EL3XIR: Fuzzing COTS Secure Monitors

Christian Lindenmeier
FAU Erlangen-Nürnberg

Mathias Payer
EPFL

Marcel Busch
EPFL

Abstract

ARM TrustZone forms the security backbone of mobile devices. TrustZone-based Trusted Execution Environments (TEEs) facilitate security-sensitive tasks like user authentication, disk encryption, and digital rights management (DRM). As such, bugs in the TEE software stack may compromise the entire system's integrity.

EL3XIR introduces a framework to effectively rehost and fuzz the secure monitor firmware layer of proprietary TrustZone-based TEEs. While other approaches have focused on naively rehosting or fuzzing Trusted Applications (EL0) or the TEE OS (EL1), EL3XIR targets the highly-privileged but unexplored secure monitor (EL3) and its unique challenges. Secure monitors expose complex functionality dependent on multiple peripherals through diverse secure monitor calls.

In our evaluation, we demonstrate that state-of-the-art fuzzing approaches are insufficient to effectively fuzz COTS secure monitors. While naive fuzzing appears to achieve reasonable coverage it fails to overcome coverage walls due to missing peripheral emulation and is limited in the capability to trigger bugs due to the large input space and low quality of inputs. We followed responsible disclosure procedures and reported a total of 34 bugs, out of which 17 were classified as security critical. Affected vendors confirmed 14 of these bugs, and as a result, EL3XIR was assigned six CVEs.

1 Introduction

Mobile devices provide critical sensitive services, including authentication [55], mobile payment [49], and DRM [28]. These services depend on hardware-based Trusted Execution Environments (TEE) (such as ARM TrustZone [2]) to protect sensitive data. The TEE enforces the integrity and confidentiality of its software components through hardware features. By shifting security-critical operations from the Rich Execution Environment (REE) into the TEE, users are protected from various application- and kernel-level exploits [3]. The smaller code base of TEEs aspires to have a small attack surface and, consequently, protect devices from full compromise.

However, fueled by market needs and recent research [31, 36, 46], the amount and size of software running inside the TEE continuously increases. Vendors are extending the TEE with more functionality such as device integrity monitoring [10], remote attestation, or secure network communication [51] – all built upon the premise of trusted software. Consequently, the trusted computing base (i.e., the code running as part of the TEE) is growing, leading to a increased risk of bugs. Moreover, regardless of this fact, TEE-based vulnerabilities significantly impact device security, as successful attacks put the entire platform's security at risk [17, 18].

Most mobile devices such as smartphones or tablets are based on the ARMv8-A architecture [1], and as such, ARM TrustZone [2] provides the dominant foundation for TEE implementations on billions of devices. TrustZone logically separates the device into two isolated execution environments; the non-trusted *normal world* and the trusted *secure world* [48]. The normal world on a typical Android device hosts conventional user space applications and a Linux kernel. The secure world exclusively runs vendor-authenticated code consisting of Trusted Applications (TAs), a Trusted Operating System (TEE OS), and the secure monitor. While the security of TAs and TEE OSes have been previously studied [17, 18, 52, 53], the security of the secure monitor remains unexplored. It has been primarily seen as a means to request services and exchange data between the worlds [5]. Indeed, all communication to the TEE and vice versa must pass through the secure monitor, which is the highest privileged component (EL3) on the platform [1].

Surprisingly, the mediation between the normal and secure world is not the only service the secure monitor offers. Additional tasks include the interaction with silicon provider-specific hardware [9], management of system power levels [8], and firmware updates [7]. Furthermore, vendors tailor the secure monitor firmware to the specific needs of their devices by adding drivers for secure peripherals (e.g., eFuses and cryptographic elements), custom TEE OS interfaces, and proprietary bootloader features. Code running in the normal world can request these services by issuing a Secure Monitor Call (SMC)

which switches to the secure monitor context. When the normal world requests such a service, it is directly handled by the secure monitor and not forwarded to the TEE OS. Unsurprisingly, this complexity results in severe bugs potentially leading to full-system compromise [4, 43].

The secure monitor is part of the secure world (Figure 1); thus, vendors do not permit introspection during runtime. This is enforced by memory controllers (e.g., the TrustZone Address Space Controller) that prevent normal world components from accessing code or data of the secure world. Consequently, modern dynamic analyses, such as coverage-guided fuzzing, are infeasible on production devices.

Rehosting offers a remedy for the limitations of on-device approaches. In the context of small embedded systems, we have recently seen significant advancements in *full-system rehosting* [21, 24, 25, 33, 34, 38, 50]. However, these approaches focus on comparatively tiny firmware samples for simple CPUs (e.g., Cortex-M). Cortex-A software stacks (e.g., all components running on the application processor of a mobile device, including the bootloader, secure monitor, TEE OS, TAs, REE OS, and REE applications) are orders of magnitudes more complex and have significantly more hardware dependencies, rendering full-system rehosting approaches infeasible. A trade-off to full-system emulation is *partial rehosting*. The essential idea is to only rehost the targeted relevant logical component and trade hardware dependencies for software dependencies. For example, Harrison et al. [32] focus on rehosting TEE OSes and TAs. Hence, they do not include irrelevant components in their rehosting efforts, excluding the secure monitor or the REE OS. By mimicking hardware behavior with software emulation, partial rehosting approaches avoid requiring complete hardware models.

We discover two limitations of prior work, making them unfit for effectively fuzzing hardware-dependent proprietary firmware components (e.g., secure monitors) with loosely defined interfaces. First, the generic and vast interface of secure monitors is challenging for general-purpose fuzzers to explore due to its large input space. Second, secure monitors in particular depend on vendor-specific hardware interactions. Public emulators lack support for these peripherals and corresponding data sheets are unavailable, thus posing a roadblock for dynamic analysis. Manually implementing simplified hardware models reduces the fidelity of rehosting environments and is not a scalable solution.

EL3XIR solves these two key challenges. First, we perform static value-flow analysis on the REE OS (e.g., Linux kernel) to collect instances of SMC interface usage. Further, we achieve *interface awareness* by probing the existing handlers of a given secure monitor. Then, we combine this knowledge with a set of domain-specific mutations in a fuzzing harness capable of generating high-quality inputs. Second, EL3XIR overcomes coverage roadblocks originating from hardware interactions using *reflected peripheral modeling*. This technique repurposes fuzzing inputs to probe Memory-Mapped I/O (MMIO) behavior and leverages coverage feedback as an oracle for appropriate peripheral behavior.

In our evaluation, we targeted seven different secure monitor binaries scattered across six vendors. Our dataset includes three proprietary implementations deployed on popular mobile devices from Samsung and Huawei. Additionally, we leverage EL3XIR to fuzz four open-source secure monitor implementations running on edge devices from Intel, NXP, Xilinx, and Nvidia showing EL3XIR's scalability. EL3XIR finds crashes in all of the targets, with a total of 34 unique bugs. After triaging and responsible disclosure, a total of 17 are security-critical with six CVEs assigned. Our evaluation shows that *interface awareness* and *reflected peripheral modeling* lead to higher code coverage than state-of-the-art fuzzing approaches. Furthermore, EL3XIR triggered 15 more crashes compared to approaches of prior work.

In summary, our main contributions are:

- The design and implementation of an end-to-end rehosting-based fuzzing framework for proprietary secure monitor implementations.
- Two essential techniques, static analysis-based *interface awareness* and *reflected peripheral modeling*, enabling target-specific fuzzing harness synthesis.
- A comprehensive evaluation of EL3XIR against existing state-of-the-art TEE fuzzing approaches.

2 Background

This section provides the necessary background on the ARMv8-A architecture, the role of the secure monitor, and the threat model of ARM TrustZone.

ARMv8-A Architecture. The 64-bit ARMv8-A architecture [1] dominates the System-on-Chip (SoC) market for embedded devices like smartphones. This architecture supports up to four privilege levels called exception levels (ELs). Further, ARMv8-A features ARM TrustZone [2, 48], a hardware extension capable of partitioning the SoC into two isolated execution environments. Figure 1 shows the separation of the execution into the non-trusted *normal world* which hosts a REE OS at EL1 (e.g., a Linux kernel) and user space applications at EL0 (e.g., an Android App) and the *secure world* which is comprised of vendor-specific software like the TEE OS at EL1 and Trusted Applications at EL0. Software running in the secure world is integrity protected by a secure boot mechanism executing only vendor-signed code. EL3 is the highest privilege level and any context switch between the worlds has to go through this exception level using an SMC.

Secure Monitor. The secure monitor is the runtime software located at EL3, and it has unrestricted access to all device resources. The bootloader authenticates the secure monitor's image during secure boot. As a standard for secure monitor interactions, ARM defined the Secure Monitor Call Calling

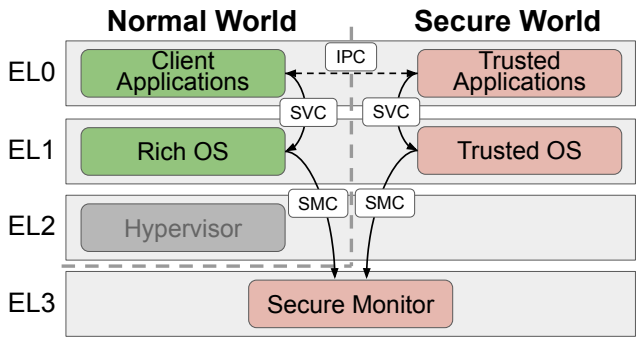

Figure 1: ARMv8-A platforms support up to four privilege levels (EL3 to EL0). On COTS devices, vendors often grant users access to the normal world (green components) but the secure world (red components) is off limits. The dashed lines indicate the security boundary between the two execution environments. The IPC interface provides a conceptual call between the worlds that needs to pass through all layers of the software stack.

Convention [5] (SMCCC). This convention defines the parameter registers (`x0-x17`) and the result registers (`x0-x3`) to be used when interacting with secure monitor services. Besides its role as a relay to the TEE OS, the secure monitor includes services like access to the power state coordination interface or silicon provider service calls intended for chip-specific drivers implemented by each vendor on a per-platform basis. This results in multiple scattered runtime services, each requested by another function identifier (via register `x0`) and a different interface.

ARM provides the ARM Trusted Firmware [6] (ATF) as an open-source reference implementation. Most device vendors use the ATF as a foundation for their TEE architecture and modify or extend the implementation according to the specific needs of their platform. The ATF is designed to be extendable and vendors will usually register runtime services in the silicon provider service call categories to enable the trusted interaction with proprietary secure peripherals. As a result, the ATF includes device drivers directly interacting with hardware components via MMIO. Whereas the ATF's core implementation is open-source, vendors usually keep their extensions private.

TrustZone's Threat Model. ARM TrustZone is designed to provide a secure execution environment for sensitive software and data even if the REE is compromised. This includes attackers taking complete control over the REE OS and sending arbitrary payload to the secure monitor when executing an SMC. Therefore, in our attack scenario, we assume root-level access to our target device to execute arbitrary code in the normal world. Yet, a kernel-level attacker should not be able to violate the integrity and confidentiality of software running inside the TEE.

3 Challenges

Effective fuzzing requires dynamic analysis of the execution to extract coverage information or crash details. We discuss challenges regarding dynamic analysis capabilities in the context of *on-device approaches* and *rehosting approaches*. We conclude that for COTS secure monitor implementations the balance shifts in the direction of favoring rehosting solutions as the locked-down nature of production devices and their TEE firmware increases the burden for on-device approaches.

On-Device Approach. *Feedback-guided* fuzzing is incompatible with COTS secure monitors on production devices. Missing or inaccessible debug interfaces prohibit instrumentation of firmware for coverage collection. Additionally, the secure monitor belongs to the secure world (Figure 1), thus vendors prohibit static modifications through the secure boot mechanism. Dynamic modification is prohibited by memory controller restrictions (e.g., the TrustZone Address Space Controller) which prevent normal world components from accessing code or data of the secure world during runtime.

Consequently, dynamic analysis approaches requiring introspection capabilities (e.g., coverage-guided fuzzing) are infeasible on production devices. Although there have been some successful black-box fuzzers in terms of discovered bugs, their effectiveness and usability are limited [16, 40]. The lack of extensive feedback makes it difficult to mutate potent test cases and the non-determinism of real-world hardware hinders the reproduction and triaging of bugs.

Rehosting Approach. Different from on-device approaches, the target may be run in an emulator. This has multiple benefits: (1) the extended introspection allows for coverage collection enabling *feedback-guided* fuzzing, (2) the absence of non-determinism improves reproduction of test cases and eases bug triaging, and (3) the performance and scalability can be improved by snapshot-based fuzzing.

Unfortunately, public emulators do not support peripherals for SoCs found on production devices running COTS secure monitor implementations. Furthermore, data sheets for these proprietary SoCs are unavailable which renders the implementation of accurate emulators infeasible [24]. While some industry research teams have implemented precise emulators for their devices [32], their emulator is not publicly available and building such emulators requires considerable manual engineering efforts. Nevertheless, the superiority of emulator-based fuzzing approaches urges to address the challenges emerging when trying to design a rehosting-based fuzzing solution for COTS secure monitor implementations.

C1: Interdependencies of Logical Components

Because of the high complexity and size of software stacks deployed on modern devices, a full rehosting approach would require the emulation of multiple intertwined software components. For example, when targeting the secure monitor im-

plementation running on an Android smartphone, this would require building a full-system emulator capable of running a customized version of Android (REE OS), the proprietary TEE OS, and the bootloader because all of those components interact with the secure monitor at some point (Figure 1). The size and complexity of the software stack on modern mobile devices makes this task practically infeasible.

Consequently, to enable rehosting-based fuzzing of COTS secure monitor implementations, we need a partial-rehosting approach that can run the target binary in a standalone way while maintaining the minimal necessary functionality of dependent relevant logical components. Harrison et al. [32] introduced the idea of a partial-rehosting approach. The key concept is to either reuse (i.e., executing the binary) or emulate (i.e., reimplementing the functionality) dependent software components. Partial-rehosting approaches exist for the TEE OS and some standalone TAs [40], while we focus on the secure monitor binary for which there is currently no solution. Ultimately, this challenging rehosting problem entails handling the secure monitor's dependencies to the REE OS, the TEE OS, and the bootloader.

C2: Infeasibility of Manual MMIO Modeling

In addition to solving dependencies on logical components, a rehosting environment capable of running COTS secure monitor implementations must emulate the original production device's hardware sufficiently precise [24]. Thus, mimicking interactions with hardware (e.g., via reading and writing MMIO regions) that would otherwise block the fuzzer from making progress as, there is no replying peripheral. We provide an indicative example for coverage walls caused by missing MMIO behavior in Appendix A.

Recent work [32] tried to manually reverse-engineer and emulate peripheral models, resulting in multiple limitations: (1) the resulting hardware models will be specific to a single SoC, limiting scalability, (2) proprietary peripherals may not be modeled extensively, (3) manual emulation requires extensive work by an expert for each SoC. In contrast, our work focuses on finding a scalable solution for fuzzing secure monitor implementations that heavily rely on hardware interactions.

C3: Complex and Diverse Input Formats

Finding meaningful inputs is a challenging task for fuzzers. This is because of non-standardized and complex input formats found at unexplored interfaces of low-level firmware like the secure monitor. The SMC interface is defined in the ARM SMCCC [5] (Section 2), but the discovery of valid inputs and, thus, the exploration of the secure monitor remains an open challenge.

Test inputs need to comply to custom protocol formats as expected by runtime services. Non-compliant inputs will fail in the parsing logic, generating no new coverage. Listing 1 in Appendix A demonstrates this challenge at the beginning of the function in the form of validation logic checking for alignment and memory pointer validation (`addr_in_nw_range()`). While this is a relatively simple example, manufacturers may include complex validation logic depending on the offered service and the necessary protocols to follow for the interaction with underlying peripherals. Naive fuzzers will struggle to create valid inputs, resulting in coverage walls and bad performance.

Another reason is the broad nature of the SMCCC used to communicate with runtime services. On modern ARMv8-A systems, up to 18 input registers are passed to the secure monitor. All lower 32 bits of `x0` are used as a function identifier and the remaining registers `x1-x17` as arguments. Those can either be a value or a memory reference, whose contents hold runtime service-specific data. This calling convention leads to a large input space since the function identifier, the number of parameters, their types, and valid parameter contents are unknown a priori. As we are dealing with proprietary binaries, manual reverse-engineering to derive the interface of registered runtime services is a tedious effort. Therefore, we need an approach to automatically discover the interface of secure monitors and subsequently equip a fuzzer with this additional knowledge to achieve efficient interface exploration.

4 EL3XIR's Approach

EL3XIR is an extensible rehosting and feedback-guided fuzzing framework for secure monitors. Figure 2 provides an overview of EL3XIR's design.

The initial step to fuzz a new secure monitor requires creating a rehosting environment to boot the secure monitor and taking a snapshot at the desired fuzzing location (①). This location is typically the first transition to the untrusted normal world. EL3XIR comes with an extensible rehosting framework that facilitates partial rehosting of secure monitor binaries, addressing **C1** (Section 3). After completing the one-time effort rehosting, EL3XIR takes a snapshot of the booted secure monitor used for the subsequent automatic fuzzing campaign enabling high execution speed due to fast resets.

To address the lack of peripheral models (**C2** in Section 3), EL3XIR monitors the SoC's MMIO regions in the physical address space and employs reflected peripheral modeling to populate MMIO read registers with raw fuzzing input. This technique mimics peripheral behavior and enables EL3XIR to explore error and success control-flow paths succeeding peripheral interactions and, thus, increases the target's code coverage without any manual intervention.

To deal with the challenge of diverse and complex input formats expected by runtime services (**C3** in Section 3), EL3XIR statically analyzes the source code of the REE OS (e.g., a Linux kernel) that interacts with the secure monitor interface to recover target-specific and interface-aware

seeds (②). Then, EL3XIR probes the 32-bit function identifier (`x0` register) used to select the requested runtime service, using coverage-feedback as an oracle to determine if a service backs a given identifier. This process allows EL3XIR to explore unseen services while filtering out false positives that originated from static analysis. Ultimately, this knowledge is used to synthesize an interface-aware harness equipped with domain-specific mutators to generate high-quality inputs to fuzz the secure monitor (③).

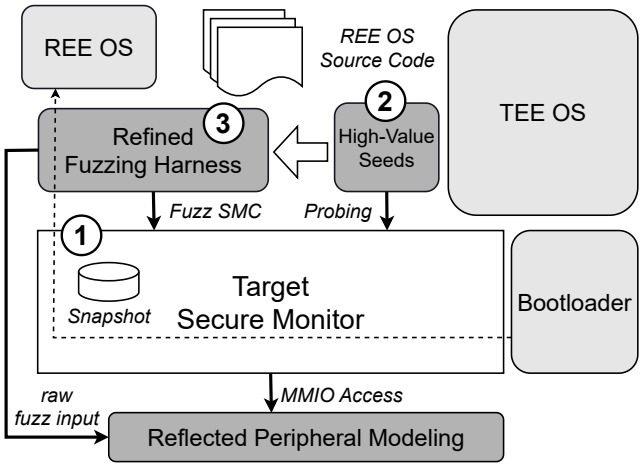

Figure 2: An overview of EL3XIR's design. Software dependencies for partial-rehosting are shown in light gray, while the dark gray components are part of EL3XIR's fuzzing framework. EL3XIR fuzzes a snapshot of the secure monitor by combining an interface-aware refined harness and on-the-fly generated peripheral models leveraging their synergetic effects for extensive code coverage and bug triggering.

4.1 Partial-Rehosting of Secure Monitors

EL3XIR provides a rehosting framework that enables fast and straight-forward partial-rehosting of COTS secure monitor binaries in a pure emulation fashion (i.e., no physical device is required). The framework is designed to drastically reduce the manual effort required to initialize a targeted secure monitor properly. Our partial-rehosting approach addresses **C1** (Section 3) by replacing missing software components with manually crafted stubs and the ability for precise injection of necessary hardware interactions into MMIO registers during boot. We evaluate the manual steps required in Section 6.1.

Our rehosting approach follows an iterative refinement process [24] as illustrated in Figure 3. We start by running the unmodified secure monitor binary in a minimal rehosting environment (e.g., matching CPU architecture and TrustZone extensions) and refine the environment step by step through repeating fidelity evaluations and root cause analysis of misbehavior (e.g., unhandled exceptions or aborts). This process

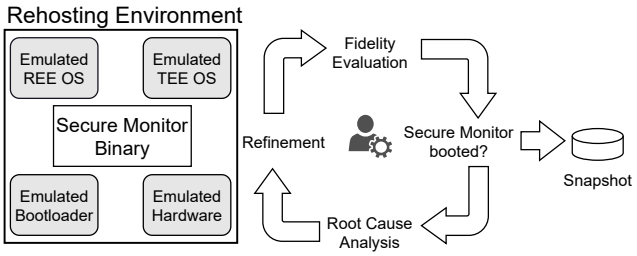

Figure 3: EL3XIR's rehosting framework is designed to help rehost unmodified and proprietary closed-source secure monitor binaries in an iterative refinement process.

continues until the secure monitor successfully boots. Then, EL3XIR takes a snapshot (① in Figure 2), making the manual part a one-time per-target effort. This snapshot is further used for automated fuzzing campaigns.

Our core insight for partial-rehosting is that the secure monitor initialization requires only a small set of emulated software and hardware dependencies. Thus, using a handful of MMIO emulations and less than 20 lines of assembly for the stubs (REE OS, TEE OS, and bootloader), it is feasible to bring the system up until the point where an adversary would take control (EL1 in the normal world), marking our snapshot and fuzzing location (Section 4.3). We decided for manually rehosting the secure monitor boot process because customized boot procedures are challenging to generalize. We discuss this limitation in Section 7. Nevertheless, most of the hardware rehosting effort arises at runtime when the REE OS requests services from the secure monitor and not during the boot process when hardware is only initialized (we solve this problem in Section 4.2).

Our rehosting framework addresses the following challenges when booting a COTS secure monitor in an emulator. First, we describe a systematic way to identify the physical memory layout required to place the secure monitor binary at the correct location in memory. Second, we provide small and adaptable software stubs to easily emulate dependent logical components (i.e., bootloader, TEE OS, and REE OS), directly addressing **C1** (Section 3). Third, our framework allows for precise injection of hardware interactions via breakpoints that are required during boot. Especially MMIO reads of status bits indicating success after the initialization and configuration of peripherals need to be emulated.

Physical Memory Layout. The targeted firmware binary must be appropriately loaded in memory to run correctly [57]. Fortunately, the secure monitor sets up its own virtual address space. For ARMv8-A binaries, the base address of the first level translation table is held in the system register `TTBR0_EL3`. By systematically searching for accesses to this register in the secure monitor binary, we can identify its physical address space.

The necessary steps boil down to (1) finding the access to the register holding the translation table base address and (2) traversing the valid descriptors to locate used physical address ranges. From the permissions in the descriptors, we can pinpoint the executable range (usually only one), thus the physical base address of the secure monitor's executable code. In our experiments, the code parts of the secure monitor where the MMU is enabled (first bit of `SCTLR_EL3` is set) turned out to be a vital oracle to check if the secure monitor is placed correctly.

Software Dependencies. Solving dependencies on other logical software components is one of the challenges faced during partial-rehosting of intertwined software stacks (**C1** in Section 3). The secure monitor directly interacts with three software components: the bootloader, the TEE OS, and the REE OS. We managed to emulate these dependencies across our dataset of seven secure monitors with three small stubs per target comprising no more than 16 AArch64 assembly instructions, demonstrating this task's feasibility. Moreover, most of the stubs are similar and only needed slight adjustment for target-specific values.

The stubs set general purpose registers to pass parameters or return values (e.g., the TEE OS stub indicates a successful boot to the secure monitor) and initiate exception level switches. These stubs are designed to be adaptable and even allow us to influence the secure monitor's behavior. For example, the bootloader passes a boot information structure to the secure monitor, including the TEE and REE OS entry addresses, allowing us to specify these locations in our emulated bootloader stub. Lastly, when the secure monitor gives control to the normal world (e.g., switch to the REE OS), we place a simple REE OS stub just executing SMCs and take a snapshot. From that point on, EL3XIR's fuzzing engine takes over and injects test inputs into the SMC interface, described in more detail in Section 5.3.

Hardware Dependencies. During the boot process, the secure monitor encounters MMIO accesses (e.g., polling of status registers) of peripherals not emulated in the rehosting environment in the first place. To locate these accesses, we can watch for endless loops or unhandled exceptions in the emulator during root cause analysis. By reverse-engineering small code parts around these loops, we can derive reasonable behavior for the concerning MMIO register and refine the emulation with these manually crafted MMIO return values. Note that this is a manual step of peripheral modeling which we evaluate in Section 6.1. Because hardware interactions are intertwined with software dependencies (i.e., they may appear alternatingly during the iterative refinement process), fully automating the boot process is out of scope for EL3XIR. However, during the subsequent fuzzing campaign, we leverage reflected peripheral modeling (Section 4.2) to handle MMIO interactions automatically.

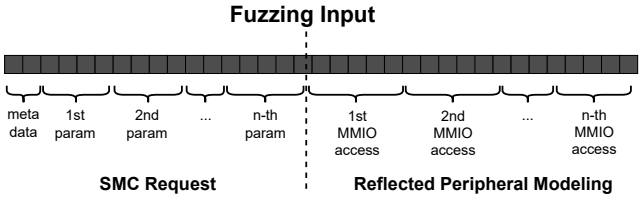

Figure 4: EL3XIR partitions the fuzzing input stream. First, all parameters for the SMC request are populated, then the remaining bytes emulate peripheral behavior in MMIO regions.

4.2 Reflected Peripheral Modeling

After a snapshot of the booted secure monitor is taken, EL3XIR's automated fuzzing campaign can start. One of the challenges during the fuzzing of the secure monitor is to handle hardware interactions in a meaningful way without manual intervention to overcome coverage walls (**C2** in Section 3). In contrast to the boot process, the runtime services exposed by the secure monitor have a more comprehensive range of hardware dependencies making a manual emulation approach infeasible given the diversity of hardware platforms.

To overcome these obstacles, EL3XIR repurposes parts of the fuzzing input to induce *reflected peripheral modeling* (Figure 2). When a memory read operation occurs from defined MMIO regions, EL3XIR captures this event and populates the corresponding MMIO region with fuzzing input simulating the peripheral's behavior.

Figure 4 gives an overview of how EL3XIR partitions the fuzzing input stream. After all parameters for the SMC request are populated, the remaining input is used on a per-MMIO access basis. This technique enables EL3XIR to emulate highly flexible and realistic peripheral behavior.

Our approach ensures that the secure monitor *eventually* receives meaningful values when reading from MMIO registers. Even better, because EL3XIR has complete control over the MMIO behavior, it can generate values that exercise both success and error conditions resulting in extensive code coverage.

4.3 Interface-aware Fuzzing

While Section 4.1 and Section 4.2 describe approaches that increase the fidelity of the rehosting environment, EL3XIR also tackles the orthogonal challenge **C3** (Section 3) of dealing with complex and diverse input formats by generating an interface-aware fuzzing harness. EL3XIR's fuzzing approach is interface-aware because it knows how to interact with the SMC interface using valid function identifiers and respective expected arguments. For an effective exploration of the secure monitor's attack surface, EL3XIR leverages the REE OS's knowledge about the structure and semantics of

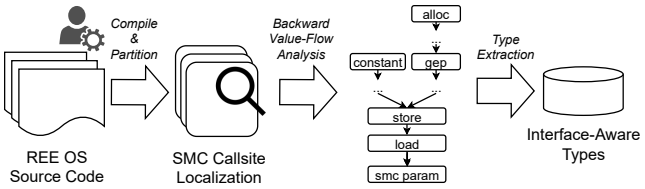

Figure 5: EL3XIR's interface inference is based on a backward value-flow analysis on a manually-picked REE OS source code partition. EL3XIR automatically identifies SMC call sites, localizes corresponding parameters in the value-flow graph, and recovers type-aware function prototypes for the SMC interface.

the SMC interface to generate and mutate highly targeted and meaningful test cases. However, the REE OS is not the only software component interacting with the SMC interface (e.g., bootloader and TEE OS). Thus, results may be incomplete (Section 7). To tackle this problem, EL3XIR performs an additional probing phase to identify all function IDs backed by a corresponding runtime service. The complete list of function IDs and the partially recovered interfaces (② in Figure 2) are combined with domain-specific mutators to generate an interface-aware fuzzing harness (③ in Figure 2).

Step 1: Interface Recovery

EL3XIR utilizes static analysis on relevant parts of the REE OS's code to extract function signatures and parameter values that correspond to the interfaces exposed by runtime services (Figure 5). The intuition is that the REE OS possesses the knowledge required to communicate effectively with the runtime services provided by the secure monitor. By analyzing hand-picked partitions of the REE OS code, EL3XIR can automatically retrieve the arity and semantic type of arguments involved in calls that utilize the SMC interface. We provide an excerpt of REE OS code that prepares and sends an SMC and the corresponding successfully recovered interface by EL3XIR in Appendix B (Listing 2 and Listing 3).

SMC Call Site Localization. The REE OS invokes the secure monitor using the SMC instruction. To identify all call sites, we employ an over-approximating heuristic due to the SMC instruction often being concealed behind indirect function calls (e.g., wrapper functions) that require sophisticated techniques to resolve [41]. Instead, EL3XIR automatically scans all function calls included in the REE OS partition and adds those closely matching the SMCCC to the list of potential SMC call sites. After empirical analysis, we implemented our heuristic to focus on calls with at least five arguments or a scalar value as their first argument. This simple approach ensures that we primarily consider more complex function signatures while filtering out incorrect candidates. The resulting false positives will later be pruned, leaving only high-quality SMC interface descriptions. We provide an overview of recovered interfaces and evaluation of our heuristic in Section 6.2.

SMC Parameter Type Identification. Given potential SMC call sites, EL3XIR performs a backward depth-first search along the value-flow graph to recover type information and scalar values of arguments, starting with each argument node of a call site as a sink. For each source node, we identify scalar values passed directly (i.e., as a constant) or indirectly (i.e., as a variable) to SMC call site locations. We retrieve semantic type information by extracting associated compiler metadata from memory locations whose values are propagated to arguments for SMC call sites.

Function Identifier Probing and Pruning. The static analysis of the open-source REE OS produces a list of potential function identifiers along with their corresponding parameters (number of arguments, types, and possibly constants). However, this list may be incomplete and might contain false positives. For instance, the secure monitor could implement runtime services that the REE OS does not use (e.g., the TEE OS could have exclusive access to some services) or our previous heuristics could simply miss edge cases.

To resolve these cases, EL3XIR performs a "probing and pruning" phase. We probe all 2^{32} function identifiers (passed via `x0`) and leverage the coverage as an oracle to determine if a function identifier is backed by a service. For instance, the coverage map will be the same for the default cases whereas entering an existing service will lead to distinct coverage profiles. Additionally, we prune the list of known function identifiers if it contains function identifiers not backed by services.

Step 2: Harness Synthesis

By utilizing the comprehensive collection of function identifiers and recovered interfaces supported by the secure monitor, we generate a refined fuzzing harness (③ in Figure 2) that leverages this target-specific knowledge and incorporates additional domain-specific mutators for the effective exploration of the secure monitor's interface.

Interface-aware Input Injection. As depicted in Figure 4, the fuzzing harness starts to decode the raw bytes of the fuzzing stream to generate secure monitor requests. To this end, the initial bytes represent metadata that defines the structure of the SMC request (e.g., function arity and type of parameters). At the same time, the harness maps the subsequent raw bytes to the SMC interface by filling registers and memory regions with values according to their inferred type. For example, the first parameter is a four-byte scalar value copied into the `x0` register to define the function identifier (first entry in Listing 3). When the SMC request is injected (all recovered parameters are filled with values), all remaining bytes are held for reflected peripheral modeling (Section 4.2).

Domain-specific Mutators. EL3XIR incorporates three domain-specific mutators into its fuzzing harness, enabling

targeted mutations of fuzzing inputs to generate high-quality test cases. It is important to note that these mutators are tailored explicitly for secure monitor implementations. However, since EL3XIR's harness is a standalone component, the mutators can be customized per-target.

The first mutator ensures that memory references primarily fall within the memory range of the REE OS. This approach is rooted in the expectation that secure monitors implement validation logic to verify that parameter addresses point to memory within the REE OS (Listing 1). If EL3XIR successfully recovered memory reference type information for a parameter (e.g., register `x2` with type `phys_addr_t` in Listing 3), our mutator ensures this parameter primarily holds valid addresses while intentionally introducing occasional invalid ones to identify potential unsanitized memory pointers.

The second mutator addresses the alignment of size values for memory buffers in communication with peripherals via MMIO. Hardware devices often have specific alignment requirements for accessing data, and failing to meet these requirements can lead to hardware errors. As a result, secure monitors incorporate alignment checks to ensure accurate communication with these peripherals that can act as coverage walls (Listing 1). If a size parameter is recovered (e.g., register `x3` with type `size_t` in Listing 3), our mutator strives to satisfy these alignment checks in most cases while still allowing for the rare possibility of sending misaligned data to ensure extensive code coverage.

Lastly, the third mutator aims to reuse known interfaces for runtime services whose interface has not been recovered. The underlying idea is that low-level firmware interfaces that adhere to the same calling convention, such as the SMCCC [5], often share identical function signatures.

5 Implementation Details

As a framework that enables effective end-to-end fuzzing of COTS secure monitors, we designed EL3XIR with expandability and adaptability in mind. In this section, we describe implementation details of EL3XIR's three main components (Section 4). This includes the partial-rehosting framework (Section 5.1), reflected peripheral modeling (Section 5.2), and interface-aware fuzzing (Section 5.3).

5.1 Secure Monitor Rehosting Framework

EL3XIR's rehosting framework (Section 4.1) extends avatar^2 (version 1.3.1) [45] and its configurable machine which is integrated into QEMU [13]. We choose QEMU as our core emulator because it has direct support for ARMv8-A CPUs and the TrustZone extensions. By leveraging avatar^2's flexible physical memory description, we can adjust the memory layout to align with the requirements of the targeted secure monitor and directly write software stub code into memory during setup. EL3XIR comes with prototypes for the bootloader, REE OS, and TEE OS which can be easily adjusted during the iterative refinement rehosting process. We solve the injection of precise hardware emulation during boot by setting breakpoints at the location of failing MMIO read operations and implementing handler functions that allow to *directly* write expected values into destination registers of the load instruction.

5.2 Reflected Peripheral Modeling

EL3XIR repurposes fuzzing data to dynamically emulate MMIO interactions (Section 4.2). This enables EL3XIR to overcome coverage walls caused by MMIO reads by providing fuzzing input at the location of MMIO memory regions mimicking peripheral behavior. For this feature, we modified avatar^2's remote memory peripheral to request fuzz data when a `read` operation occurs. Consequently, we replaced all physical memory regions belonging to MMIO regions with our modified memory peripheral that will return fuzzing data to our target. We ignore MMIO `write` operations because they only result in internal hardware state changes and our fuzzer sends the expected peripheral output to the target on subsequent MMIO reads.

5.3 Interface-Aware Fuzzing

Interface Recovery. EL3XIR's static analysis is based on LLVM (version 14.0.0) and SVF (version 2.5) [54]. We generate LLVM-bitcode by compiling the REE OS source code without any optimizations to retain metadata. Furthermore, EL3XIR only analyzes a smaller hand-picked partition as building the value-flow graph for the entire kernel is infeasible [37]. Picking a REE OS partition can be as simple as including the driver files that communicate with the targeted vendor's secure monitor. Finding these via keywords in the source code is manageable manual effort as the partitions only comprise of a couple of files (Section 6.2).

EL3XIR performs a depth-first search on the value-flow graph for each argument of a potential SMC call site, previously identified by scanning all call instructions. The execution of an SMC instruction is usually written in assembly code and wrapped in a function that consumes the corresponding register values as parameters. Because the kernel makes heavy usage of indirect calls for these wrappers, statically identifying SMC call sites is non-trivial as function pointers may only be resolved at runtime. While there exist more holistic solutions for this problem [41], we employed an over-approximating heuristic (Section 4.3), that was sufficient for this task in combination with the fact that REE OS partitions remained relatively small.

During backward tracing of the value-flow graph, we search for LLVM instructions that load, store or calculate pointer addresses to trigger EL3XIR's extraction routine (example

in Appendix B). While the extraction of scalar values from the value-flow graph was trivial, extracting semantic C-types required more effort. EL3XIR does this by traversing LLVM intrinsic function calls and matching parameters to find LLVM metadata using the source code line as an indicator. We hold the results in a list with the basic block being an identifier. Clustering function prototypes purely by the function identifier (`x0`) is not possible because there might be multiple interfaces for one function identifier. For example, a runtime service might implement a sub-service that uses `x1` as a command identifier. Every recovered interface is saved as a `CVS` file in which each tuple represents the associated register, a C-Type, and an optionally recovered constant (e.g., Listing 3).

Fuzzing Framework. We implemented EL3XIR's fuzzing approach by extending *qemuafl*, the QEMU user-mode of AFL++ (version 3.15a) [26] to support full-system emulation. We achieve this by integrating avatar^2's configurable machine and a forkserver capable of executing copies of the entire VM as child processes. Because QEMU is a multi-threaded process, forking complete VMs turned out to be a non-trivial task. Our solution is inspired by TriforceAFL [30]. We shut down the CPU thread on reaching the initial fuzzing location (e.g., the first instruction in the REE OS) and spin up the forkserver that creates a new CPU thread for each child. By loading the snapshot of the targeted secure monitor beforehand, each new child is initially in a reset state.

Unlike TriforceAFL, our fuzzing harness is implemented as a shared object, thus being part of the QEMU process which saves the effort to run an agent in the VM. Instead, the harness is triggered when a defined address is hit during emulation (e.g., the first instruction in the REE OS stub). EL3XIR's harness takes AFL++'s input via shared memory and writes test cases directly into the VM's registers and memory. For the detection of crashes, we modified the CPU thread to abort if it encounters an exception raised by the MMU (e.g., data or prefetch abort) and forward the signal to AFL++ indicating a crash. Coverage information is fed back to AFL++ through a bitmap in shared memory which is filled by the CPU thread during the translation of instructions.

6 Evaluation

Our evaluation of EL3XIR answers these research questions:

RQ1 Are the manual steps of EL3XIR's partial-rehosting approach feasible? (Section 6.1)

RQ2 Is EL3XIR's interface recovery effective in prototyping function signatures of runtime services? (Section 6.2)

RQ3 Does the reflected peripheral modeling technique improve EL3XIR's exploration capability? (Section 6.3)

RQ4 How does EL3XIR compare against existing methods to fuzz TEE firmware? (Section 6.3 and Section 6.4)

RQ5 Can EL3XIR be used to uncover previously unknown bugs in real-world secure monitors? (Section 6.4)

To answer these questions, we conduct several experiments targeting the secure monitors of different platforms (Table 1). We use a diverse set of platforms from several reputable vendors. EL3XIR enables fuzzing *proprietary* secure monitors. For the open-source targets, we compiled the secure monitors according to the available platform-specific instructions and only used the binaries. Moreover, we obtained proprietary secure monitors from COTS devices or firmware updates.

Table 1: Dataset of secure monitors used for EL3XIR's evaluation. Proprietary (Prop.) indicates if the implementation is closed-source.

Vendor	SoC	Prop.	Size
Intel	Stratix 10 SoC FPGA		45 KB
NXP	i.MX 8 Series		37 KB
Xilinx	Zynq MPSoC		49 KB
Nvidia	Tegra X2 T186		75 KB
Huawei	Kirin 659	✓	112 KB
Samsung	Exynos 7420 Octa	✓	192 KB
Samsung	Exynos 8890 Octa	✓	144 KB

6.1 Manual Effort for Rehosting

We now quantify the manual effort required to build an initial rehosting environment that can be used to properly boot the targeted secure monitor (**RQ1**). Naturally, we were required to carry out some reverse engineering efforts to implement target-specific software stubs and hardware interactions. However, while rehosting the first secure monitor did take multiple weeks, we iteratively improved EL3XIR's rehosting framework which cut this time down to only some days.

Software Stubs. To solve software dependencies, we replace logical dependent components with software stubs implemented in assembly (Section 4.1). We create the bootloader stub by identifying the entry point in the secure monitor binary and reverse-engineer expected arguments (e.g., boot information structure for the TEE OS and REE OS). For each target, we implemented bootloader stubs with around five lines of assembly, that fill registers with expected values and execute a return instruction to jump to the secure monitor's entry address. We replace the TEE OS with a four-line assembly stub that populates the register `x0` with a value indicating the successful initialization of the TEE OS and executes an SMC, handing control back to the secure monitor. We replace the REE OS with two assembly instructions executing the exception level switch to EL3. EL3XIR provides skeletons for each software stub thus the effort to extend it with a new

secure monitor boils down to changing a handful of values in the boot information structures and register parameters.

Hardware Interactions. To boot the secure monitor, we need to place manually crafted hooks, that emulate successful peripheral behavior to keep the target running (Section 4.1). In most scenarios, we can allow the secure monitor to execute from its entrypoint and observe any instances when it gets stuck (e.g., enters an infinite loop) or an exception is thrown. This behavior often indicates the polling of MMIO status registers. We briefly analyze the assembly code at the error location and then insert a hook that sets the relevant destination registers to the expected values to satisfy the control flow graph constraints validating successful peripheral initialization. One exception for MMIO polling were the Exynos-based secure monitors. Depending on MMIO registers, these secure monitors would enable the MMU at different locations, resulting in a crash if the translation tables were not properly set up yet. By investigating the crash location, we identify involved registers, and guide the control flow to boot the secure monitor. In total, we needed one (Kirin 659), two (i.MX 8), three (Stratix 10), three (Tegra T186), seven (Zynqmp MPSoC), eight (Exynos 8890), and ten (Exynos 7420) of these MMIO hooks for the respective chipset.

As detailed in this section, partially rehosting the diverse set of open- and closed-source secure monitors from Table 1 to the point where the REE OS is booted, is feasible (**RQ1**).

6.2 Interface Recovery

In this section, we evaluate EL3XIR's interface recovery (Section 4.3). EL3XIR operates on REE OS partitions handpicked for each target. In total, the partitions consisted of four, eleven, five, and four REE OS files for the Stratix 10, i.MX 8 Series, Zynq MPSoC, and Tegra X2 T186 respectively. The size of partitions ranged from around 400KB to 1.4MB with EL3XIR's analysis time remaining within minutes per target. To answer **RQ2** we compare the number of successfully recovered interfaces Σ_{rt}^{rec} to the total number of offered runtime services by the targeted secure monitor Σ_{rt}, as depicted in Table 2. As we do not have a reliable ground truth for the closed-source targets, we only focus on the open-source ones. Overall EL3XIR recovers between 36% and 61% of all runtime services for our targets. After closer analysis, we found that nearly all of the remaining runtime services were not exercised by the REE OS but rather previous bootloader stages running in the REE which are not part of EL3XIR's analyzed set as there is usually no source code available (Section 7). For example, EL3XIR only recovered 36% of runtime services for the Nvidia Tegra target which is rooted in the fact that our analyzed REE OS did not support all functionality offered by the secure monitor (e.g., the Linux kernel did not support the Software Delegated Exception Interface).

False Positives. In order to evaluate the accuracy of EL3XIR's interface recovery heuristic (Section 4.3), we count the rate of falsely recovered interfaces during static analysis of the REE OS partitions. In column Σ_{rt}^{cand} of Table 2 we count the total number of interface candidates after EL3XIR's static value-flow analysis. After analyzing the results we identified duplicate candidates with the same function identifier Σ_{rt}^{dup}. EL3XIR included these because it successfully recovered multiple scalar values for other registers (e.g., register `x1` is used as a command identifier for a sub-service). Furthermore, EL3XIR recovered interfaces that were not offered by the targeted secure monitor but are still part of the REE OS code Σ_{rt}^{REE}, because SoCs often share REE OS drivers. If we consider candidates that have no or no valid function identifier (Σ_{rt}^{nofid} and $\Sigma_{rt}^{wrongfid}$) we reach between 57 and 96 false positives, with percentage rates 30% and 54%, respectively. After sorting out candidates without any function identifier we can narrow this down to 0% and 12%, resulting in false positive rates between zero and twelve percent. In the end, EL3XIR's probing phase will also sort out candidates with no valid function identifier as they will all result in the same coverage profile.

6.3 Coverage

We provide a comparative evaluation of EL3XIR's achieved coverage (**RQ4**) and investigate the effect of reflected peripheral modeling on the exploration capability (**RQ3**).

To establish a coverage baseline, we repurpose the fuzzing harness originally developed for PartEmu [32]. While PartEmu is the closest work to ours, it does not include evaluation of secure monitors due to their heavy reliance on hardware interactions (**C2** in Section 3). Although the PartEmu prototype was not publicly released, we reimplemented their fuzzing harness based on their publication. To promote reproducibility and support future research, we will make all of our artifacts publicly available.

The PartEmu harness is not interface-aware, meaning that it does not employ any knowledge about the function signatures of runtime services. Thus, we dubbed this approach adhering to the generic SMCCC *interface-unaware*. Additionally, we introduce a *naive* MMIO modeling approach where all MMIO ranges are backed with zeroed `rw` memory always returning the last stored value. EL3XIR is designed such that both *interface awareness* and *reflected peripheral modeling* can be turned on or off before starting a fuzzing campaign.

In total, we compare the following four different fuzzing configurations of EL3XIR targeting the seven targets from Table 1:

- $iface^{-}/mmio^{-}$: Iface-unaware + naive MMIO (PartEmu [32])
- $iface^{+}/mmio^{-}$: Iface-aware + naive MMIO
- $iface^{-}/mmio^{+}$: Iface-unaware + reflected peripheral modeling
- $iface^{+}/mmio^{+}$: Iface-aware + reflected peripheral modeling

Table 2: Σ_{rt} states the number of runtime services we found in each target binary (ground truth), while Σ_{rt}^{rec} counts all successfully recovered ones by EL3XIR (percentage share in parentheses). EL3XIR's static analysis grants Σ_{rt}^{cand} interface candidates with Σ_{rt}^{dup} duplicates. We filter out candidates only present in the REE OS partition (Σ_{rt}^{REE}), those without (Σ_{rt}^{nofid}) or a wrong ($\Sigma_{rt}^{wrongfid}$) function identifier. The false positive rates are given in percentage in parentheses with respect to Σ_{rt}^{cand}.

SoC	Σ_{rt}	Σ_{rt}^{cand}	Σ_{rt}^{dup}	Σ_{rt}^{REE}	Σ_{rt}^{nofid}	$\Sigma_{rt}^{wrongfid}$	Σ_{rt}^{rec}
Stratix 10 SoC FPGA	122	177	2 (1%)	0 (0%)	96 (54%)	5 (3%)	74 (61%)
i.MX 8 Series	68	89	0 (0%)	9 (10%)	42 (47%)	8 (9%)	30 (44%)
Zynq MPSoC	91	188	24 (13%)	31 (17%)	57 (30%)	23 (12%)	53 (58%)
Tegra X2 T186	78	55	2 (4%)	2 (4%)	23 (42%)	0 (0%)	28 (36%)

Figure 6 illustrates the accumulated edge coverage over time for all four fuzzing configurations. For each target, we run each fuzzer eight times for 24 hours, and plot the average coverage and the min/max coverage (shaded area). We run these experiments on a 16-core Intel Xeon Gold 5218 processor (hyperthreading disabled) with 64GB of RAM and limit each 24-hour run to one core.

Reflected Peripheral Modeling. One of EL3XIR's technical contributions is *reflected peripheral modeling* (Section 4.2). In contrast to the *naive* approaches $^{iface^-}/_{mmio^-}$ and $^{iface^+}/_{mmio^-}$, we leverage the fuzzing input to populate read accesses to MMIO registers. We hypothesize that this technique contributes to code exploration (**RQ3**) as naive approaches are not able to overcome MMIO coverage walls (**C2** in Section 3). If we compare the reflected peripheral modeling configurations ($^{iface^-}/_{mmio^+}$ and $^{iface^+}/_{mmio^+}$) to their naive counterparts, we can see a robust gap in edge coverage across all seven targets (Figure 6).

For a deeper evaluation, we identify coverage walls in the four open-source target's runtime services and check if EL3XIR overcomes them through successful MMIO read operations provided by reflected peripheral modeling. We present the results in Table 3 showcasing that EL3XIR's reflected peripheral modeling is used in up to 87% of runtime services Σ_{rt}^{wall} at least once. In all affected ones, reflected peripheral modeling leads to an increase in code coverage. By rerunning all resulting inputs after the fuzzing campaign and counting MMIO access addresses, we identify all unique MMIO addresses across all runtime services in column Σ_{MMIO}. EL3XIR successfully models several unique MMIO register accesses across all targets. Additionally, we count the total number of uniquely returned MMIO *values* for these registers which resulted in 7742, 8815, 1802, and 1866 different values returned for the Stratix 10, i.MX 8 Series, Zynq MPSoC, and Tegra X2 T186 chipsets, respectively. In summary, EL3XIR's reflected peripheral modeling automatically emulates the behavior of several peripherals (with multiple MMIO registers being involved Σ_{MMIO}) by providing thousands of different return values during fuzzing, exercising both success and failure states of MMIO handling routines. Therefore, recent approaches trying to manually emulate these peripheral interactions [32] can not be considered scalable solutions.

Comparison with the State-of-the-Art. By comparing the achieved coverage of $^{iface^-}/_{mmio^-}$ against $^{iface^+}/_{mmio^-}$ (Figure 6), we can reason about **RQ4**. Taking $^{iface^-}/_{mmio^-}$ as an optimistic baseline to represent prior work on fuzzing TEE firmware, we can see that EL3XIR using its contributions $^{iface^+}/_{mmio^+}$, outperforms the state-of-the-art in terms of coverage in every single experiment. Furthermore, we observe that $^{iface^-}/_{mmio^+}$ and $^{iface^+}/_{mmio^-}$ reach coverage levels between the base line and EL3XIR. This indicates that EL3XIR's technical contributions (Section 4.2 and Section 4.3) indeed complement each other. Additionally, when comparing the maximum edge coverage reached (Table 4) by EL3XIR's full configuration ($^{iface^+}/_{mmio^+}$) and our adaption of PartEmu [32] ($^{iface^-}/_{mmio^-}$), we can see that EL3XIR increases the reached coverage by up to 51%.

Table 3: Total number and percentage of runtime services affected by MMIO coverage walls Σ_{rt}^{wall}. Σ_{MMIO} states the number of unique MMIO read registers successfully modeled by EL3XIR across all affected runtime services.

SoC	Σ_{rt}	Σ_{rt}^{wall}	Σ_{MMIO}
Stratix 10	122	51 (42%)	37
i.MX 8 Series	68	6 (9%)	58
Zynq MPSoC	91	79 (87%)	5
Tegra X2 T186	78	24 (31%)	158

6.4 Finding Bugs

Our fuzzing campaigns resulted in a variety of crashes across all seven targets. After deduplication and root cause analysis, we identified 34 bugs with 17 being security relevant.

We compare the number of unique bugs found by the baseline fuzzer and full-featured EL3XIR to reason about **RQ4**. In total, EL3XIR found 34 and $^{iface^-}/_{mmio^-}$ found 19 unique bugs. EL3XIR triggered significantly more crashes, and the

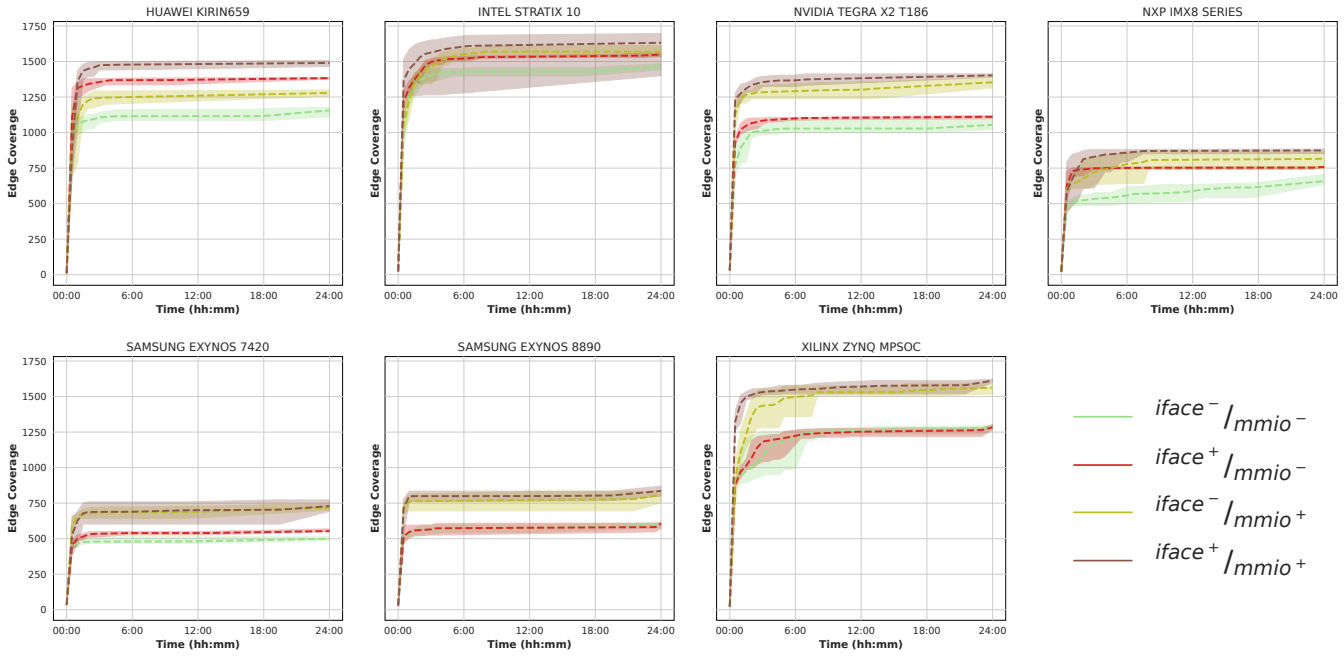

Figure 6: Coverage plots for our seven targets with all four fuzzing configurations. In its full configuration ($^{iface^+}/_{mmio^+}$) EL3XIR achieves the most coverage for every target, while the state-of-the-art fuzzing approach ($^{iface^-}/_{mmio^-}$) finds the least coverage. This shows that EL3XIR's contributions and their interplay are effective in better exercising the secure monitor's interface and overcoming coverage walls through MMIO modeling.

ones found by $^{iface^-}/_{mmio^-}$ are a strict subset of the crashes found by $^{iface^+}/_{mmio^+}$. EL3XIR's crashes contain 17 security-critical bugs whereas the $^{iface^-}/_{mmio^-}$ mode's set of bugs only contains 10. These bugs are distributed over five out of the seven targets, and contain arbitrary read and write primitives that can be leveraged to breach the confidentiality and integrity of the secure monitor and therefore TEE. We present a detailed overview of affected targets and descriptions for security-relevant bugs in Table 5.

To answer **RQ5**, we manually triaged all vulnerabilities and responsibly disclosed them to the respective vendors. While we selectively tested and reproduced some crashes on production devices as well, we did not have access to all devices in question, thus by including the vendor we can verify the reproducibility of bugs found by EL3XIR. In summary, vendors assigned six CVEs and six additional crashes were confirmed with patches already rolled out.

Case Study I: 0-day bugs in Intel's Stratix 10 SoC FPGA EL3XIR uncovered seven previously unknown vulnerabilities in the secure monitor implementation running on Intel's Stratix 10 SoC FPGA for which four CVEs were assigned. The secure monitor implements functionality to interact with a Field Programmable Gate Array (FPGA) for the accelerated and protected calculation of cryptographic actions (e.g., AES). The production device is running a Linux kernel driver that sends a payload to runtime services via shared kernel space memory. The secure monitor receives the physical address and copies the payload to secure memory in order to prepare the request for the FPGA which is only accessible in the TEE.

We identified that the bugs cause bounds validations for `memcpy` calls resulting in out-of-bound writes, corrupting secure memory. Thus, an attacker can overwrite the secure monitor's memory with an arbitrary payload. Due to the lack of stack canaries or ASLR, developing an exploit to gain code execution with this primitive is straightforward. For example, an attacker can overwrite the page tables to map additional memory on behalf of the secure monitor and inject executable payload. As a result, a privilege escalation to the highest privilege level (EL3) is possible, demolishing any security guarantees provided by ARM TrustZone.

Case Study II: N-day bugs in Huawei's Kirin 659 To validate that EL3XIR can also find bugs in proprietary secure monitor implementations, we verified that EL3XIR can reproduce already known bugs in TEE firmware running on Huawei P20lite devices based on the Kirin 659 SoC. Huawei decided to implement multiple runtime services on top of ARM's reference implementation, including drivers for replay-protected memory blocks (RPMB). A statically shared memory location is used to exchange data between the non-trusted kernel drivers and secure runtime services.

In total, EL3XIR found seven unique crashes scattered across three different runtime services, of which three were

Table 4: Maximal edge coverage reached by EL3XIR for each target. We compare the percentage increase of each of EL3XIR's configuration to our baseline PartEmu adaptation ($iface^-/mmio^-$) in parentheses with the highest increase being marked in bold.

Total Edges Covered				
SoC	$iface^-/mmio^-$	$iface^+/mmio^-$	$iface^-/mmio^+$	$iface^+/mmio^+$
Huawei Kirin 659	1178	1388 (18%)	1300 (10%)	1507 (**28%**)
Intel Stratix 10	1485	1560 (5%)	1612 (9%)	1697 (**14%**)
Nvidia Tegra X2 T186	1088	1121 (3%)	1387 (27%)	1413 (**30%**)
NXP i.MX 8 Series	706	762 (8%)	864 (22%)	884 (**25%**)
Samsung Exynos 7420 Octa	513	569 (11%)	747 (46%)	773 (**51%**)
Samsung Exynos 8890 Octa	609	616 (1%)	834 (37%)	870 (**43%**)
Xilinx Zynq MPSoC	1303	1302 (0%)	1594 (22%)	1622 (**24%**)

in the RPMB runtime service. Two of those crash locations turned out to be arbitrary write and read primitives respectively. As found before [56], both crashes are exploitable bugs that can lead to code execution at EL3. EL3XIR was capable of pinpointing the exact crash locations used during the privilege escalation exploit. In essence, an attacker can map non-secure memory on behalf of the secure memory, fill it with payload code, and by overwriting control flow data execute the payload.

7 Discussion

In this section, we discuss design decisions and future research directions of EL3XIR.

Encrypted Binaries. EL3XIR is a rehosting-based fuzzing framework. Thus, similar to other rehosting approaches, we need access to the plaintext target binaries. Obtaining these binaries might be challenging for COTS devices that receive encrypted firmware updates. For instance, we observed that the secure monitor is encrypted in recent updates for the newest phones of Samsung and Huawei.

Manual Effort. Although EL3XIR reduces the manual effort involved in rehosting COTS firmware binaries compared to the state-of-the-art approach described by Harrison et al. [32], we still require the implementation of software stubs to emulate dependent components and smaller hardware emulation during the boot process (Section 4.1). Automating the initial rehosting process proved to be challenging due to the intertwined and non-standardized sequences of the boot flow. For example, Samsung's secure monitor tried to boot separate loadable firmware components before switching to the REE OS. In our experiments, we manually bypassed these loading sequences to focus our fuzzing efforts on the secure monitor and to cut down on additional rehosting effort. Therefore, while EL3XIR still requires some manual work from an expert, we consider our design to be a significant advancement toward achieving a more comprehensive automation solution for rehosting highly-intertwined firmware components.

Modeling Direct Memory Access. Furthermore, EL3XIR's MMIO modeling approach (Section 4.2) cannot deal with direct memory access (DMA). While we did not encounter any DMA-capable peripherals in our dataset, we note that these peripherals are currently out of scope. Supporting DMA-capable peripherals would need dedicated handling partially addressed by prior work [44].

Closed-source Consumers. EL3XIR's approach to recovering interfaces for unknown runtime services requires the source code of corresponding consumers of the SMC interface. In its current implementation EL3XIR only focuses on open-source REE OS drivers to retrieve interfaces. While we did not encounter any proprietary drivers communicating with the secure monitor, vendors might deploy closed-source drivers which would render EL3XIR's current implementation ineffective. However, there is work trying to recover interfaces from unknown function signatures on a binary-level [19]. Beyond that, additional consumer components (e.g. TEE OS or subsequent bootloader) could also be analyzed to collect even more function prototypes for runtime services not exercised by the REE OS. As interface awareness is only one of EL3XIR's components, we believe that these approaches would complement our work.

Horizontal Extension. Since PartEmu [32] was severely limited by the lack of interface- and state-awareness, reflected by low coverage results, our future plans include the extension of EL3XIR to fuzz proprietary TEE OSes and TAs. Furthermore, we believe that EL3XIR's approach is applicable to fuzz components of other highly-intertwined firmware binaries like bootloader components (e.g., uboot or UEFI).

8 Related Work

Prior research on TrustZone-based systems focuses on static analysis of TEE components [15, 17, 27, 42, 52, 53] with relatively few researchers attacking the SMC interface to the secure monitor [43]. Cerdeira et al. [18] give an overview of vulnerabilities in TrustZone-based systems, including publicly

Table 5: Overview of bugs found by EL3XIR.

SoC	Affected Runtime Service	Description	Disclosure
Intel Stratix 10	`intel_fcs_aes_crypt_init`	Buffer Overflow	CVE-2022-38787, CVE-2023-49614
Intel Stratix 10	`intel_fcs_ecdsa_sha2_data_sig_verify_update_finalize`	Buffer Overflow	CVE-2023-22327
Intel Stratix 10	`intel_fcs_ecdh_request_finalize`	Buffer Overflow	CVE-2023-22327
Intel Stratix 10	`intel_fcs_ecdsa_hash_sign_finalize`	Buffer Overflow	CVE-2023-22327
Intel Stratix 10	`intel_fcs_ecdsa_hash_sig_verify_finalize`	Buffer Overflow	CVE-2023-22327
Intel Stratix 10	`intel_fcs_mac_verify_update_finalize`	Buffer Overflow	CVE-2023-22327
Intel Stratix 10	`intel_fcs_decryption`	Buffer Overflow	CVE-2024-22390
NXP i.MX 8 Series	`imx_gpc_set_affinity`	Buffer Overflow	Confirmed
NXP i.MX 8 Series	`imx_gpc_pm_domain_enable`	Buffer Overflow	Confirmed
NXP i.MX 8 Series	`imx_gpc_hwirq_unmask`	Buffer Overflow	Confirmed
NXP i.MX 8 Series	`imx_gpc_set_wake`	Buffer Overflow	Confirmed
NXP i.MX 8 Series	`imx_gpc_hwirq_mask`	Buffer Overflow	Confirmed
Xilinx Zynq MPSoC	`pm_api_clock_get_name`	Improper Input Validation	CVE-2023-31339
Nvidia Tegra X2 T186	`psci_affinity_info`	Improper Input Validation	Confirmed by ARM
Nvidia Tegra X2 T186	`sdei_interrupt_bind`	Improper Input Validation	CVE-2023-49100
Huawei Kirin 659 SoC	`smc_rpmb_state`	Buffer Overflow	Rediscovered [56]
Huawei Kirin 659 SoC	`smc_rpmb_state`	Buffer Overflow	Rediscovered [56]

reported bugs in secure monitor implementations. EL3XIR is the first automatic testing platform tailored for fuzzing exactly this proprietary TEE firmware component.

Dynamically analyzing proprietary TEE components is challenging. On-device approaches [14,16] lack the introspection capabilities to enable feedback-guided fuzzing. Harrison et al. [32] report on the feasibility of enabling dynamic analysis for TEE OSes and TAs by manually building emulators. In contrast, EL3XIR addresses the unexplored secure monitor, and advances approaches like PartEmu by adding interface awareness and automated peripheral modeling.

EL3XIR relies on rehosting techniques [24] to enable dynamic analysis. The majority of rehosting work focuses on small, standalone embedded firmware [21,25,38,50], whereas EL3XIR addresses highly-intertwined components from a significantly larger software stack. However, these approaches share the problem of correct peripheral behavior modeling. HALucinator [21] is based on manually crafted high-level emulation hooks that are infeasible to build for the large number and variety of undocumented custom hardware encountered in secure monitor implementations. Another line of research [25, 50, 58] aims to automate MMIO register behavior. These works have a heavy-weight static or dynamic analysis phase in common. In comparison, EL3XIR uses coverage feedback as an oracle to assess the quality of generated MMIO behavior, and smoothly integrates its reflected peripheral modeling approach into the fuzzing process.

Interface-aware fuzzing has been previously employed for various targets [12, 16, 22, 23, 47]. TEEzz [16] is the first to target ARM TrustZone firmware by trying to recover TEE interface-aware seeds on production devices via recording interactions between the Client Application and TA. Instead EL3XIR automatically derives interface-aware information by statically analyzing the REE OS, eliminating the dependency on real-world devices and enhancing scalability. Several works [23, 37] apply static analysis to the Linux kernel source code to improve fuzzing campaigns. In comparison, EL3XIR does not analyze the closed-source secure monitor (i.e., the target) itself but rather its open-source consumers.

EL3XIR derives function prototypes by analyzing the consumer of the SMC interface (REE OS driver) which is similar to work addressing automated test driver generation from library source code [11, 29, 35]. Moreover, there exist approaches that try to derive interfaces from closed-source binaries [20, 39]. As described in Section 7, we believe that EL3XIR might benefit from statically analyzing additional

closed-source consumers (e.g., TEE OS and bootloader) to uncover even more interfaces of runtime services. However, it is worth noting that even by solely analyzing the REE OS, EL3XIR can already identify the majority of function signatures, leaving the additional tasks for future research.

9 Conclusions

EL3XIR presents the first end-to-end rehosting and fuzzing framework targeting proprietary secure monitor implementations. The secure monitor is part of the TEE and runs on the highest privilege level on ARM-based devices. Because its complex interface is directly exposed to untrusted components (REE OS at EL1), security-critical bugs pose a single point of failure and can annihilate the entire platform's security.

Our evaluation covered seven different platforms, and EL3XIR found 17 security-critical bugs. We systematically evaluated EL3XIR with experiments providing evidence that our interface-awareness approach and peripheral modeling contribute during fuzzing to explored code coverage and the number of bugs found. In contrast, naive fuzzing approaches are ineffective for fuzzing secure monitors.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback and comments. Additionally, we would like to thank our shepherd for guiding us during the major revision process. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_186974, and DARPA HR001119S0089-AMP-FP-034.

Availability

EL3XIR's source code and artifacts are openly accessible at https://github.com/HexHive/EL3XIR.

References

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

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

[3] Google Android Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.htm, 2023.

[4] A. Adamski and M. Peterlin. Huawei Secure Monitor Vulnerabilities, 2022. https://blog.impalabs.com/2212_advisory_huawei-secure-monitor.html.

[5] ARM. SMC Calling Convention for ARMv8-A. https://documentation-service.arm.com/static/5f8ea482f86e16515cdbe3c6?token=.

[6] ARM. Trusted Firmware-A - Reference Secure Firmware for Armv7-A, Armv8-A and Armv9-A systems. https://developer.arm.com/tools-and-software/open-source-software/firmware/trusted-firmware/trusted-firmware-a.

[7] ARM. ARM Firmware Update, 2023. https://trustedfirmware-a.readthedocs.io/en/latest/components/firmware-update.html.

[8] ARM. ARM Power State Coordination Interface, 2023. https://developer.arm.com/Architectures/Power%20State%20Coordination%20Interface.

[9] ARM. ARM SiP Services, 2023. https://trustedfirmware-a.readthedocs.io/en/latest/components/arm-sip-service.html.

[10] A. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proc. of the ACM CCS*, 2014.

[11] D. Babic, S. Bucur, Y. Chen, F. Ivancic, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. FUDGE: Fuzz Driver Generation at Scale. In *Proc. of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[12] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing Program Input Grammars. *ACM SIGPLAN Notices*, 2017.

[13] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proc. of the USENIX Annual Technical Conference*, 2005.

[14] G. Beniamini. FuzzZone, 2016. https://github.com/laginimaineb/fuzz_zone/tree/master/FuzzZone.

[15] M. Busch and K. Dirsch. Finding 1-day Vulnerabilities in Trusted Applications using Selective Symbolic Execution. In *Workshop on Binary Analysis Research (BAR)*, 2020.

[16] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer. TEEzz: Fuzzing Trusted

Applications on COTS Android Devices. In *IEEE Symposium on Security and Privacy*, 2023.

[17] M. Busch, J. Westphal, and T. Müller. Unearthing the TrustedCore: A Critical Review on Huawei's Trusted Execution Environment. In *14th USENIX Workshop on Offensive Technologies*, 2020.

[18] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *IEEE Symposium on Security and Privacy*, 2020.

[19] W. Chen, Y. Wang, Z. Zhang, and Z. Qian. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *Proc. of the ACM CCS*, 2021.

[20] W. Chen, Y. Wang, Z. Zhang, and Z. Qian. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *Proc. of the ACM CCS*, 2021.

[21] A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium*, 2020.

[22] P. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol Specification Extraction. In *30th IEEE Symposium on Security and Privacy*, 2009.

[23] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. Difuze: Interface Aware Fuzzing for Kernel Drivers. In *Proc. of the ACM CCS*, 2017.

[24] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proc. of the ASIA CCS*, 2021.

[25] B. Feng, A. Mera, and L. Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium*, 2020.

[26] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proc. of the 14th USENIX Conference on Offensive Technologies*, 2020.

[27] F. Fleischer, M. Busch, and P. Kuhrt. Memory Corruption Attacks within Android TEEs: A Case Study Based on OP-TEE. In *15th International Conference on Availability, Reliability and Security*, 2020.

[28] Google. DRM, 2022. https://source.android.com/devices/drm.

[29] H. Green and T. Avgerinos. GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. In *Proc. of the 44th International Conference on Software Engineering*, 2022.

[30] N. Group. ProjectTriforce, 2016. https://github.com/nccgroup/TriforceAFL.

[31] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In *Proc. of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017.

[32] Lee H., Hayawardh V., Rohan P., Koushik S., and Michael G. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *29th USENIX Security Symposium*, 2020.

[33] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, and K. Butler. FIRMWIRE: Transparent Dynamic Analysis for Cellular Baseband Firmware. In *Proc. of the NDSS*, 2022.

[34] F. Hofhammer, M. Busch, Q. Wang, M. Egele, and M. Payer. SURGEON: Performant, Flexible and Accurate Re-Hosting via Transplantation. In *Workshop on Binary Analysis Research (BAR)*, 2024.

[35] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium*, 2020.

[36] J. Jang, S. Kong, M. Kim, D. Kim, and B. Kang. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proc. of the NDSS*, 2015.

[37] D. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *IEEE Symposium on Security and Privacy*, 2019.

[38] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *30th USENIX Security Symposium*, 2021.

[39] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim. WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Proc. of the 28th NDSS*, 2021.

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

[41] K. Lu and H. Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proc. of the ACM CCS*, 2019.

[42] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. Ryn Choe, C. Kruegel, and G. Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proc. of the NDSS*, 2017.

[43] F. Menarini and M. Bogaard. Bug Hunting S21's 10ADAB1E FW. OffensiveCon22, 2022.

[44] A. Mera, B. Feng, L. Lu, and E. Kirda. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In *42nd IEEE Symposium on Security and Privacy*, 2021.

[45] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti. Avatar2: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[46] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. TrustZone Explained: Architectural Features and Use Cases. In *Proc. of the Collaboration and Internet Computing*, 2016.

[47] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu. V-SHUTTLE: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *Proc. of the ACM CCS*, 2021.

[48] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.*, 2019.

[49] M. Pirker and D. Slamanig. A Framework for Privacy-Preserving Mobile Payment on Security Enhanced ARM TrustZone Platforms. In *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 2012.

[50] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium*, 2022.

[51] F. Schwarz. TrustedGateway: TEE-Assisted Routing and Firewall Enforcement Using ARM TrustZone. In *25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022.

[52] A. Shakevsky, E. Ronen, and A. Wool. Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design. In *31st USENIX Security Symposium*, 2022.

[53] D. Suciu, S. McLaughlin, L. Simon, and R. Sion. Horizontal Privilege Escalation in Trusted Applications. In *29th USENIX Security Symposium*, 2020.

[54] Y. Sui and J. Xue. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proc. of the 25th international conference on compiler construction*, 2016.

[55] H. Sun, K. Sun, Y. Wang, and J. Jing. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proc. of the 22nd ACM CCS*, 2015.

[56] Guanxing W. EL3 Tour: Get The Ultimate Privilege of Android Phone, 2021. https://github.com/hhj4ck/EL3Tour.

[57] C. Wright, W. Moeglein, S. Bagchi, M. Kulkarni, and A. Clements. Challenges in Firmware Re-Hosting, Emulation, and Analysis. *ACM Computing Surveys*, 2021.

[58] W. Zhou, L. Guan, P. Liu, and Y. Zhang. Automatic Firmware Emulation through Invalidity-Guided Knowledge Inference. In *30th USENIX Security Symposium*, 2021.

A Appendix

Example for MMIO Coverage Wall. The problem of coverage walls caused by missing MMIO behavior is illustrated in Listing 1. Each `mmio_read_32` function call to memory regions associated with the underlying crypto engine (e.g., `CRYPTO_CTRL_REG`) will not yield a meaningful value without a proper peripheral model behind the MMIO registers. Consequently, a naive fuzzing approach would never execute the exemplary `sip_encryption` function beyond interactions with the crypto engine, as read operations from `CRYPTO_CTRL_REG` would never return a valid status code to break out of the loop.

```
// parameters are passed via registers x1 - x4
int sip_encryption(u32 src_addr, u32 src_size, u32
    dst_addr, u32 dst_size) {

 // parsing and validation logic
 if (dst_size == NULL || src_size == NULL)
 return SIP_SMC_STATUS_REJECTED;

 if (src_size % 4 != 0, dst_size % 4 != 0)
 return SIP_SMC_STATUS_REJECTED;

 // verify that source and destination memory is in
      normal world
 if (!addr_in_nw_range(src_addr, src_size) ||
 !addr_in_nw_range(dst_addr, dst_size))
 return SIP_SMC_STATUS_REJECTED;

 for (u32 i = 0; i < dst_size; i += 4) {
  // write payload chunk to mmio data register
  mmio_write_32(CRYPTO_DATA_REG, *(src_addr + i));
```

```

  // activate crypto engine with exemplary status
      code
  mmio_write_32(CRYPTO_CTRL_REG, 0x1);

  // wait for crypto engine to finish or fail
  do {
   u32 status = mmio_read_32(MMIO_CTRL_REG);
   if(status & 0x2) { break; }
   else if (status & 0x1) { continue; }
   else { return SIP_SMC_STATUS_REJECTED; }

  } while (1);

  // read encrypted payload chunk
  *(dst_addr + i) = mmio_read_32(CRYPTO_DATA_REG);
 }

 return SIP_SMC_STATUS_OK;
}
```

Listing 1: Indicative encryption function of a silicon provider runtime service using a cryptographic engine accessed via MMIO.

B Appendix

Example for Successfully Recovered Interface. EL3XIR's recovers the SMC interface from the REE OS source code. Our system localizes SMC call site candidates and applies a backward-directed value-flow analysis for each parameter of the interface. Listing 2 shows a code snippet from the Linux kernel of the Stratix 10 SoC FPGA by Intel. EL3XIR identifies line 24 as a potential SMC callsite and follows the value-flow of each parameter `a0 - a7` backwards. EL3XIR recovers both scalars and data types for each parameter (where possible) by scanning for `store`, `load`, and `gep` LLVM instructions in the value-flow graph.

```
1  ...
2  struct stratix10_svc_data {
3   struct stratix10_svc_chan *chan;
4   phys_addr_t paddr;
5   size_t size;
6   phys_addr_t paddr_output;
7   size_t size_output;
8   u32 command;
9   u32 flag;
10  u64 arg[6];
11 };
12 ...
13 struct stratix10_svc_data *pdata = NULL;
14 pdata = kmalloc(sizeof(*pdata), GFP_KERNEL);
15 ...
16 case COMMAND_FCS_DATA_ENCRYPTION:
17  a0 = INTEL_SIP_SMC_FCS_CRYPTION;
18  a1 = 1;
19  a2 = (unsigned long)pdata->paddr;
20  a3 = (unsigned long)pdata->size;
21  a4 = (unsigned long)pdata->paddr_output;
22  a5 = (unsigned long)pdata->size_output;
23 ...
24 arm_smccc_smc(a0, a1, a2, a3, a4, a5, a6, a7);
```

Listing 2: Code snippet from the Linux kernel setting up and executing an SMC call. The a0 parameter is the function ID used to define the requested runtime service.

The successfully recovered interface is illustrated in Listing 3. EL3XIR recovers the function identifier for the `x0` register (`INTEL_SIP_SMC_FCS_CRYPTION = 0x4200005B`) and type information or constants for the remaining parameters (e.g., `a2` holds a physical address and `a1` can hold the constant `0x1`). The interface is saved in a `CVS` file and used to refine EL3XIR's harness.

```
1 0, unsigned long, 0x4200005B
2 1, u64, 0x1
3 2, phys_addr_t,
4 3, size_t,
5 4, phys_addr_t,
6 5, size_t,
```

Listing 3: Recovered interface by EL3XIR with format: `register idx, C-type, constant (optional).`