changes in the `memref` buffer accessed by the TA.

The results of this evaluation on the availability of zero-copy shared memory in major TEEs can be found in Table III. *Four* out of the five major TEEs used on Android devices support zero-copy shared memory. As such, TOCTTOU vulnerabilities in TAs deployed on these TEEs are triggerable. This underlines the criticality of TOCTTOU vulnerabilities in TAs.

VI. Shared Memory Handling in Future Memory-Safe TAs

All the COTS TAs analyzed in our dataset are written in C/C++. In the future, stakeholders (vendors, developers) may move to address memory corruption vulnerabilities, the most widely exploited category of vulnerabilities, in TAs, by reimplementing them in a memory-safe language. However, irrespective of the language in which the TA is written, the underlying shared memory API design still exposes the TA to a pointer shared with the untrusted normal world. This pointer may be abstracted by the language, and thus the fact that it is shared is hidden from the developer. To understand how shared memory double fetches affect TAs written in memory-safe languages, we analyze TAs written in Rust. While we did not observe any Rust TAs deployed on COTS devices, we believe that the TEE space will start adopting Rust, just as is the case for other Android components such as parts of the Linux kernel [5] or of the user space [38].

Apache Teaclave [40], [7] is the official SDK for writing TAs in Rust for OP-TEE. OP-TEE is the reference TrustZone TEE implementation and most TEEs deployed on COTS devices are derived from OP-TEE. It follows that Rust TAs for commercial TEEs will be derivatives of OP-TEE's Rust TA SDK. We inspect the Teaclave SDK and find that TAs obtain a mutable `[u8]` slice when accessing the `memref` buffer. This slice is obtained using the `std::slice from_raw_parts_mut` API, with the `memref`'s buffer pointer and size. The Rust documentation states the following in the safety section of `from_raw_parts_mut` [8]:

TA	SHM Operation	# Detected Double Fetches
message_passing_interface-rs	`serde_json::from_slice`	490
eth_wallet	`bincode::deserialize`	0
mnist-rs	`bytemuck::cast_slice`	0
udp/tcp_client-rs*	`core::str::from_utf8`	351
Bitcoin-Wallet-for-Trusted-OS-OPTEE	custom deserialization and signing	1

Table IV: The Rust TAs that operate on shared memory. The "SHM Operation" column denotes either the API called on the shared memory slice or the operations on said slice. (*) We only fuzz the tcp_client TA since the udp TA has an identical operation on shared memory.

> The memory referenced by the returned slice must not be accessed through any other pointer (not derived from the return value) for the duration of lifetime `'a`. Both read and write accesses are forbidden.

This requirement is violated when the mutable slice is derived from a pointer to shared memory, since the app has a pointer to the slice's underlying physical memory and may read/write to it. Thus, for TAs written in Rust, which operate directly on the shared memory slice, the Rust compiler can not give any safety guarantees.

We inspect 28 TAs written in Rust (part of the Teaclave reference examples or independent Github projects [7], [23]) and check if the TAs operate on shared memory directly or make a copy. We find that 22 TAs use APIs such as `to_vec` to make a local copy of the buffer. 6 TAs operate directly on shared memory in Rust, deserializing data directly from the shared memory buffer. We fuzz these TAs with ScHMuzz to understand if the Rust code has double fetches. Table IV shows the results. Four of the six TAs have double fetches.

While ScHMuzz did not trigger any crashes during Fetch-Anchored Fuzzing, manually analyzing the overlapped fetches, we found that in three out of four TAs, these double fetches lead to TOCTTOU logic bugs. The double fetches in the udp/tcp_client TAs happen due to the fact that `core::str::from_utf8` first traverses the shared memory slice to ensure it contains valid UTF-8. Afterwards, the API just returns a pointer to the underlying slice. If the TA afterwards accesses the string, its contents may have been changed to invalid UTF-8. The Bitcoin-Wallet-for-Trusted-OS-OPTEE TA deserializes a `Transaction` object directly from the shared memory buffer. Afterwards, the TA refetches the entire shared memory and signs the transaction data. In between the two fetches, an attacker can modify the shared memory, and thus the TA will sign arbitrary attacker-controlled data instead of a correctly deserialized transaction.

Our results demonstrate that even TAs written in memory-safe languages, such as Rust, are affected by double fetches.

VII. Mitigation

As demonstrated in Section IV, V, and VI, vulnerabilities due to shared memory double fetches are an insidious problem plaguing TAs, both for TAs currently deployed on COTS phones and future TAs written in memory-safe languages. Analysis of the discovered vulnerabilities revealed that the affected TAs do not rely on zero-copy shared memory for their functionality. Instead, it appears the TA developer did not consider the `memref` buffer to be shared.

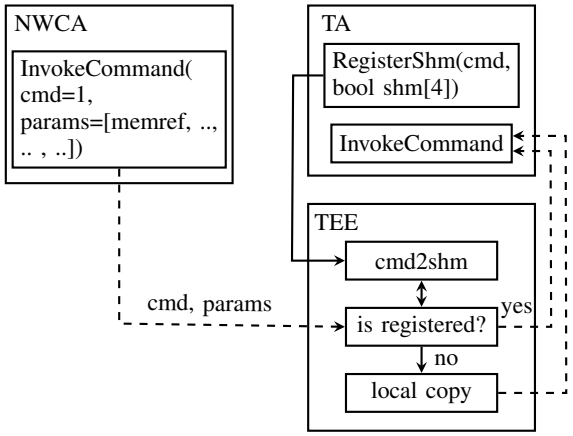


Figure 6: An overview of our mitigation. A TA may register `memref` buffers of one or more specific commands as zero-copy shared memory using `RegisterShm`. When the app invokes the TA, the TEE runtime checks if the input buffers were registered, creating a local copy if not.

This is not the developer's fault. Instead, it is an oversight of the GlobalPlatform specification and the TEE implementations, which obscure the fact that `memref` buffers may be shared. First, TEE implementations may or may not implement zero-copy shared memory. A TA developed for a TEE without zero-copy shared memory may later be deployed on a TEE with shared memory, or the TEE itself may switch to zero-copy shared memory in a later version. Furthermore, while there is a warning in the GlobalPlatform specification [2], the proposed solution is ineffective. The document suggests using `TEE_CheckMemoryAccessRights` to check if a memory region is shared with an app. By *not* setting the `TEE_MEMORY_ACCESS_ANY_OWNER` flag, the API returns -1 if the memory is shared. It is unclear to us whether TEEs support this flag, since we have not found a single TA in our dataset calling this API to check for shared memory.

Instead of leaving shared memory handling to developers and obscure APIs, we propose to mitigate this design weakness with an extension to the GlobalPlatform specification. This extension makes zero-copy shared memory explicit, removing ambiguity on shared memory and ensuring that double fetches

on memref buffers are not a security issue by default. If a TA developer wishes to make use of zero-copy shared memory, the developer needs to explicitly register a commandID, memref buffer combination as zero-copy shared memory with the TEE runtime. For memref buffers not registered, the TEE runtime will create a local copy of the buffer and pass the local buffer to the TA. This extension changes the shared memory model to a fail-closed design, meaning that TAs that do not consider shared memory will **not** have double fetch vulnerabilities. Essentially, it makes the two scenarios outlined in Figure 1 explicit to the developer. Our mitigation fixes all vulnerabilities discovered by ScHMuzz, without changes to the existing TAs.

To demonstrate the feasibility of our mitigation, we implement it for OP-TEE, the reference GlobalPlatform-compliant TEE implementation. We add a new API TEE_RegisterShm, which allows a TA to register an entry in the params array for a given commandID as shared. In its TA_CreateEntryPoint function, the TA registers the memref that needs to be zero-copy shared memory. Listing 5 shows how a TA leverages this new API to request a zero-copy shared memory buffer. When the TA's TA_InvokeCommandEntryPoint function is called, the TEE user space runtime checks for each memref parameter, if for the given commandID, the memref was registered as zero-copy shared memory. If yes, the runtime passes the pointer to the shared memory to the TA. If not, the runtime makes a local copy of the buffer and sets the pointer in memref to the local buffer. Figure 6 shows our mitigation.

Our mitigation incurs no performance overhead for TAs that opt into shared memory. Without opting in, the mitigation roughly doubles the time between normal world invocation and the TA's entrypoint. The overhead stems from making the local copy. It is important to note that the 36 TAs in our dataset that make a local copy incur a similar overhead but require explicit implementation by the developer. One case where a TA breaks due to our mitigation is a TA, whose functionality depends on concurrent buffer modifications, that has not yet been ported to opt into shared memory. Such behavior is not supported by the GlobalPlatform standard and we did not observe any TA in our dataset relying on concurrent modifications.

Our patch to OP-TEE adds this mitigation in 140 lines. We have submitted our mitigation proposal to the GlobalPlatform alliance and are in active discussion with them to update the GlobalPlatform TEE internal core specification to address shared memory.

```
1 TEE_Result TA_CreateEntryPoint()
2 {
3     bool paramsShm[4] = {true,false,false,false};
4     TEE_RegisterShm(3, &paramsShm);
5 }
```

Listing 5: The updated TA_CreateEntryPoint function of the TA from Listing 1, which explicitly registers the first memref buffer when handling command 3 as zero-copy shared memory. All the other memref buffers are not zero-copy, and thus the double fetch bugs in commands 1 and 2 are mitigated.

VIII. Discussion

a) Manual Effort: Our ecosystem-wide study from Section IV requires manual effort to determine if a TA makes a local copy of the shared memory buffer, and harness the TA for ScHMuzz.

To understand if the TA makes a local copy of the memref buffer, we manually analyze the TA entrypoint function with a decompiler. Following the memref arguments is trivial, and it is immediately obvious if a local copy is created. It took us, on average, 5 minutes to check for shared memory usage for a single TA (5 hours for all TAs in our dataset). To harness the 30 TAs, we further analyzed the TA entrypoint function to extract the commands and expected parameter types. Writing and testing a harness for a given TA took us, on average, 30 minutes to write and test the harness (15 hours in total).

While this represents a non-trivial one-time manual effort of approximately 20 hours, it enabled us to analyze and harness *all* TAs operating on shared memory in our dataset, which constitutes a representative snapshot of currently deployed TAs. Although these two steps could likely be automated, we believe the engineering and validation effort required would not be justified given the relatively modest manual overhead.

b) Beyond Shared Memref Buffers: In this study, we focus on the normal world reachable attack surface exposed by GlobalPlatform-compliant TAs handling shared memory buffers. Besides GlobalPlatform-compliant TAs, Kinibi and QSEE support non-compliant TAs, whose API may also support zero-copy shared memory. Extending ScHMuzz for such TAs is a matter of engineering effort to support loading and fuzzing these TAs in ScHMuzz's emulator component.

Besides the attack surface of TAs, the TEE itself handles shared memory for any other communication with the normal world, such as when loading a TA. We leave auditing these shared memory interactions to future work.

c) ScHMuzz Compared to Reshaping-based Fuzzing Approaches: Reshaping-based fuzzing approaches [13], [19] inject fuzzing inputs at memory reads that access attacker-controlled memory and are capable of dynamically triggering TOCTTOU vulnerabilities. Unlike ScHMuzz, reshaping-based techniques cannot directly target individual double fetch "use" instances. They rely on re-executing seed inputs to probabilistically re-trigger double fetches. In contrast, ScHMuzz's snapshot-based double fetch fuzzing enables specifically targeting double fetches without having to re-execute the target until the second fetch occurs.

To quantify this difference, we measure how often double fetches are triggered during exploration. Table V shows the results across all harnesses for which we detected overlapped fetches, fuzzed for the same time as during Fetch-Anchored Fuzzing for the evaluation in Section IV. Across all harnesses, only 11% of fuzzing iterations actually execute a code path containing a double fetch. Since our study targets double fetch vulnerabilities, reshaping-based approaches would spend the

TA	# Execs	# Execs DF	Execs DF %
SEMeSE	24’364	22260	91%
Mirisikm	75’843	53’027	69%
SoterApp	1’788’240	1’041’636	58%
eID	19’345	5215	27%
GF_TA	14’741’203	2’757’134	18%
FbCkmR	568’237	127’009	15%
Mlipay	490’558	75’528	15%
ifaa-key	32’143’889	2’480’790	7%
fido	11’938	866	7%
paytrigger	812’135	41’193	5%
VSIMApp	3’863’250	117’860	3%
HDCP	52’012	891	1%
OTrPAPP	6’718’937	39’572	0.5%

Table V: The overall executions and executions triggering overlapped fetches resulting from fuzzing the harnesses with overlapped fetches for the same CPU time as during the Fetch-Anchored Fuzzing campaign in Section IV.

majority of their effort on executions that cannot trigger the vulnerability of interest.

In its current form, ScHMuzz is unable to detect nested double fetch vulnerabilities, i.e., double fetches that are reachable only after triggering another double fetch bug. Reshaping approaches can trigger such nested double fetches. Extending ScHMuzz for nested double fetches is possible by collecting snapshots during Fetch-Anchored Fuzzing. We deliberately opted to focus on single double fetch vulnerabilities to ensure we were able to reproduce our findings on COTS devices.

d) Harnessing TAs for Exploration: As pointed out by related work [15], [27], TAs may expect specific input formats in the input buffers or only expose certain functionality in specific states. Our current approach to harnessing does not take this into account; we fuzz one invocation of `TA_InvokeCommandEntryPoint`, directly inserting fuzzer-mutated bytes into the shared memory buffer. As such, the current harnesses are limited in their ability to explore TAs, and the number of overlapped fetches detected in Section IV should be viewed as a lower bound. We leave extending ScHMuzz with type/state-aware harnesses to future work.

e) Non-Memory Corruption Double Fetch Bugs: In its current design, ScHMuzz uses memory safety oracles to detect double fetch vulnerabilities. However, double fetches may trigger security-relevant non-memory corruption bugs, such as logic bugs. ScHMuzz could automatically detect such vulnerabilities if the corresponding bug oracles are implemented in the emulator component. Designing and implementing bug oracles for non-memory corruption bugs in TAs is orthogonal to our work. We choose to focus on double fetches leading to memory corruptions as this category of vulnerability has historically been the most critical one for TA security.

f) Exploitability of TOCTTOU Vulnerabilities in TAs: The six vulnerabilities in our evaluation are memory corruption vulnerabilities, which may be weaponized to achieve code execution and/or extract the TA’s memory. While triggering the vulnerability is not deterministic, since it requires racing the TA’s accesses, for all our findings there is a clear oracle that allows the attacker to determine if the vulnerability was triggered. Past works have leveraged non-TOCTTOU memory corruption vulnerabilities to exploit TAs and there is precedent for exploiting memory corruptions arising from shared memory TOCTTOU bugs: In 2017, Eloi Sanfelix presented an exploit of a TOCTTOU memory corruption vulnerability in Samsung TAs [32]. Exploitability is an orthogonal problem that requires TEE-internal details. Obtaining these requires either a high-fidelity emulation environment or significant on-device engineering. TÄMU lacks this low-level fidelity and the engineering required is out of scope for this work.

g) Applying ScHMuzz to Other Targets: ScHMuzz’s three-stage approach can be ported to any emulated target (for example, kernels, or hypervisors). Porting requires first supporting emulation for the target and addressing target-specific challenges, such as which memory regions are shared and how long fetches are related. Porting ScHMuzz to other non-GlobalPlatform-compliant TEEs requires extending TÄMU to support the TEE-specific emulation first. Afterwards, ScHMuzz can be applied directly.

IX. Related Work

Both industry and academia have identified the importance of TEE/TA and security. Researchers have repeatedly demonstrated the impact of vulnerabilities in TAs [32], [10], [11], [14], [17], [16], [25], [26], [9], [30], [20]. A large number of works propose systems to fuzz TAs [31], [15], [37], [22], [27], [35], [36]. Beyond one-off bugs, GlobalConfusion [16] identifies a type confusion vulnerability arising from a design flaw in the GlobalPlatform API standard. Suciu et al. identified horizontal privilege escalation vulnerabilities in TAs stemming from incorrect session management [39]. Busch et al. discuss missing rollback protection for TAs [17] on COTS devices. Cerdeira et al. [18] systematize common TEE vulnerabilities. Before our work, the only publicly documented instance of a shared memory double fetch vulnerability in a TA was in 2019 by Sanfelix [32], demonstrating an exploit on Samsung Kinibi. Our work complements existing TEE research by systematically studying double fetch vulnerabilities in TAs arising from shared memory.

Double fetches are a well-known problem in other domains, such as the Windows [1] or Linux kernel [41] and often lead to exploitable vulnerabilities [42], [24]. One research thrust to address this problem is to automatically detect double fetch vulnerabilities. Wang et al. studied double fetches in the Linux kernel, using static analysis to detect double fetches and then manually analyzing them to understand when a double fetch becomes a vulnerability [41]. DEADLINE [43] leverages static analysis to detect double fetches and uses symbolic constraint solving to understand if a double fetch is vulnerable. Unlike static approaches, ScHMuzz’s dynamic approach does not require checking if double fetch bugs turn into vulnerabilities, and as long as the fuzzing harness, used during exploration, fuzzes attacker-controlled data, all found double fetch vulnerabilities will be attacker-triggerable. Bochspwn [24] runs the Windows kernel in an emulated

environment and tracks memory accesses from the kernel to user space memory to detect overlapped fetches during normal execution. Overlapped fetches detected by Bochspwn are manually triaged to determine if the double fetch leads to a bug. DECAF [33] uses hardware cache side channels to detect double fetch accesses. Unlike ScHMuzz, DECAF can not reliably or precisely detect double fetches due to using cache side channels, detecting at cache line granularity, leading to a large number of false positives. Reshaping-based approaches such as EnclaveFuzz [19] or Morphuzz [13] may find double fetch vulnerabilities during fuzzing, however as discussed in Section VIII, these approaches may waste fuzzing cycles on fuzzing inputs not triggering double fetches. Another research thrust is mitigating double fetch vulnerabilities [12], [21], [33], [28]. We propose a domain-specific solution that fixes existing double fetch vulnerabilities without changes to existing TAs.

X. Conclusion

In this work, we demonstrated that shared-memory double fetches constitute a previously overlooked but highly critical attack surface in modern TEEs. The absence of prior work is due to the lack of suitable systems for detecting double fetch vulnerabilities in proprietary, binary-only TAs and the uncertainty about real-world zero-copy shared memory support.

To enable an ecosystem-wide study of shared memory vulnerabilities in TAs, we presented ScHMuzz, a fuzzer capable of automatically identifying crashes due to shared memory double fetches in binary-only TAs at scale. We assembled a representative dataset of TAs and applied ScHMuzz to 30 TAs, which operate directly on shared memory. We uncovered six 0-day TOCTTOU vulnerabilities in five TAs that lead to memory corruption.

We reproduced these vulnerabilities on COTS devices and demonstrated for the first time that zero-copy shared memory is widely supported in commercial TEEs, underlining the impact of the attack surface of shared memory TOCTTOU vulnerabilities in TAs.

Beyond existing C-based TAs, we showed that shared memory TOCTTOU vulnerabilities persist even in TAs written in memory-safe languages such as Rust, highlighting that shared memory undermines language-level safety guarantees.

Overall, our results elevate shared memory double fetch vulnerabilities in TAs from an overlooked corner case to an ecosystem-wide threat to the security of modern TEEs. To address this threat, we proposed an extension to the GlobalPlatform specification that makes zero-copy shared memory explicitly opt-in. Our mitigation eliminates all discovered vulnerabilities without requiring changes to existing TAs and is practical to deploy. We have shared our findings and proposal with the GlobalPlatform consortium and made both ScHMuzz and our mitigation open source.

Acknowledgments

This work was supported, in part, by SNSF 10004669 ”LinSpecteur” and SNSF Eccellenza PCEGP2 186974.

Ethics Considerations

a) Vendors: The TAs analyzed in this work are commercially deployed. We discover vulnerabilities in these TAs. Publication of these vulnerabilities may lead to financial or reputational harm to the vendors.

To mitigate this risk, we have responsibly disclosed all discovered vulnerabilities to the affected vendors. By the time of publication, vendors will have had more than 90 days to address the disclosed vulnerabilities.

By adhering to responsible disclosure practices, we provide vendors with sufficient time to patch the vulnerabilities and improve the security posture of their products. We have disclosed all vulnerabilities (both the conventional and TOCTTOU vulnerabilities) and are awaiting feedback from the vendors. Furthermore, we hope that publication of this work encourages vendors to proactively deploy ScHMuzz or our proposed mitigation to prevent future double fetch vulnerabilities.

b) End Users: The affected TAs are deployed on billions of smartphones worldwide. Exploitation of the discovered vulnerabilities may impact end users' privacy and safety.

Only device vendors can deploy patches for these vulnerabilities. Our responsible disclosure, as described above, enables vendors to address the issues. All on-device experiments were conducted exclusively on devices owned and controlled by the authors.

End users are already exposed to these vulnerabilities. Responsible disclosure enables vendors to deploy patches and allows users to update their devices. While patch deployment may be delayed or unavailable on some devices, we believe that public disclosure is necessary to drive long-term improvements. Although publication may draw attention to this class of vulnerabilities, security by obscurity ultimately benefits only malicious actors. We aim to improve the security of TAs by making our findings available to vendors and the research community.

c) TEE Ecosystem and GlobalPlatform: Our mitigation necessitates changes to existing TEE implementations and to the GlobalPlatform API standard, imposing engineering costs.

We engage with the GlobalPlatform consortium to propose a backward-compatible mitigation that does not require modifications to existing TAs. This approach minimizes ecosystem disruption while addressing the root cause of the identified vulnerabilities.

By addressing shared memory safety at the specification level, we prevent entire classes of vulnerabilities and ultimately strengthen the security of the TEE ecosystem.

References

[1] MS08-061: The case of the kernel mode double-fetch. https://www.microsoft.com/en-us/msrc/blog/2008/10/ms08-061-the-case-of-the-kernel-mode-double-fetch, 2008.

[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] Not To Be Trusted - A Fiasco in Android TEEs. https://media.ccc.de/v/39c3-not-to-be-trusted-a-fiasco-in-android-tees#t=1528, 2025.

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

[5] rust_binder: add Rust Binder driver. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=eafedbc7c050c44744fbdf80bdf3315e860b7513, 2025.

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

[7] Apache TeaClave TrustZone SDK. https://github.com/apache/teaclave-trustzone-sdk, 2026.

[8] Rust Docs: Function from_raw_parts_mut . https://doc.rust-lang.org/std/slice/fn.from_raw_parts_mut.html, 2026.

[9] Alexandre Adamski, Joffrey Guilbon, and Maxime Peterlin. A deep dive into samsung's trustzone (part 2). Quarkslab Blog, 2019. Available at: https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-2.html.

[10] 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.

[11] 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.

[12] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. Midas: Systematic kernel TOCTTOU protection. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 107–124, Boston, MA, August 2022. USENIX Association.

[13] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. Morphuzz: Bending (input) space to fuzz virtual devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1221–1238, Boston, MA, August 2022. USENIX Association.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] Liheng Chen, Zheming Li, Zheyu Ma, Yuan Li, Baojian Chen, and Chao Zhang. Enclavefuzz: Finding vulnerabilities in sgx applications. In *Network and Distributed System Security (NDSS) Symposium*, 2024.

[20] Adam Crenshaw. On the nose bypassing huaweis fingerprint authentication by exploiting the trustzone. Derbycon, 2018. Available at: https://www.youtube.com/watch?v=QFFhdqP7Dxg/.

[21] Victor Duta, Mitchel Josephus Aloserij, and Cristiano Giuffrida. SafeFetch: Practical Double-Fetch protection with Kernel-Fetch caching. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1207–1224, Philadelphia, PA, August 2024. USENIX Association.

[22] 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.

[23] joaopeixoto13. Bitcoin-Wallet-for-Trusted-OS-OPTEE. https://github.com/joaopeixoto13/Bitcoin-Wallet-for-Trusted-OS-OPTEE/tree/main, 2023.

[24] Mateusz Jurczyk and Gynvael Coldwind. Identifying and exploiting windows kernel race conditions via memory access patterns. In *Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns*, page 69, 2013.

[25] Daniel Komaromy. Unbox your phone – exploring and breaking samsung's trustzone sandboxes. Ecoparty, 2021. Available at: https://www.youtube.com/watch?v=L2Mo8WcmmZo.

[26] 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.

[27] 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.

[28] Yingqi Luo, Pengfei Wang, Xu Zhou, and Kai Lu. Dftinker: Detecting and fixing double-fetch bugs in an automated way. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 780–785. Springer, 2018.

[29] Dominik Maier, Lukas Seidel, and Shinjo Park. Basesafe: baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, page 122–132, New York, NY, USA, 2020. Association for Computing Machinery.

[30] 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/.

[31] Philipp Mao, Li Shi, Marcel Busch, and Mathias Payer. Tämu: Emulating trusted applications at the (globalplatform)-api layer. In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2026.

[32] Eloi Sanfelix. TEE Exploitation by Example: Exploiting Trusted Applications in Samsung's TEE. In *INFILTRATE 2019*, 2019. Conference talk.

[33] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 587–600, 2018.

[34] 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.

[35] 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.

[36] 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.

[37] Juhyun Song, Eunji Jo, and Jaehyu Kim. Dta: Run trustzone tas outside the secure world for security testing. *IEEE Access*, 12:16715–16727, 2024.

[38] Jeff Vander Stoep. Rust in Android: move fast and fix things. https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html, 2025.

[39] 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.

[40] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. Rustee: Developing memory-safe arm trustzone applications. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, page 442–453, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How Double-Fetch situations turn into Double-Fetch vulnerabilities: A study of double fetches in the linux kernel. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1–16, 2017.

[42] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. Warpattack: Bypassing cfi through compiler-introduced double-fetches. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1271–1288, 2023.

[43] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678, 2018.

APPENDIX A
OBSERVABLE DOUBLE FETCHES IN QSEE AND KINIBI

A. Distinct Return Values in QSEE

```
1  TEE_Result TA_InvokeCommandEntryPoint(.., params){
2      ...
3      return installIDCertifcateCtx(params[0].memref.buffer);
4  }
5  TEE_Result installIDCertifcateCtx(int* shm){
6    if(shm[0] == 0){
7      return -5;
8    }
9    return parse_write_data(shm);
10 }
11 TEE_Result parse_write_data(int* shm){
12   if(shm[0] == 9){
13     if(!parse(shm)){
14       return -5;
15     }
16     return 0;
17   } else {
18     return -24;
19   }
20 }
```

Listing 6: The simplified code of the double fetch in the `A985D3EB-3B52-4D44-BE6C-628A813561E8` TA, which we used to determine if QSEE supports zero-copy shared memory.

```
1  TEE_Result TA_InvokeCommandEntryPoint(.., params){
2    int* buf0 = params[0].memref.buffer);
3    int cmd = buf0[0];
4    if(cmd < 0x100c){
5      ut_pf_log_msg(3,"cmd : 0x%x \n",cmd));
6      cmd = buf0[0];
7    }
8    if(cmd == 0x1009){
9      ut_pf_log_msg(3,"paytrigger_check_key");
10     ...
11   } else if(cmd == 0x1001){
12     ut_pf_log_msg(3, "paytrigger_ta_get_hmackey");
13     ...
14   } else {
15     ...
16   }
17 }
```

Listing 7: The simplified code of the double fetch in the `df1edda8627911e980ae507b9d9a7e7d` TA, which we used to determine if Kinibi supports zero-copy shared memory.

Listing 6 shows the double fetch, which we used to test if QSEE supports zero-copy shared memory. The TA fetches the first four bytes from the shared memory twice. Our POC repeatedly writes `0` and `9` to the shared memory. Thus, the only way our POC could trigger a return value of `-24` is if the POC modifies `shm[0]` between lines 6 and 12. Running this POC, we observe return values -5 and -24, confirming that QSEE supports zero-copy shared memory.

B. Different Output Logs in Kinibi

Listing 7 shows the double fetch, which we used to test if Kinibi supports zero-copy shared memory. The TA fetches the first four bytes from the shared memory twice. Our POC repeatedly writes `0x1009` and `0x1001` to the shared memory. On the testing device, the TA's logs are written to the kernel log. Thus, we can observe the output of all `uf_pf_log_msg` functions. Running our POC, we observe the following log output: `cmd: 0x1001`, `paytrigger_check_key`, which is only possible if `buf0[0]` was modified between lines 5 and 9, demonstrating that Kinibi supports zero-copy shared memory.

APPENDIX B
ARTIFACT APPENDIX

A. Description & Requirements

The artifact contains all the code and binaries to reproduce the paper's claims. The mitigation is stored in `optee_shm_patch`. `ae` contains the artifact evaluation wrapper scripts. The remaining folders all pertain to ScHMuzz.

1) How to access: The artifact is accessible here https://github.com/HexHive/taemu/tree/oversharing_ae or https://doi.org/10.5281/zenodo.22873417.

2) Hardware dependencies: x86-64 CPU with 8 cores and 16 GB of RAM running a recent Linux.

3) Software dependencies: Docker is required to run the artifact.

4) Benchmarks: None

B. Artifact Installation & Configuration

Clone the artifact and then run the following command to setup:

```
./ae/ae.sh setup
```

To run experiement E4 please send us your SSH public key over hotcrp. We will add it to the authorized keys for our VPS. The VPS will be used to access our testing devices for E4 over adb.

C. Experiment Workflow

All experiments except E4 are run from the `ae/ae.sh` scripts. The script can be used to run individual experiments or all relevant sequentially.

For E4, first connect with SSH to our VPS:

```
ssh ae@65.108.89.50
```

Then follow the instructions for the experiment.

Without access to the VPS, reproducing this experiment requires rooted devices with the vulnerable firmware.

D. Major Claims

- (C1): ScHMuzz automatically detects crashes only triggerable via double fetches in TAs (Table 1, Figure 4, Figure 5). This is proven by experiment E1
- (C2): We found six double fetch vulnerabilities (Table 2). This is proven by experiment E2.
- (C3): Rust TAs are also affected by overlapping fetches (Table 4). This is proven by experiment E3.
- (C4): Teegris, Mitee, QSEE, and Kinibi support zero-copy shared memory (Table 3). This is proven by experiment E4.

E. Evaluation

1) Experiment (E1): [Automatic Double Fetch Detection] [45 compute-hour + 10 human-minutes]: Runs the fuzzing campaign on the 30 harnessesd TAs including all three of ScHmuzz's stages.

[Preparation] `./ae/ae.sh setup`

[Execution] `./ae/ae.sh e1_automatic_df_detection`

[Results] The script prints the resulting data in the same format as Table 1 and generates the coverage graph figures (Figure 4,5). Compare the results from the table to Table 1 in the paper, the numbers should be similar. Compare the coverage graphs with Figure 4 and 5 from the paper, again they should be similar. By default this eval reduces fuzzing times to allow it to run on a commodity machine, the numbers will not be the same as in the paper.

2) Experiment (E2): [TOCTTOU Vulnerabilities] [<1 compute-hour + 5 human-minutes]: Runs the proof of concepts for the six discovered vulnerabilities against the emulator unilt the crashes reproduce.

[Preparation] None

[Execution] `./ae/ae.sh e2_vulns`

[Results] The script runs until it triggers the crashes for each vulnerability. Output should be that all the vulnerabilities reproduce.

3) Experiment (E3): [Rust TAs] [3 compute-hours + 5 human-minutes]: Runs ScHMuzz on the five Rust TAs.

[Preparation] None

[Execution] `./ae/ae.sh e3_rust`

[Results] After finishing the script prints the data in the format of Table 4. Compare the results, for 3 out of the 5 TAs ScHMuzz should detect overlapping fetches.

4) Experiment (E4): [On-device Shared Memory] [<1 compute-hour + 5 human-minutes]: runs pocs on-device that trigger double fetches to confirm zero-copy shared memory.

[Preparation] Obtain access to the VPS and login via SSH.

[Execution]
```
cd oversharing_ae
./ae/ae_ondevice.sh
```

[Results] The script prints the results in the same format as Table 3.

F. Customization

The `ae.sh` script can be customized using a number of environment varaiables.

`AE_SCALE=paper` uses the same times as the paper's evaluation.

`AE_JOBS` scales the number of parallel emulator containers.

`AE_EXPLORE_TIME` scales the time to fuzz each TA in the exploration phase.

`AE_FAF_TIME` scales the time spent on fetch anchored fuzzing per snapshot.

G. Notes

For E1, you may notice there are 27 harnesses even though we cover 30 TAs. The reason is that the 3 Kinibi TAs also exist for Beanpod. TÄMU does not support Kinibi TAs but we can still cover Kinibi by simply reusing the Beanpod TÄMU harnesses. (The static analysis of copying shared memory was done on the actual Kinibi Tas).

The reproduction for C4 includes only a subset of the testing devices used for the experiment in the paper. This is because some of these devices are currently being used for other projects. Additionally, both our mitee testing devices were unrooted during an update and can not be easily rerooted.