SyncEmu: Enabling Dynamic Analysis of Stateful Trusted Applications

Christian Lindenmeier
FAU Erlangen-Nürnberg

Matti Schulze
FAU Erlangen-Nürnberg

Jonas Röckl
FAU Erlangen-Nürnberg

Marcel Busch
EPFL

Abstract—**Modern mobile devices leverage ARM TrustZone to implement a Trusted Execution Environment (TEE). The security-critical services, called Trusted Applications (TAs), deployed in these TEEs form the backbone of those devices' security architectures. Unfortunately, TAs are not free from bugs and constitute the biggest attack surface of the TEE. A vulnerability in a TA can have devastating consequences, fundamentally compromising the whole system's security. Given the locked-down nature of COTS smartphones, the analysis of closed-source TAs remains challenging for independent security researchers.**

In this paper, we present SyncEmu to enable dynamic analysis of proprietary TAs found on COTS Android devices. To this end, we develop a framework to execute unmodified TEE firmware in an emulated environment (so-called *rehosting*). Using SyncEmu, we successfully *rehost* TrustedCore, a closed-source TEE implementation found on older Huawei devices. Furthermore, we propose and implement a novel technique called *CA-in-the-loop*, that allows SyncEmu to forward realistic requests of Client Applications (CAs) running on a physical smartphone to the rehosted TAs, pushing the boundaries of state-of-the-art in TEE rehosting.

1. Introduction

Modern ARM-based mobile devices leverage ARM TrustZone [2] to provide a Trusted Execution Environment. The device is split into two isolated execution environments: the *normal world* (NW) and the *secure world* (SW). While the NW runs a commodity OS (e.g., Android) and user-installed Apps, the SW hosts vendor-controlled firmware protected by TrustZone's hardware extensions. Only software deployed inside the SW has access to the full hardware of the device, so even if the NW is compromised, security guarantees by the TEE still hold [36]. As part of the TEE firmware, vendors ship Trusted Applications that implement security-critical services like device attestation [4], [30], digital rights management [17], or (biometric) authentication [18].

Because the TEE builds the backbone of the security architecture, a vulnerability in a TA can lead to full device compromise [9]. Consequently, thorough security testing of code running inside the TEE is vital to ensure the confidentiality and integrity of security-critical data. But as seen in the past, TEEs often have severe shortcomings in their security design and are prone to vulnerabilities [10]. This urges for possibilities to analyze proprietary TEE firmware more extensively. However, TEE firmware deployed on COTS mobile devices is shipped in a closed-source fashion, making static analysis cumbersome. Additionally, the locked-down nature of COTS smartphones (e.g., no debug interfaces are accessible) prevents on-device instrumentation of software running in the TEE, limiting powerful dynamic analysis techniques such as fuzzing [7].

Recent approaches propose a remedy for this challenge by trying to execute TEE firmware in a specially crafted emulated environment. This process is often called *rehosting* to differentiate it from full emulation because the goal is *not* to build perfect emulators for physical hardware as this is often not necessary for security analysis [13]. The vast amount of peripherals and the fact that they are often proprietary makes building perfect emulators cumbersome and sometimes impossible. Instead, in rehosting, we execute firmware in a *rehosting environment* that recreates the original device sufficiently good enough, finding a trade-off between precise emulation and practicability.

When trying to employ rehosting to COTS smartphones, Harrison et al. [20] identified that building a rehosting environment capable of running the entire software stack (e.g., Android *and* the TEE firmware) is infeasible. In their state-of-the-art approach, they propose *partial* rehosting, dividing the software stack and focusing on components targeted for security analysis (e.g., TEE firmware). In essence, they minimize the required rehosting effort by excluding the huge NW software stack, thus saving the task of implementing the majority of peripheral models found on smartphones (e.g., screen and camera). Instead, they mimic the existence of NW software by partially re-implementing its behavior in simple models.

However, simplifying the complex NW (i.e., Android, user Apps, and CAs) comes with huge costs in the fidelity of the rehosting environment. The effects are reported by Harrison et al. [20], who identify low code coverage during TA analysis. CAs (running in the NW) and TAs are highly intertwined, following specific stateful custom protocols [7]. For example, to send a request to a TA handling the encryption of data on a real device, a CA must interact with the user, Android, the file system, and the TA multiple times. Meanwhile, TAs build up an internal state, thus reacting differently to CA requests and therefore executing varying code blocks. State-of-the-art TEE rehosting approaches are currently limited to re-implemented and simplified CA software models, preventing realistic CA-to-TA interactions and therefore hindering holistic security analysis. On the contrary, rehosting CAs is also unreasonable, as they are, similar to their TA counterparts, often proprietary and require interactions with the rest of the device, thus we would have to rehost the entire NW software stack, which is not feasible [20].

Contributions. We propose *SyncEmu* to solve this conflict of goals by implementing a state synchronization mechanism between a rehosting environment and a rooted COTS smartphone. Our approach is motivated by the fact that trying to manually re-implement CA models is both cumbersome and inherently incomplete while entirely emulating CAs is also not practical. Instead, SyncEmu follows a *hardware-in-the-loop* principle, synchronizing a real device hosting fully functional CAs with rehosted TAs running in an emulated environment. In summary, we make the following contributions:

- We design and implement SyncEmu to enable the execution of real-world TAs in an emulated environment. While some industry research teams have implemented emulators for their devices [20], their emulators are not publicly available.
- We describe and implement the novel technique *CA-in-the-loop*, that allows the synchronized execution of CAs running on a rooted smartphone with rehosted TAs.
- We evaluate SyncEmu for an open-source and closed-source TEE implementation (Linaro's OP-TEE [29] and Huawei's TrustedCore found on Huawei P9 Lite smartphones) showcasing its practical applicability. We make SyncEmu publicly available.

2. Background

ARM TrustZone. The majority of mobile devices are powered by the ARMv8-A [1] architecture that supports up to four privilege levels, called exception levels (ELs), and a partition into two isolated execution environments. Figure 1 gives an overview. For simplicity, we exclude EL2 as software running at this level (e.g., hypervisor) is not in the scope of this work. On COTS smartphones, the NW (light gray) comprises a feature-rich Rich OS at N-EL1 (e.g., Android) and user-installed Apps together with vendor-provided CAs at N-EL0. In general, software running in the NW can be modified by the user after the device is rooted [24], [34], [36]. In contrast, the SW is used by the vendor to deploy integrity-protected TEE firmware. Security-critical services are implemented by TAs which are managed by a TrustZone Operating System (TZOS) running at S-EL1. CAs may request security services by issuing system calls (SVCs) to a TEE Driver running as part of the Rich OS that will execute a

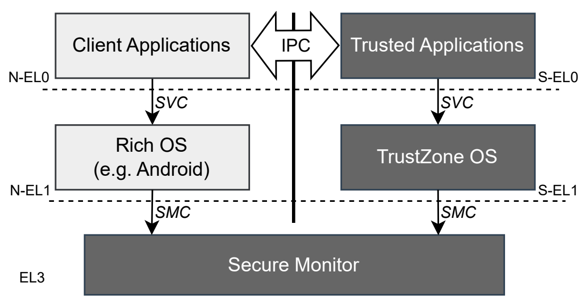

Figure 1. ARMv8-A architecture with common software components. Light gray components run in the normal world, while dark gray ones are part of the secure world. We indicate the vertical world isolation with a solid line, while the dashed lines represent the horizontal privilege isolation via exception levels. Client Applications and Trusted Applications communicate via inter-process communication (IPC).

secure monitor call (SMC) taking the execution to EL3. The secure monitor interprets arriving SMCs and handles world switches.

Lifecycle of Trusted Applications. CAs running in the NW request security services offered by TAs via a client-server principle. GlobalPlatform provides the TEE Client API [15] and TEE Internal Core API [16] which try to standardize the foundation of CA to TA communication. In essence, a CA implements the following protocol to request a service offered by a TA:

1) `TEEC_InitializeContext`: The fundamental object for a logical connection to the TEE is the `TEE Context`. As CAs and TAs are running in different execution worlds they use shared memory to exchange data, that is managed by the TEE Driver at N-EL1.
2) `TEEC_OpenSession`: To start an interaction with a TA, the CA prepares a session by sending a request to the TEE, identifying the TA via a UUID. During this process, initial data exchanges can take place (e.g., authentication) and the TA can run setup routines. If necessary, the TZOS loads the TA binary from the NW filesystem first.
3) `TEEC_InvokeCommand`: After the CA received a `sessionID` it can send commands to the TA instance. To this end, the CA indicates the requested TA function using a numeric identifier (called `commandID`) and parameters passed in shared memory. Each parameter can be of type value or memory reference. The content is not defined by the TEE Client API, allowing the implementation of custom protocols. Usually, a CA invokes multiple commands during one session, building up the internal TA state.
4) `TEEC_CloseSession`: The CA closes the session to the TA instance, giving the TEE the opportunity to clean up associated resources.
5) `TEEC_FinalizeContext`: Usually, the CA only ends the `TEE Context` when it has finished all communication with the TEE.

In summary, interactions between CAs and TAs have two layers of state involved. First, the sequence of calls required to establish a connection to a TA is dictated by the GlobalPlatform APIs. For example, a TA will only accept commands *after* the CA successfully opens a session. Second, on top of the generic TA interface, TAs implement custom services. Thus, a CA might be required to invoke a long sequence of `TEEC_InvokeCommand` calls to conduct an operation. For example, a TA offering secure storage requires a CA to open, read, write, and close a file. In general, a TA can be described as a state machine, where each request by the CA can be seen as input, thus their execution is intertwined.

3. SyncEmu

State-of-the-art rehosting approaches only run the TEE firmware, as emulating the complex NW software stack is not feasible (Section 1). Unfortunately, this limits the build-up of the TA state, as simple CA models can not create requests following the expected TA's API. Even worse, CAs are highly dependent on interactions with

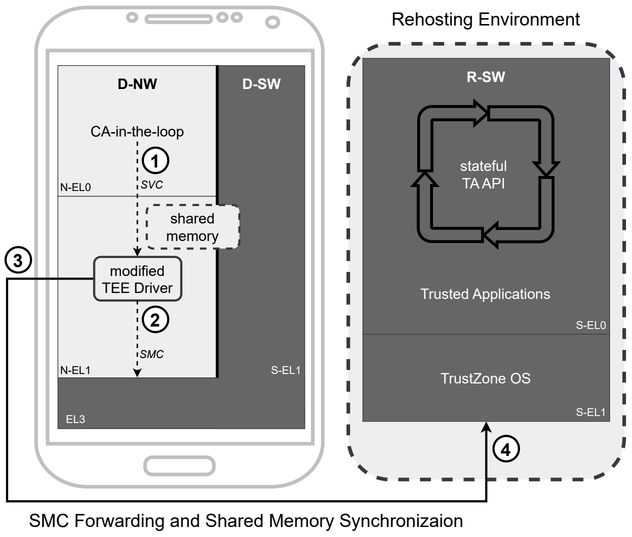

Figure 2. SyncEmu's approach. We modify the TEE driver of D-NW such that it can dump shared memory and instrument SMC execution. If a CA in D-NW ① requests a TA service, it fills shared memory by interacting with the TEE driver via system calls. When done, the TEE driver will try to perform an SMC ② which is suspended by SyncEmu. Meanwhile, the shared memory content and SMC are forwarded to SyncEmu's rehosting environment ③. In ④, SyncEmu injects the received CA requests in R-SW, thus feeding the same input into the rehosted TA like into the TA running in D-SW.

the rest of the NW, the device, and the user to generate meaningful inputs for TAs. Thus, rehosting approaches excluding a functional NW are inherently unfit to fully analyze stateful TAs.

SyncEmu overcomes this limitation by synchronizing the execution of a real physical device's NW (D-NW) with a rehosted SW (R-SW) running in an emulator. Figure 2 gives an overview of SyncEmu's approach. We attach a rooted mobile device to a host (e.g., a laptop) via USB and communicate using Android's debug interface `adb`. We note that `adb` can not be used to instrument SW software.

SyncEmu records and forwards communication by CAs running in D-NW to corresponding TA instances executing in R-SW adhering to the TA's expected protocol. Thus, the TA running in the emulator receives an *exact* copy of the request from the CA running on a fully functional device. We call this technique *CA-in-the-loop* inspired by *hardware-in-the-loop* [13], with the difference that we focus on forwarding software dependencies instead of peripheral interactions.

In the end, we aim for an equivalence between the execution of our rehosted TA and the one on the physical device in D-SW, enabling dynamic analysis of highly realistic TA behavior. In the following, we describe SyncEmu's two technical contributions: In Section 3.1, we present our TEE rehosting framework that enables the emulated execution of real-world TEE firmware. More specifically, we focus on running the TZOS and TA binaries, allowing for a TA-generic solution. We decided against emulating TAs in a standalone fashion, as emulating vendor-specific system calls offered by the TZOS is not straightforward [22]. Our framework enables prototyping callback functions to implement small hardware models and functionality of the bootloader and secure monitor to boot the TZOS and TAs.

In Section 3.2, we depict SyncEmu's novel *CA-in-the-loop* technique. In contrast to previous work, we focus on solving strong software dependencies of TAs on CAs during execution. As CA emulation is not feasible (we would have to rehost the entire NW), we coordinate the communication of an on-device CA with the corresponding TA running in R-SW. To this end, we modify the TEE driver of D-NW to include a forwarding mechanism that synchronizes requests of on-device CAs with our emulator at runtime exactly when an SMC is executed.

3.1. Rehosting Framework

To dynamically analyze TAs, we leverage a rehosting environment (Figure 2) of the targeted TEE implementation (R-SW). As part of SyncEmu, we develop a rehosting framework enabling the emulated execution of TEE firmware running on COTS mobile devices powered by ARMv8-A chips. To that end, we extend avatar[2] [26] that provides a configurable machine that is emulated using QEMU [5]. We use QEMU as our core emulator as it already comes with support for emulating ARMv8-A CPUs including the ARM TrustZone feature.

SyncEmu's rehosting framework enables the implementation of models to mimic software and hardware behavior in Python. An overview of SyncEmu's framework is depicted in Figure 3. In general, SyncEmu is designed to aid an analyst in manually working through an iterative refinement process as described by Fasano et al. [13] with a focus on TEE peculiarities. We illustrate a concrete practical walkthrough in Section 4.1. At its core, SyncEmu provides a basis for a rehosting environment that only needs to be refined for specific requirements of the targeted TEE implementation. Our rehosting environment is tailored to run COTS TZOS and TA binaries.

When creating a rehosting environment, the analyst starts by executing the TZOS and TAs, observing the logging output to identify errors, and trying to derive their root cause. Next, the analyst needs to refine the

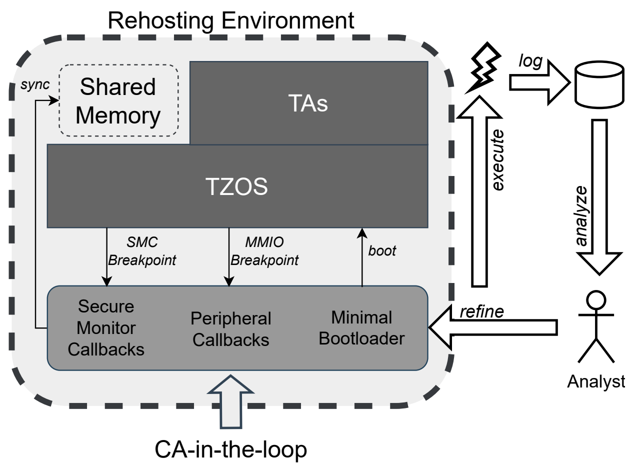

Figure 3. SyncEmu's rehosting framework. The goal is to run unmodified TZOS and TA binaries (dark gray components). An analyst goes through an iterative refinement process (big white arrows) to create the rehosting environment (box with bold dashed line). This comprises the three main components shown in medium gray. SyncEmu is designed to receive CA requests via the CA-in-the-loop interface.

environment, repeating this process until the TZOS boots. To this end, SyncEmu enables setting breakpoints and the implementation of callback functions to aid the analyst's process until all dependencies of the TZOS are handled. In the following, we present the dependencies of a TZOS during runtime and SyncEmu's corresponding solution.

Minimal Bootloader. We do not run the original bootloader binaries in our emulator, as this comes with high hardware emulation efforts [20]. Still, we need to re-implement a small bootloader that loads the TZOS binary in memory, configures the CPU, and prepares arguments to ensure a seamless boot process.

SyncEmu's solution: In our experiments, analyzing code that configures the MMU (e.g., usage of the page table base register `TTBR0_EL1`) turned out to be valuable oracles for identifying the physical memory layout. We leverage avatar^2's configurable machine to adapt the emulator's memory layout accordingly. SyncEmu comes with blueprints for emulating a minimal bootloader, that configures the CPU, places boot information structures in memory, and fills registers with argument values, that have been obtained by selective reverse engineering.

Peripheral Callbacks. During operation, the TZOS will try to interact with peripherals. On ARM-based devices, this happens via Memory-Mapped Input/Output (MMIO) accesses, i.e., the TZOS directly reads from and writes to memory regions mapped to peripheral registers. Since no models of (proprietary) peripherals are available in our emulator, SyncEmu needs a strategy to handle hardware accesses by the TZOS at runtime.

SyncEmu's solution: A core principle of rehosting is the observation that a rehosting environment is not required to give a perfect and complete emulation of every peripheral [20]. Thus, we emulate hardware behavior at the MMIO level on a per-read basis. Writes can be ignored as they do not directly influence the TZOS execution, and their effect can be handled by subsequent reads. SyncEmu enables hooking the execution at instructions that access MMIO regions and mimicking peripheral behavior by implementing callbacks that inject expected values *directly* into corresponding return registers. The analyst can identify these instructions by looking for polling loops (or error conditions) during iterative refinement.

Secure Monitor Callbacks. As the TZOS is deployed in S-EL1 it interacts with the secure monitor at EL3 by executing SMCs. Similar to the bootloader, we decided to not run the original secure monitor binary, because of the high hardware emulation effort required. Nevertheless, we require a strategy to handle SMCs.

SyncEmu's solution: Our rehosting framework provides secure monitor callbacks that are triggered when an exception level switch to EL3 occurs. We set a breakpoint at the exception vector table of the TZOS to catch all SMCs, pausing the TZOS in the process. SyncEmu then instruments the attached CA-in-the-loop Figure 2, waiting for arriving SMCs that are intercepted and forwarded on the attached device (see Section 3.2). When SyncEmu receives an SMC, it interprets the received data, injects the content into shared memory, and registers according to callbacks implemented by the analyst.

3.2. CA-in-the-loop

SyncEmu's goal is to enable the synchronization of the NW of a physical device with the rehosted TZOS in our rehosting environment. This includes all communication between an on-device CA and a rehosted TA, thus we keep the device running the CA *in the loop*. SyncEmu needs to satisfy the requirement that R-SW receives the equivalent input as D-SW at the exact same time. D-SW receives its input by communicating with D-NW, specifically the TEE driver. Thus, we develop an agent that enables intercepting and forwarding all data transfer between D-NW and D-SW to our emulator running R-SW.

We can break this task down into (1) synchronizing all of the shared memory and parameter registers and (2) at the exact time when an SMC is executed in D-NW. To this end, we deploy a modified version of a TEE driver running as part of D-NW on a rooted smartphone [24]. We provide details for the implementation in Section 4.2.

Shared Memory Synchronization. Shared memory is usually allocated dynamically by the TEE driver and is used to exchange data between CAs and TAs. While TEE driver implementations from vendors may differ, they loosely follow the TEE Client API (in other cases more code analysis may be necessary). TEE drivers based on GlobalPlatform usually manage `SMC_CMD` structs representing CA requests. As these are directly sent to the TEE, they will only hold physical addresses. A request may include data like the TA's UUID, `sessionID`, or `commandID`. Furthermore, it holds pointers to up to four `CMD_OP` structs that define parameter types and values. We implement a function `smc_write_out(SMC_CMD)` that extracts all payload associated with a specific CA request and sends the content to SyncEmu. The exact implementation needs to be adjusted for the targeted TEE driver, but this comes down to unrolling the `SMC_CMD` struct which is a manageable manual effort considering the fact that TEE drivers are often open-source since the Android system is mostly licensed under the GNU General Public License [12].

SMC Forwarding. Additionally, temporal dependencies have to be considered. SMCs are synchronous exceptions, thus D-NW will hand the control flow to the secure monitor and wait until it returns. Normally, D-SW would access shared memory provided by D-NW, thus potentially modifying its content. To ensure correctness, we need to forward all of the memory associated with the SMC right before trapping to D-SW. SyncEmu achieves this by modifications to the TEE driver of D-NW, that allow SMC executions to be blocking, keeping the device in D-NW, until memory has been forwarded to R-SW. In detail, we introduce a function `smc_block()` that waits on an internal variable (similar to a lock) that is controlled by SyncEmu. When SyncEmu received all forwarded data, it releases the lock, and the SMC on the physical device is executed normally. After identifying all occurrences of the SMC instruction in the TEE driver, the analyst has to add calls to `smc_block()` and `smc_write_out(SMC_CMD)` right before. Similarly, by adding `smc_write_out(SMC_CMD)` calls *after* the SMC instruction, returning values of D-SW can be obtained. We use this when evaluating SyncEmu's correctness in Section 4.2.

4. Evaluation

Evaluating rehosting approaches is challenging as both measuring the ground truth (i.e., original device's execution) and identifying metrics to compare the fidelity are not trivial. In our case, an idea would be to strictly compare the execution of D-SW and R-SW, however, access to D-SW (ground truth) is off limits (Section 2). Instead, we follow the evaluation design described by Fasano et al. [13] who propose to compare the input and output behavior of the original device and the rehosting environment. We compare the observable behavior of D-SW with the one of R-SW to have an indicator of the equivalence of their execution. CAs and TAs communicate via a defined API (Section 2), thus we compare the return values of these calls. We evaluate SyncEmu along the following research questions (RQs):

RQ1 Can we use SyncEmu to rehost COTS TZOS implementations?
RQ2 Does SyncEmu enable the build-up of TA internal state?
RQ3 Does SyncEmu's CA-in-the-loop technique preserve correctness?

To evaluate SyncEmu, we target two TZOS implementations, namely Linaro's open-source OP-TEE [25], [29] and Huawei's proprietary TrustedCore (TC) running on Huawei's P9 Lite smartphones (**RQ1**). We note that OP-TEE already runs in QEMU, thus we use it to evaluate our CA-in-the-loop technique since this setup excludes side effects because of missing peripheral emulations (**RQ3**). We targeted TC to complement recent TEE rehosting approaches [20] that did not target Huawei devices.

4.1. Rehosting COTS TZOS Implementations.

In this section, we demonstrate SyncEmu's rehosting framework (Section 3.1) to emulate Huawei's proprietary TZOS implementation TC, addressing **RQ1**.

Rehosting Process. We take the following steps in the iterative refinement process to implement a rehosting environment that enables booting TC.

1) Obtaining the TC Firmware Image. We extract the TC binary from a storage partition on a rooted Huawei P9 Lite. Alternatively, TEE firmware can often also be obtained from update images.

2) Initial Static Analysis. We use Ghidra [28] for disassembling and reverse engineering parts of the TC binary. For setting up an initial emulator, we identify the corresponding instruction set architecture to be ARMv7. However, Huawei's P9 Lite is powered by an ARMv8-A CPU, thus we use the CPU model `cortex-a57` running TC in backward-compatible mode. We found TC's initial physical loading address after analyzing code that sets up page tables for the MMU.

3) Running TC until Failure. We configure SyncEmu's minimal bootloader to configure the emulator accordingly and load TC in memory. After our bootloader, TC is executed starting at its first instruction. SyncEmu enables logging for exceptions, basic block translations, and aborts, thus by observing the output we can identify error behavior during emulation. In the beginning, we expect errors after a very short emulation period as our initial rehosting environment does not handle any peripheral interactions.

4) Interpreting and Handling of Errors. Next, we analyze the output logs to identify the failing instruction and derive the root cause. We parse the address of the first instruction leading to a fault, identify the corresponding code blocks in the binary, and focus our efforts on backtracking the control flow. During our experiments, looking for aborts and MMIO polling loops turned out to be valuable starting points.

5) Refinement of Rehosting Environment. We derive necessary improvements to our rehosting environment and extend SyncEmu's callback implementations. Depending on the observed abort, we identify commonly required modifications. A data abort usually indicates that the loading address is wrong (e.g., the faulting instruction enabled the MMU) or bootloader parameters are missing. A polling loop indicates that MMIO read instructions failed due to missing peripheral emulation. A prefetch abort indicates that the CPU tries to execute memory not holding valid instructions (e.g., missing hook for secure monitor callbacks).

We repeat steps three to five until TC boots in the refined rehosting environment.

Running TrustedCore. We now report on the complexity of our implemented rehosting environment (i.e., callbacks and bootloader) for running Huawei's TC. This includes concrete solutions for secure monitor and peripheral callbacks as well as a minimal bootloader Section 3.1.

We implement a minimal bootloader stub consisting of only 19 assembler instructions that are executed at EL3 before jumping to the TC binary. Our emulated bootloader sets appropriate values in system registers (e.g., `SCR_EL3`, `ELR_EL3`, `SPSR_EL3`, and `SCTLR_EL1`) to configure S-EL1 for the execution of TC. Additionally, we fill argument registers (e.g., `x0-x4`) necessary for TC's boot process. We were not required to pass more complex bootloader structures. We believe this to be the case because our targeted TC binary is compiled to only run on one specific smartphone, thus instead of using parameters, required values are hardcoded.

When our bootloader is finished, it executes an `eret` instruction, jumping to TC's entry address at `0x36208000`. When running, TC tries to access peripherals via MMIO. Using SyncEmu's rehosting framework, we set hooks at 18 locations to return peripheral values keeping TC from crashing. Crucially, we were not required to assign MMIO accesses to specific peripherals nor understand their functionality in-depth, minimizing reverse engineering efforts. After observing the first SMC, we hook the control flow at the exception vector table and implement a secure monitor callback function. Our callback interprets the function identifier set by TC in `x0` (i.e., the value indicating successful boot), and reads out TC's expected return address from `x1` required to resume execution. We pause the emulation and wait for inputs from the attached device.

4.2. CA-in-the-loop

For SyncEmu we need to deploy agent software running on the attached smartphone, in order to forward re-

TABLE 1. TOTAL NUMBER OF CA REQUESTS PER TA API FUNCTION AND CORRESPONDING EQUIVALENT RETURN VALUES IN PARENTHESES.

API function	OP-TEE's aesTA	TC's keymasterTA
TEEC_InitializeContext	79 (79)	56 (56)
TEEC_OpenSession	1 (1)	56 (56)
TEEC_InvokeCommand	8 (8)	56 (0*)
TEEC_CloseSession	1 (1)	56 (56)

quests sent by a CA. We modify the TEE driver implementation of the attached device to include this functionality.

Modifications to TrustedCore's TEE Driver. For TC, we used the corresponding open-source TEE driver for Huawei P9 Lite devices [21]. The communication takes place in a ring buffer inside shared memory, whose location is transmitted to TC by setting a register parameter in one of the first SMCs. The ring buffer is divided into input and output queues, holding messages sent to and from TC respectively. The current message is referenced by a 4-byte index at the start of the memory region. Each message consists of a `TC_NS_SMC_CMD` struct, indicating the requested TA UUID, and a pointer referencing a physical address to a `TC_NS_Operation` struct. This struct holds up to four pointers to `TC__NS_Parameter` objects for the actual payload (i.e., data sent by a CA).

We deploy SyncEmu's kernel module on the device, that provides the function `smc_block()` to instrument the execution of SMCs on the device. The module exposes the file `/proc/smc_forwarder` to SyncEmu enabling the transfer of commands and data. Next, we implement the function `smc_write_out(TC_NS_SMC_CMD *cmd)` that is capable of parsing the CA request struct and extracting all associated shared memory. We identified that TC's TEE driver issues SMCs in the file `drivers/hisi/tzdriver/smc.c` and add calls to our implemented functions, thus enabling SyncEmu to instrument and intercept communication to TC.

In summary, our kernel module consists of around 300 lines of C code and we needed to add three calls to SyncEmu's functions in the TEE driver.

Experiments. We conduct experiments for both OP-TEE and TC to evaluate **RQ2** and **RQ3**. For OP-TEE, we run a setup with two identical emulators (i.e., we replace the physical device) and use SyncEmu to synchronize the execution of the *aesTA*. This experiment is especially interesting to answer **RQ3** since it excludes side effects due to missing peripheral emulations. For TC, we use a Huawei P9 Lite, deploy our kernel module, and use an Android App that enables us to directly interact with the on-device CA, requesting functionality from the *keymasterTA*. In each setup, we compare the return values of API calls to the TA running in D-SW with the return values of the rehosted TA in R-SW (Section 2). We present our results in Table 1.

SyncEmu intercepts and forwards a total of 89 and 224 CA requests for our experiment with OP-TEE and TC, respectively. In both scenarios, SyncEmu's CA-in-the-loop technique leads to a successful build-up of internal TA state (e.g., TAs are loaded and sessions established), thus answering **RQ2**. For OP-TEE, we observe identical return values over all requests, while for TC in 75% of cases. We identified that the reason for different return values (marked with *) can be attributed to the lack of peripheral emulation. TC's keymasterTA leverages a proprietary crypto cell [3] for cryptographic operations. Properly emulating complex peripherals is not in the scope of SyncEmu and can be a task for future work. We confirm that SyncEmu's CA-in-the-loop technique nevertheless forwards the correct requests by looking at the secure UART output of our rehosting environment (Listing 1) and manual debugging. Another indicator for SyncEmu's correctness (**RQ3**) is the fact that in a perfectly emulated environment (i.e., in our case with OP-TEE), we observe entirely equivalent behavior.

```
[2task_keymaster]8/24/2021 3:27:19.176 [error]
     TEE_GenerateRandom:
    CRYS_RND_GenerateVector failed
[2task_keymaster]8/24/2021 3:27:19.176 [error]
     TEE_GenerateRandom:
    CRYS_RND_GenerateVector failed
[2task_keymaster]8/24/2021 3:27:19.176 [error]
     keyblob_crypto: derive key from kdf
    failed, ret=0x00f0020f
[2task_keymaster]8/24/2021 3:27:19.176 [error]
     generate_symmetric_keymaterial: encrypt
    privatekey failed, ret = ffff0000
[2task_keymaster]8/24/2021 3:27:19.176 [error]
     GenerateSymmetricKey:
    generate_symmetric_keyblob failed, ret is
    ffff0000
[2task_keymaster]8/24/2021 3:27:19.176 [error]
     km_generate_key: GenerateSymmetricKey
    failed
[2task_keymaster]8/24/2021 3:27:19.176 [error]
     TA_InvokeCommandEntryPoint: keymaster
    invoke failed, ret=ffff0000
```

Listing 1. Emulator UART output of rehosted keymasterTA instance of TC. After invoking the generation of a symmetric key, errors occur. The TA tries to communicate with the crypto cell which is not implemented in the rehosting environment leading to expected failure.

5. Discussion

5.1. Limitations

In the first place, the main requirement for the usage of SyncEmu is the availability of a physical smartphone. Because of the locked-down nature of COTS smartphones, SyncEmu still comes with some limitations depending on the specific device. First, for the deployment of the custom TEE driver, we need the ability to flash custom kernel software, thus the bootloader of the device needs to be unlocked. While most vendors allow unlocking the bootloader for supporting custom OSes (e.g., GrapheneOS), the TEE will typically notice this during boot. For example, when unlocking the bootloader on Samsung devices an eFuse is irretrievably burned, marking the device as untrusted [31], thus potentially limiting communication with TAs.

Second, we require access to the TEE firmware (e.g., TZOS and TA binaries) for rehosting. While for our target device, we extracted the TEE firmware from the rooted device, vendors might encrypt (parts of) their firmware running in the TEE. Consequently, the TEE firmware is shipped and stored in an encrypted fashion, while the decryption only happens in secure volatile memory by an early bootloader stage, making it inaccessible.

Third, services like attestation or cryptography that may leverage unique secrets stored in hardware elements (e.g., root of trust) can hinder the implementation of high-fidelity rehosting environments.

5.2. Future Work

The implementation of SyncEmu's rehosting environment takes manual effort by an expert. While it took us months for the initial implementation of SyncEmu's rehosting framework, subsequent extensions with TEE implementations only required some weeks. Still, when considering that there currently is no out-of-the-box solution for TEE rehosting, we believe this to be a justifiable manual effort. We leave the extension of SyncEmu with additional TEE implementations for future work.

Furthermore, our evaluation results showed that missing peripheral emulation prevents SyncEmu from completely emulating TA services that have heavy hardware dependencies. Unfortunately, peripherals found on COTS smartphones are typically proprietary. Without access to documentation, manually simulating fully functional peripheral behavior is expected to be not feasible. Instead, we envision extending SyncEmu's rehosting framework with strategies that approximate peripheral behavior at the MMIO level, as shown by prior work [32].

We see SyncEmu as a tool with multiple use cases regarding security analyses. For example, one aspect could be the application of fuzzing to the rehosted TEE firmware. Fuzzing could allow an in-depth evaluation of SyncEmu's CA-in-the-loop technique in regard to improving the reached code coverage and a holistic comparison with state-of-the-art fuzzing approaches. We see multiple options for integrating fuzzing to SyncEmu's current design. Challenges are scalability and performance since SyncEmu's CA-in-the-loop technique depends on a physical smartphone. For example, we expect rebooting the device after a crash to be slow [7]. However, since SyncEmu runs the TEE firmware in an emulator, snapshot-based fuzzing might be applicable. Additionally, finding a good interface for the fuzzer and enabling effective mutation may be non-trivial. For example, one could feed fuzzing inputs into CAs running on the smartphone to trigger TA services in the first place while mutating the forwarded CA requests before injecting them into the rehosting environment.

6. Related Work

Trying to dynamically analyze TEE firmware on physical devices comes with inherent limitations [7], [27] due to inaccessible debug interfaces and lack of introspection. While there exists work that identifies security issues in TEE firmware using static analysis, these either require heavy reverse engineering [9], [33] or focus on specific security issues [8]. Currently, the two dominant approaches for rehosting are *pure emulation* and *hardware-in-the-loop* [13].

In a pure emulation approach [11], no physical hardware is involved. Peripheral interactions are solved by implementing models mimicking their behavior [14], [19]. The most similar to our work is PartEmu [20] that enables the execution of proprietary TEE firmware in an emulator. However, their evaluation shows limitations in building up TA state during fuzzing. SyncEmu tackles this by its novel CA-in-the-loop technique, thus feeding highly realistic inputs to rehosted TAs. Unfortunately, the authors never published their source code, hindering a direct comparison. In contrast, we make our artifacts publicly available, fostering future TEE research. Other research tries to implement tailored emulators targeting the execution of stand-alone TAs [6], [22]. However, these are TA-specific and require emulating the custom system call interface of the TZOS, making the process tedious. In contrast, SyncEmu runs the TZOS in the emulator, making a TA-generic solution possible. Furthermore, there exists research that targets other software layers of the TEE firmware such as the secure monitor [23], while SyncEmu is tailored for TAs.

Hardware-in-the-loop approaches forward peripheral accesses to a physical device [26], [35]. This ensures higher fidelity but may require partial debugging access to the original device. SyncEmu uses the forwarding methodology for its CA-in-the-loop technique, but focuses on solving software dependencies instead.

7. Conclusion

While most improvements in the field of rehosting focus on hardware modeling, rehosting TEE implementations comes with the additional challenge of strong software dependencies because of the interplay of multiple components. With SyncEmu, we propose and develop a potential solution to tackle this challenge and enable holistic analysis techniques for COTS TAs. We successfully rehost and demonstrate our novel *CA-in-the-loop* technique targeting a real-world TZOS implementation, showcasing the practical applicability of our approach. We believe that SyncEmu is a valuable tool to aid TEE research and empower analysts to apply techniques such as fuzzing to proprietary TEE firmware, allowing for improved vulnerability discovery.

Data Availability

SyncEmu is openly accessible at https://github.com/syncemu/syncemu.

Acknowledgements

We thank our anonymous reviewers for their valuable feedback and comments. This research was partly supported by the German Federal Ministry of Education and Research (BMBF) as part of the CELTIC-NEXT project AI-NET-ANTILLAS ("Automated Network Telecom Infrastructure with inteLLigent Autonomous Systems", Förderkennzeichen "16KIS1314").

References

[1] ARM. Arm Architecture Reference Manual Armv8. https://documentation-service.arm.com/static/60119835773bb020e3de6fee?token=.

[2] ARM. ARM TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone.

[3] ARM. CryptoCell-300 Family, 2024. https://www.arm.com/products/silicon-ip-security/crypto-cell-300.

[4] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 90–102. ACM, 2014.

[5] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005.

[6] 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.

[7] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. TEEzz: Fuzzing Trusted Applications on COTS Android Devices. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1204–1219. IEEE, 2023.

[8] 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 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[9] Marcel Busch, Johannes Westphal, and Tilo Müller. Unearthing the TrustedCore: A Critical Review on Huawei's Trusted Execution Environment. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020.

[10] 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 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1416–1432. IEEE, 2020.

[11] 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 2020, August 12-14, 2020*, pages 1201–1218. USENIX Association, 2020.

[12] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2123–2138. ACM, 2017.

[13] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, pages 687–701. ACM, 2021.

[14] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1237–1254. USENIX Association, 2020.

[15] GlobalPlatform. TEE Client API Specification v1.0. https://globalplatform.org/specs-library/tee-client-api-specification/.

[16] GlobalPlatform. TEE Internal Core API Specification v1.3. https://globalplatform.org/specs-library/tee-internal-core-api-specification/.

[17] Google. DRM, 2024. https://source.android.com/devices/drm.

[18] Google. Show a Biometric Authentication Dialog, 2024. https://developer.android.com/training/sign-in/biometric-auth.

[19] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pages 135–150. USENIX Association, 2019.

[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 2020, August 12-14, 2020*, pages 789–806. USENIX Association, 2020.

[21] Huawei. Huawei P9 Lite open-source Linux kernel. https://github.com/twicejr/HUAWEI-P9-Lite_OpenSource.

[22] D. Komaromy. Unbox Your Phones, 2018. https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c.

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

[24] Magisk. Rooting with Magisk Manager Android App. https://magiskmanager.com/.

[25] Brian McGillion, Tanel Dettenborn, Thomas Nyman, and N. Asokan. Open-TEE - An Open Virtual Trusted Execution Environment. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 400–407. IEEE, 2015.

[26] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, volume 18, pages 1–11, 2018.

[27] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[28] NSA Research. Ghidra, 2024. https://ghidra-sre.org/.

[29] OP-TEE. Open Portable Trusted Execution Environment - OP-TEE. https://www.op-tee.org/.

[30] Samsung. Knox Platform for Enterprise White Paper. https://docs.samsungknox.com/admin/whitepaper/kpe/samsung-knox.htm. Last accessed: 20.03.2024.

[31] Samsung. Samsung Knox Documentation, Root of Trust. https://docs.samsungknox.com/admin/fundamentals/whitepaper/samsung-knox-for-android/core-platform-security/root-of-trust/.

[32] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 1239–1256. USENIX Association, 2022.

[33] D. Shen. Attacking your "Trusted Core" Exploiting TrustZone on Android. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf.

[34] TWRP. TeamWin Recovery Project Custom Recovery for Android. https://twrp.me/.

[35] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[36] Hang Zhang, Dongdong She, and Zhiyun Qian. Android Root and its Providers: A Double-Edged Sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1093–1104. ACM, 2015.