Journal of Hardware and Systems Security (2023) 7:44–54
https://doi.org/10.1007/s41635-023-00133-3

CROWBAR: Natively Fuzzing Trusted Applications Using ARM CoreSight

Haoqi Shan[1,2] · Moyao Huang[1] · Yujia Liu[3] · Sravani Nissankararao[4] · Yier Jin[1] · Shuo Wang[1] · Dean Sullivan[4]

Received: 31 January 2023 / Accepted: 3 May 2023 / Published online: 15 June 2023

Abstract
Trusted execution environments (TEE) are deployed on many platforms to provide both confidentiality and integrity, and their extensive use offers a secure environment for privacy-sensitive operations. Despite TEE prevalence in the smartphone and tablet market, vulnerability research into TEE security is relatively rare. This is, in part, due to the strong isolation guarantees provided by its implementation. In this paper, we propose a hardware assisted fuzzing framework, CROWBAR, that bypasses TEE isolation to natively evaluate trusted applications (TAs) on mobile devices by leveraging ARM CoreSight components. CROWBAR performs feedback-driven fuzzing on commercial, closed source TAs while running in a TEE protected environment. We implement CROWBAR on 2 prototype commercial-off-the-shelf (COTS) smartphones and one development board, finding 3 unique crashes in 5 closed source TAs that are previously unreported in the TrustZone fuzzing literature.

Keywords Trusted execution environments · ARM CoreSight · Trusted applications · Fuzzing

1 Introduction

Fuzzing ARM TrustZone software is imperative especially given the number of devices in the market whose security relies on trusted execution environments (TEE). This popularity is, in part, due to the confidentiality and integrity guarantees provided by TrustZone of which many fundamental applications employ to securely provision and deploy critical resources. It is surprising, therefore, how few are the number of works investigating TrustZone fuzzing despite its popularity and extensive use in mobile and notebook environments. The scarcity, however, is not without a warrant as TEEs are built upon strong isolation guarantees. Major vendors also ensure their TEEs are locked down by the time they hit the market and release little information regarding their SoC implementation. ARM's business model further complicates the matter. Its inventory is built mostly upon soft processor core IP that ARM licenses to vendors with extensive functionality that accounts for nearly 100 unique products [1] each of which can be configured with a rich variety of features.

The two most relevant works on fuzzing TrustZone employ rehosting [2] and blackbox fuzzing [3] techniques, and each is a significant achievement. Harrison et al. developed PartEMU [4], which is the first design for rehosting via emulation of a whole system TEE capable of dynamically analyzing several closed-source TrustZone Operating Systems (TZOSes). In so doing, they were able to perform a large-scale study of trusted applications (TAs) from 13 vendors, thereby unveiling several unknown bugs. Busch et al. recently proposed TEEzz [5], a blackbox fuzzing framework

✉ Dean Sullivan
dean.sullivan@unh.edu

Haoqi Shan
haoqi.shan@ufl.edu

Moyao Huang
moyaohuang@ufl.edu

Yujia Liu
liuyujia1@lixiang.com

Sravani Nissankararao
sravani.nissankararao@unh.edu

Yier Jin
yier.jin@ieee.org

Shuo Wang
shuo.wang@ece.ufl.edu

1 Department of Electrical and Computer Engineering, University of Florida, Gainesville, USA

2 CertiK, New York, USA

3 Li Auto Inc., Beijing, China

4 Department of Electrical and Computer Engineering, University of New Hampshire, Durham, USA

capable of natively fuzzing TAs on COTS devices by leveraging observable information from the normal world. Their approach resulted in the discovery of 40 unique bugs across several COTS vendors.

Despite their achievements, both PartEMU and TEEzz have drawbacks. PartEMU is built using propriety knowledge that is closed-source, hence difficult to recreate. It also naturally suffers from slowdowns caused by emulation and potential inaccuracies in implementation. TEEzz is innately limited from true introspection due to its blackbox nature. This makes coverage and crash analysis difficult because of the limited information recoverable from the normal/secure world interface. Crashes can typically only be triaged based on a return code or system reboot using blackbox techniques. In practice, however, TEEzz circumvented this by obtaining stack traces via a normal world-accessible TEE log interface. TEEzz approaches nearly ideal behavior in its ability to natively evaluate TAs in their host environment. Our approach builds upon this feature while circumventing introspection restrictions of black box fuzzing techniques using the trace features of ARM CoreSight [6].

In our paper, we demonstrate the feasibility of fuzzing TAs natively using ARM CoreSight-enabled smartphones in a framework called CROWBAR. This overcomes the limitations of blackbox fuzzing by allowing introspection of executing TAs control flow in a native environment, thereby enhancing feedback-driven fuzzers, improving code coverage analysis, and allowing binary-only crash triage. We show how to enable and configure ARM CoreSight for trace-assisted fuzzing on prototype devices, integrate and evaluate trace information using AFL++ [7], a popular control-flow-driven mutational fuzzer, and perform the native evaluation on COTS devices running Trusty and QSEE. In summary, we make the following contributions:

- We propose and evaluate the use of ARM CoreSight to perform coverage-driven fuzzing on TrustZone-protected applications code with prototype smartphones to achieve native introspection.
- We implement CROWBAR, outlining in detail the process by which CoreSight features are discovered, enabled, and configured on prototype smartphones.
- We demonstrate CoreSight-assisted trace integration and evaluation by fuzzing TAs from multiple vendors, i.e., Google Trusty and Qualcomm QSEE natively. We evaluate 5 TAs and find 3 unique crashes.
- We provide a detailed discussion of the challenges faced in using CoreSight-assisted fuzzing across a representative range of device vendors.

The remainder of the paper first addresses background in Section 2. We then introduce the design of CROWBAR in Section 3 before detailing its implementation in Section 4. We then evaluate CROWBAR in Section 5 and present an analysis of challenges in Section 6. We then present related works in Section 7 and conclude in Section 8.

2 Background

In this section, we present the necessary technical background and terminology used throughout the paper. We describe the CoreSight Architecture, CoreSight tracing, and TrustZone architecture.

2.1 ARM CoreSight Infrastructure

The ARM CoreSight infrastructure defines a set of standard interfaces and components to assist ARM-based SoC manufacturers to debug their software and hardware. Specifically, ARM supplies a series of IP components to implement SoC-level debug and trace functionalities, such as the CoreSight SoC-400 library for Cortex-A class designs [8]. The CoreSight SoC library provides a comprehensive set of trace macrocells including the Embedded Trace Macrocell (ETM), Program Trace Macrocell (PTM), and Embedded Cross Trigger (ECT) among others. In addition, ARM also licenses the Trace Memory Controller (TMC) to configure where traces are stored. In our paper, we aim to provide a fuzzing framework for ARM Cortex-A-based commercial devices which are equipped with the CoreSight SoC-400 library. These components can be accessed either through a standard JTAG interface (off-chip access) or can be accessed via the memory bus (on-chip access).

We access and configure the CoreSight components via memory-mapped I/O, as shown in Fig. 1. We use the Embedded Trace Macrocell and Program Trace Macrocells provided by the CoreSight SoC-400 to collect traces

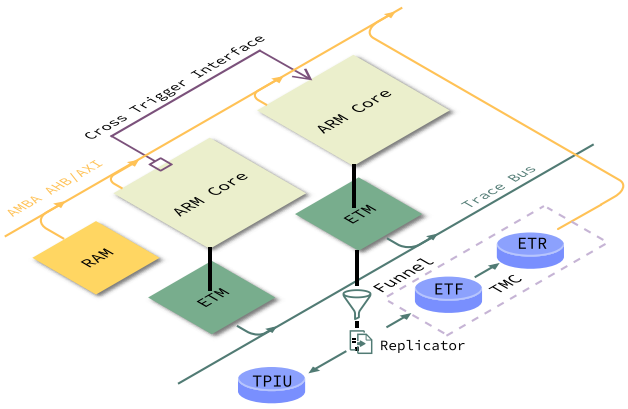

Fig. 1 ARM CoreSight Architecture that illustrates how the ARM Core, RAM, ETM, AMBA AXI, and other CoreSight components are interconnected

while TAs execute. We use the Embedded Cross Trigger to actively invoke TAs to avoid the cost of world switching. We also use the Trace Memory Controller to control where the generated trace is stored. Detailed information on each component is described below.

Embedded Trace Macrocell The ETM is used as a non-invasive debug method on ARM-based SoCs to enable developers to collect the instruction flow of a target application while running without affecting the performance of the CPU. The ETM is tapped on the instruction and data bus to monitor the current instruction and data being manipulated. It can be programmed via multiple interfaces, including system registers, memory-mapped I/O, or via an external debugger interface. It is worth noting that ARMv8-A, the most popular ARM architecture used by COTS devices, does not support data tracing if ETMv4 [6] is used.

Embedded Cross Trigger The ECT sub-system is used to pass debug events from one debug component to another within the CPU complex. In other words, a processor can pass debug information to another and thus perform invasive debugging between processors on the same die using the ECT. The ECT comprises the Cross Trigger Interface, Cross Trigger Matrix, and Event Asynchronous Bridge.

Trace Memory Controller The TMC serves as the final trace component by terminating the trace bus into dedicated on-chip SRAM. The TMC allows developers to choose to store the trace via the embedded trace buffer (ETB) or embedded trace FIFO (ETF). The embedded trace router (ETR) is also often used to route the trace into system memory via the Advanced eXtensible Interface (AXI) bus due to the limited memory size of on-chip SRAM.

2.2 ARM TrustZone and TEE

ARM TrustZone is a hardware-based security extension that offers hardware-enforced isolation for ARM-based CPUs. TrustZone technology is integrated into ARMv7-A and ARMv8-A architecture-based processors and occupies the majority of the mobile devices today. TrustZone is the hardware foundation of a TEE on ARM-based processors. Together, they act as a firewall to enforce access control to secure both peripheral and memory regions used and executed upon by TAs. The isolation separates two execution environments: the secure world and the normal world. The ARM TrustZone architecture design is illustrated in Fig. 2.

Secure World The secure world hosts the TEE and runs trusted code. The trusted code can be developed by ARM, SoC manufacturers, or trusted third-party developers. The secure world guarantees that the code and data that runs in the TEE are protected with respect to confidentiality and integrity [9].

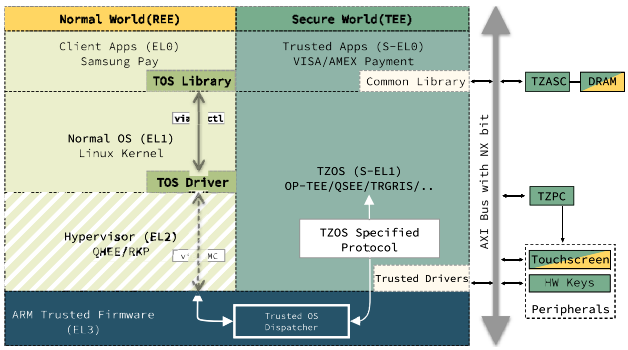

Fig. 2 ARM TrustZone Architecture on ARMv8

Normal World The normal world hosts the Rich OS Execution Environment (REE) and runs non-secure world code. The REE can be used to run a regular Linux OS or Android OS and is considered untrusted by the TEE. The REE resides outside the isolation boundary of the TEE and thus cannot access its peripherals and protected memory.

TrustZone Implementation ARM uses the non-secure (NS) bit in the Secure Configuration Register (SCR) to represent the current security state of the processor. The main system AXI bus propagates the NS bit to indicate whether an access terminates in the secure or non-secure region. The processor switches between the normal world and the secure world (world switching) by issuing a Secure Monitor Call (SMC). The SMC is handled by the Secure Monitor in Exception-Level 3 (EL3). Because the SMC cannot be issued within the non-privileged mode, the Rich OS often exposes an SMC call as a device driver interface for user privilege applications. SELinux is used to restrict applications that request switching into the secure world in order to use secure-world services. In addition, software that runs in a secure world can access both secure and non-secure memory and peripherals, while software running in the normal world can only access non-secure memory and peripherals.

3 On Device Fuzzing—in the Secure World

In this section, we introduce the technical challenges in fuzzing TAs and how to achieve such goals on prototype devices, including popular smartphones and tablets.

3.1 Feasibility of Trace Collection in Secure World

The current challenge in fuzzing TAs mainly lies in the fact that the trusted world is not intended to be monitored

by untrusted third parties. The problem is exacerbated further because TAs that are running on most commercial devices are not open source. The main accessible way for security researchers to find vulnerabilities in TAs is through reverse engineering which involves tremendous human effort. However, as was mentioned in Section 2, CoreSight is designed to help developers collect raw execution traces regardless of the world (secure/non-secure) in which the CPU is running. In other words, if CoreSight components are both equipped and enabled on the device hosting the trusted application, it is possible to collect the trusted application execution trace.

With that said, to protect the TrustZone design, ARM implements a mechanism that allows the vendor to decide whether CoreSight is functional while the CPU runs in the secure world via a fuse setup. In 2019, Ning and Zhang [10] demonstrated that certain COTS devices from different vendors have CoreSight components enabled. This was used to recover a fingerprinted image from the secure world. And, even though this attack is not directly related to fuzzing TAs, it still betrays the possibility of collecting runtime traces with CoreSight in the trusted world on COTS devices. After the publication of this paper, many vendors fixed the issue outright by simply disabling CoreSight. We have verified this approach on several newly released smartphones from different vendors, see Table 1.

3.2 Enabling CROWBAR with Prototype Devices

However, there is one exception that we have found. Although CoreSight is most likely disabled in production devices, it is likely still enabled in the prototypes for those devices. Prototype devices are typically used for the development and validation of the hardware design prior to the massive production stage. The prototype devices are not for sale and are only used by the vendor's internal team. They generally remain strictly controlled until the release of actual devices. After the release of production devices, these prototype devices are either destroyed or sold to third parties such as rare phone collectors or used markets. We also encountered cases wherein certain vendors allow their employees to purchase these prototype devices.

In many cases, the prototypes that are available on the used market have nearly identical hardware designs as the production stage devices. Even the stock firmware can be used by these prototypes, excepting features requiring hardware key provisioning using development keys such as fingerprint unlocking or secure payments. Otherwise, they can be used normally, but are popularly employed in security research due to the lack of restrictions compared to their COTS counterparts. For example, iPhone prototypes are very popular in the Apple jailbreak community [11].

To verify if we can use these prototypes to conduct fuzzing on TAs with unlocked CoreSight components, we purchased 9 prototype devices spanning 7 vendors from both Ebay and other online used markets. The collected devices are listed in Table 1. We further compiled and loaded kernel modules to access the CoreSight authentication registers to verify if it can be used to perform either invasive or noninvasive secure world debugging. We found that many support secure world debugging in either scenario.

We develop CROWBAR based on the use of prototype devices and their exposure of ARM CoreSight features as illustrated in Fig. 3. CROWBAR consists of 4 parts: a fuzzing framework that handles TA executions; a state manager that collaborates with the fuzzer to track and maintain the

Table 1 Prototype smartphones and tablets. Note that all tested models deploy the ARMv8 architecture except the Redmi 6 device, which deploys the ARMv7 architecture

Model	SoC	Batch	TZOS	SPNIDEN	SPIDEN	DBGEN	NIDEN	Coresight
Pixel 2	MSM8998	MP	QSEE	✓	✓	✓	✓	✓
Pixel 3XL	SDM845	PT	QSEE	✓	✓	✓	✓	✓
Pixel 4	SM8150	PT	QSEE	✓	✓	✓	✓	✓
Pixel 4	SM8150	MP	QSEE	✗	✗	✗	✗	✗
Samsung S10+	Exynos 9820	PT	TEEGRIS	✗	✗	✗	✗	✗
Huawei P9	Kirin 955	MP	TrustedCore	✗	✗	✓	✗	✗
Huawei Y Max	SDM660	PT	QSEE	✓	✓	✓	✓	✗
Honor Note 10	Kirin 970	PT	TrustCore	✗	✗	✗	✗	✗
Moto One Power	SDM636	PT	QSEE	✓	✓	✓	✓	✗
Moto G8 Power	SM6125	PT	QSEE	✓	✓	✓	✓	✗
Xiaomi A2	SDM660	PT	QSEE	✓	✓	✓	✓	✓
Redmi 6	MT6762	PT	OP-TEE	✓	✓	✓	✓	✗

PT ProtoType, *MP* Massive Production

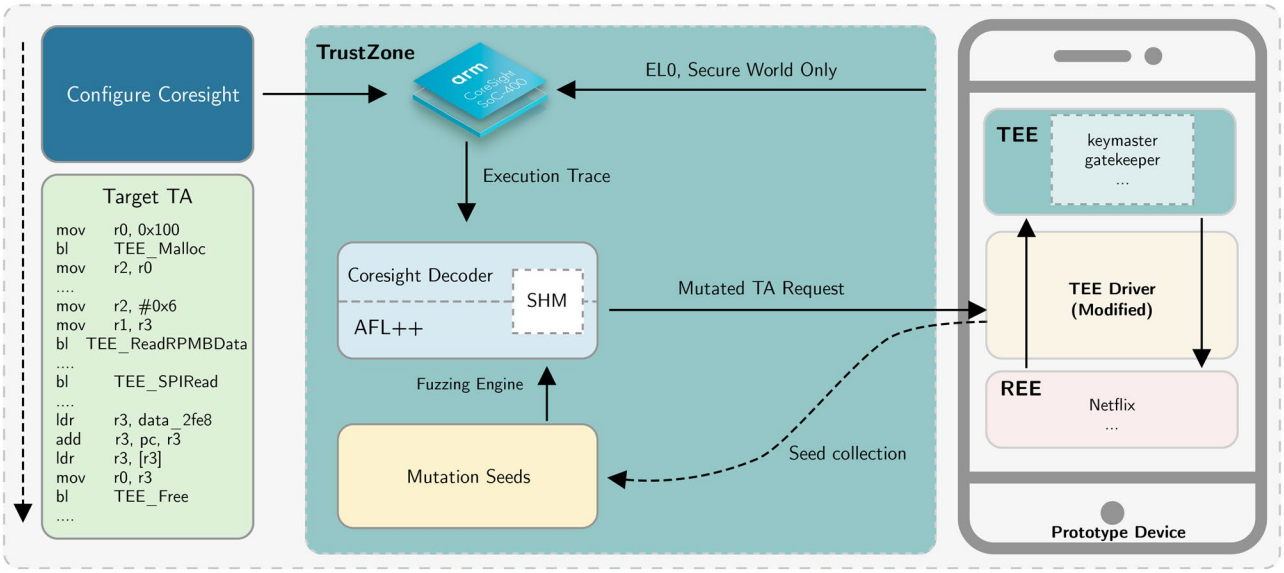

Fig. 3 CROWBAR design on prototype device

state of TAs during fuzzing; CoreSight discovery and configuration that is used to collect the native execution trace of TAs to perform effective coverage-guided fuzzing; and a trace integrator responsible for reconstructing TA control based on CoreSight supplied packets.

4 CROWBAR Implementation on Prototype Devices

In this section, we first describe how to verify if the prototype device has debugging enabled and provide details on configuring CoreSight per prototype.

4.1 Debug Authentication

Verification of CoreSight components is achieved by reading the debug authentication signal from the authentication status system register `DBGAUTHSTATUS_EL1`, in which the invasive/non-invasive debugging in normal/secure world enable bits can be inferred. Specifically, there are four authentication signals defined: (1) `DBGEN`; (2) `NIDEN`; (3) `SPIDEN`; and (4) `SPNIDEN`. In general, they can be interpreted as signals that control if the corresponding debug feature can be configured in each world, as shown in Table 2. The `DEVICEEN` bit must be enabled for any of the four authentication signals to be enabled. Furthermore, vendors may use the eFuse to hard wire them so that the debug features can be leveraged during development and locked down in production. Although this is an approach for vendors to have more granular control over debug feature availability, ARM does not prevent the SoC manufacturer from implementing extra protection to secure CoreSight as we found and will discuss in Section 6.

Access to the debug authentication register is privileged, so we opted to read the register on each individual prototype device using a kernel module. Table 1 shows that the majority of prototype devices have all the authentication signals asserted. Interestingly, on certain massive production stage devices, such as the Pixel 2, the authentication signal is asserted as well which allows us to use CoreSight in both worlds. Comparatively, newer massive production-stage devices, such as the Pixel 4, are correctly fused to disable CoreSight.

4.2 CoreSight Configuration

There are several prerequisites that have to first be met to be able to configure the CoreSight components on prototype devices for coverage-driven feedback: (1) the prototype devices have to be rooted; (2) the vendors have to release the kernel source code; (3) the CoreSight component topology has to be known; and (4) the CoreSight component clock domain has to be enabled.

Table 2 Debug Authentication Signals

Signal	Description	World
`DBGEN`	Invasive debug enable	Normal world
`NIDEN`	Noninvasive debug enable	Normal world
`SPIDEN`	Invasive debug enable	Secure world
`SPNIDEN`	Noninvasive debug enable	Secure world

Rooted Device First, the configuration of CoreSight components is privileged and can only be performed by the root user, which means the bootloader of the prototype devices should be unlocked so that we can load our customized kernel. Note that, even though the devices we purchased are prototype devices, we found in many cases the sellers flashed the stock firmware and subsequently locked the phone to be able to use them as daily phones.

Kernel Source To make the prototype devices functional and successfully boot, we need the kernel source code from vendors to configure all the hardware properly. This is challenging even though the Linux kernel uses the GPL license because certain vendors choose not to release the kernel source used in their firmware. In addition, the kernel version should be consistent with the stock ROM running on our device. Typically, vendors release old source code and never update it.

CoreSight Topology Access to the kernel source code from the vendor allows us to configure CoreSight properly for the specific prototype device. Albeit CoreSight is licensed by ARM as IP, the manufacturer is free to customize its integration into their SoC. CoreSight IP is highly configurable which allows the manufacturer to choose, for example, the amount of ETR, ETB, and TPIU, how these are connected to one another, and the physical memory address of these components. All of this information needs to be detected. We can reuse the released kernel code to find the corresponding device tree file, if there is any; however, this is not always available. Instead, our implementation first reads the ROM table base address from the system register via our kernel module and dynamically enumerates all available CoreSight components present on the SoC.

CoreSight Clock Domain On certain devices, such as the Pixel 2, the clock domain of the CoreSight component is disabled by default which results in a bus error fault when we try to access CoreSight via memory-mapped registers. To successfully configure CoreSight, the debug clock domain has to be configured and enabled via its device tree. We use the released kernel source code to find the corresponding debug clock domain and enable ite

4.3 Fuzzing Framework

In Sections 4.1 and 4.2, we discussed the prerequisites to configure CoreSight on different prototype device platforms. We now show how to effectively fuzz TAs given access to CoreSight. We use QSEE as an example to explain our methodology as all of the prototype smartphones in which we can capture TrustZone execution traces are equipped with Qualcomm CPUs. QSEE is implemented by Qualcomm as the trusted execution environment for their SoC designs.

As illustrated in Fig. 4, the TAs are invoked by client applications in the normal world. Such a request traverses multiple layers of privilege levels across the normal and secure world. To request a corresponding trusted application, the client application first issues a `QSEECom_send_command` call exposed by the QSEE normal world common library. The common library then interacts with a QSEE normal world kernel driver (`QSEECom Driver`) using an `ioctl` call. The `QSEECom Driver` further invokes the `Secure Channel Monitor (SCM) Driver` to issue a secure monitor call to request a world switch and transfer control and command to the trusted world. The QSEE secure world operating system then dispatches the request to the corresponding trusted application and passes the request details as an event. When the trusted application finishes the command execution and copies the result into a shared response buffer, control flow is then reversed and the client application can then retrieve the result from the preallocated shared memory region.

To effectively mimic this process and fuzz the trusted application, we leverage our capability of customizing a kernel module and rewrite the `QSEECom Driver` to record all regular commands and associated memory requests from client applications. To do so, we rewrite the `QSEECOM_IOCTL_SEND_CMD_REQ` `ioctl` handler and extract the useful information from a request memory pointer parameter including the command ID, request data, and request data length. We then store the request details as a seed for our fuzzing process. This allows us to record both the command ID and request buffer content for each invoked trusted application.

Once we have the seed in place to properly initialize the fuzzing campaign, we can configure the prototype devices to

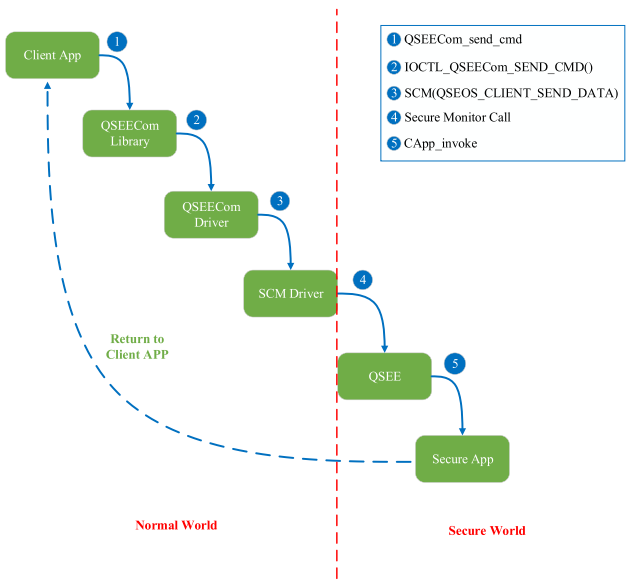

Fig. 4 QSEE end-to-end call flow

capture the trusted application execution trace as feedback for the fuzzing process. We first configure the correct CoreSight topology using the `CSAL` module from ARM [12] with the inferred topology using either the device or topology discovery process described in Section 4.2. We then find and connect the CPU core which runs the TA and configure it as a trace source tapped to the CoreSight funnel. The funnel component is then either connected to a replicator component so the trace can be duplicated and fed into a different sink, or directly into on-chip SRAM memory using the trace buffer. In addition to the CoreSight interconnection configuration, we also need to configure the ETM to only capture the trace of TAs. The filter is applied by setting up the ETM to only activate with a specific world and at a specific privilege level, i.e., secure world and EL0. Note that per specification, the ETM is able to filter the trace using a specific process ID; however, we found that almost all TZOSes do not properly configure the context ID register (CIDR) of the CPU core which results in the ETM context ID value filter being unusable.

After CoreSight is configured to capture the trusted application runtime trace on a specific CPU core, we then instruct our fuzzer, AFL++ [7], to generate the mutated input based on the seed input previously collected. Instead of directly fuzzing the target application with AFL++, we implement a wrapper program to take the mutated input and pack it into the normal request format so it can be passed to the `QSEECom Driver`. This wrapper program is also pinned to the same CPU core that we configured to host the trusted application. In addition, the wrapper program takes care of the fuzzing context setup, such as initializing the trusted application if it is not loaded in the trusted world and examining the shared buffer allocated by the `QSEECom Driver`. The wrapper program evaluates the return value within the response buffer to determine if the previous command execution was successful. A failed attempt is then recorded and later analyzed. In addition, the kernel message is examined to determine if the failed attempt is caused by an invalid command ID or due to a crash in the trusted application. The crash status is sent back to AFL++. Finally, the `QSEECom Driver` is modified to block requests that invoke non-fuzzed TAs to eliminate tracing interference. After the completion of the wrapper program, the program execution trace is dumped from the CoreSight on-device trace buffer and parsed to reconstruct the trusted application's control flow.

4.4 Trace Integration

To reconstruct the TA control flow, the trace data needs to be extracted from the CoreSight component first and parsed. CoreSight is normally exposed to the memory bus so that developers can configure it through memory-mapped registers and perform on device tracing and debugging. Developers can utilize two ways to read/write these memory-mapped registers: via `sysfs` or `/dev/mem` on Linux. If the Linux kernel has the correct CoreSight kernel modules and they are correctly configured during compilation, a set of `sysfs` interfaces are exposed under `/sys/bus/coresight`. However, this is not always true. In many cases, the kernel for the test device may not have any CoreSight drivers provided. To achieve the maximum flexibility, we use `/dev/mem` as a way to access the CoreSight registers as long as we know where CoreSight is mapped into physical memory and how the CoreSight component topology is connected. We use the `CSAL` module to assist the CoreSight configuration by directly reading/writing to `/dev/mem`.

In our scenario, we mainly use the ETM from CoreSight to collect instruction traces from a specific ARM CPU core. The ETM also can be configured to filter the trace so it only generates an instruction trace when TA code runs instead of TZOS. Once we have CoreSight configured, we instruct the fork server from the fuzzer to enable CoreSight tracing and invoke the TA from normal world. We instruct the fork server to disable CoreSight tracing once the TA execution is completed in case the ETB is overflowed. The ETB is an on-chip SRAM buffer with a typical size between 4KB and 16KB. We begin the decoding process with `OpenCSD` by reading from the ETB after execution of the TA command request is completed. `OpenCSD` is an open source library developed by Linaro to perform trace data deformatting and packet decoding. The decoded packet is used to find all instruction addresses that were executed by the previous command handling. The ETM saves the generated trace using address elements that contain the address of instructions indicating their start address and instruction set. The address elements are generated in certain scenarios where the trace analyzer cannot infer the address of the current trace, such as when an indirect branch is taken, an exception is taken, or misspeculation occurs. We describe how we handle this in the subsequent section.

5 Evaluating CROWBAR: CoreSight-Assisted Fuzzing on Prototype Devices

5.1 Trace Evaluation

To better illustrate how CoreSight generates traces while the CPU is running, we execute a sample program that invokes a syscall, library call, input validation, and indirect branch operations. In Fig. 5, we list the assembly instructions and decoded CoreSight traces side-by-side. The recorded trace and corresponding assembly code show that the ETM does not generate traces for every instruction the CPU core executes. Instead, it only generates **P0** elements (elements that

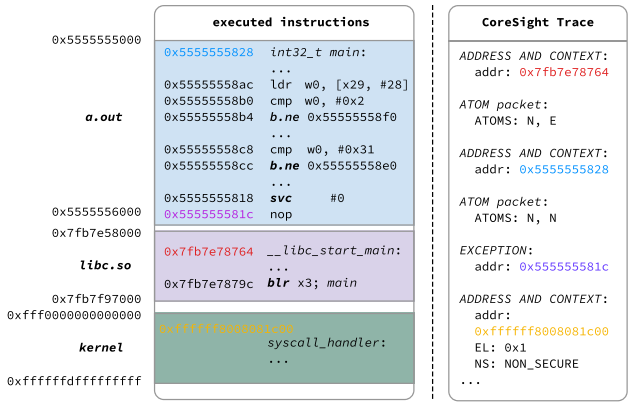

Fig. 5 Example of collected CoreSight trace

contain instruction address information) when the CPU core takes exceptions, branch instructions, or Instruction Synchronization Barrier (ISB) instructions [13]. CoreSight does not generate trace data for regular instructions that do not change the control flow. This is because it is assumed the trace analyzer has the binary code of the traced program and can reconstruct the control flow information depending on where and when the branch instruction is taken. For conditional branch instructions, the ETM generates **ATOM** elements to record whether a branch was taken or not. The combination of these two types of elements enables us to directly decode the trace and use the packet information to represent the control flow with a coarse granularity without taking time to parse the assembly code and fully recover the control flow.

In our framework, we parse the trace and find all P0 elements that directly include address values when the executed instruction address cannot be inferred. We also include the ATOM elements that contain the branch choice of conditional branch instructions. The ATOM elements always follow the P0 elements to provide extra control flow information. In practice, developers can choose to reconstruct the full control flow by parsing the assembly code and tracing binary data at the same time to find the addresses of the ATOM elements. This, however, is time-consuming and not necessary for our purposes. Instead, we first use the `MurmurHash` hash algorithm [14] to hash the address of the P0 elements so that we can map them into the shared memory region of AFL++ to generate uniform coverage distribution.

Regarding the ATOM elements themselves, due to the fact that they are nothing but an `EENE`-like string, we cannot directly hash and map them. As such, we process the conditional branch choices in an ATOM element one by one. For the first choice, we concatenate the address of the P0 element with the first choice and then hash the result to generate a unique value for this combination. The hash result is further used to concatenate with the next choice and hashed again. All the hash results are mapped into the shared memory region of AFL++ where a counter is maintained to keep track of the number of times certain addresses are visited. The shared memory region is used by AFL++ as the coverage bitmap for feedback purposes to generate the next mutated input. The pseudo code for this process can be found in listing 1 as following.

```
1
2  static vaddr_t last_addr = 0;
3  switch(packet_type) {
4      case CS_ETM_PKT_TYPE_ATOM:
5          for char in (context.atom_str) {
6              last_addr = murmur_hash(↩
                   last_addr, char);
7              shm[map(murmur_hash(last_addr)↩
                   )] += 1;
8          }
9          break;
10     case CS_ETM_PKT_TYPE_P0:
11         shm[map(murmur_hash(context.addr))↩
               ] += 1;
12         last_addr = context.addr;
13         break;
14 }
```

5.2 CoreSight-Assisted Fuzzing Evaluation

We implemented our CoreSight-assisted fuzzing on the NVIDIA TX2 development board (Trusty) and Pixel 3/4 prototype device (QSEE). We describe concrete details and fuzzing results below.

Pixel 3/4 We first recompiled the kernel for Pixel 3 and Pixel 4 to support userspace hardware memory addresses by disabling the `CONFIG_STRICT_DEVMEM` option so that we can directly interact with the memory-mapped CoreSight component using `/dev/mem`. We also modify the `QSEECom Driver` to intercept the trusted world request for fuzzing seed collection purposes. We collected roughly 100 different TA commands and associated request buffer memory content to use as seed input for AFL++. These seed inputs were then used to interact with the `keymaster` and `gatekeeper` TAs. The execution speed of these TAs is relatively slow. The fuzzing speed of `keymaster` and `gatekeeper` is around 14 iterations per second. Analysis of the slowdown revealed that most of time consumption comes from reboots caused by crashing trusted world code. This has the side-effect, however, of making it difficult to recover fuzzing context.

NVIDIA TX2 To evaluate if our framework works on other TZOS designs, we also implemented our framework on the NVIDIA TX2 development board which uses Trusty as its TZOS. Trusty is developed by Google and used on Pixel 7 [15] as the TEE. We also modify the normal world TEE driver to log the request buffer content and then invoke the client application to perform encryption/decryption/key generation in the trusted world. The

fuzzing speed on sample TAs `hwkey-agent` and `luks-srv` is around 10 iterations per second. We further modified the trusted application `hwkey-agent` to contain extra code that will crash using an Out-of-Bound memory access. Our fuzzer triggered the crash within 10 min of fuzzing without previous knowledge of the correct command ID.

Overall Evaluation We evaluated CROWBAR's feedback-driven fuzzing on Pixel 3/4 prototype devices. Our approach found 3 new and unique crashes in 5 preinstalled TAs. The analysis herein did not extend to crafting exploits from the found vulnerabilities. However, these crashes are not covered in the previous TrustZone fuzzing research based on a review of the literature. We argue this shows the feasibility of our approach, particularly its capabilities for exposing coverage information across TrustZone worlds for feedback-driven fuzzing analysis.

6 Discussion of CROWBAR Limitations

As we have shown in the previous section, there are many challenges to enable and configure CoreSight on prototype devices. This process may even render some prototype devices unusable. Here, we provide a brief summary of issues that prevent researchers from using these prototype devices.

Qualcomm SoC We found that Qualcomm SoC-based devices are the easiest to find on the used market and the most third-party researcher-friendly among all the prototypes we purchased. We were able to recover CoreSight memory address and topology using its open-source kernel code, which allowed us to successfully access and configure CoreSight on Qualcomm SDM845 and SM8150 devices and perform CoreSight-assisted fuzzing on the Pixel 3 and Pixel 4 prototype devices, respectively. We were also able to probe the debug authentication register and find the CoreSight topology dynamically. However, for the SDM660 used by Huawei Y Max, the debug clock domain was disabled, and we were unable to enable it via kernel code modifications. As such, the kernel crashed whenever CoreSight memory-mapped addresses were accessed. On certain devices, such as the SDM636 and SM6125 for Motorola, CoreSight can be configured and accessed yet the compiled display driver did not work rendering it unusable.

Samsung SoC Exynos is a powerful SoC that is made by Samsung and used on high-end Samsung smartphones. We found that the authentication signal on Samsung S10+ is all asserted. However, Samsung has a unique proprietary design [16] to add an extra layer of protection for their debug component, named Secure JTAG. The Secure JTAG uses the eFuse to protect the debug component with a password hash. The Secure JTAG further controls the debug authentication signals which prevent unauthorized access to the debug component even on prototype devices, regardless of whether hardware JTAG or memory-mapped registers are employed.

Huawei SoC Kirin is Huawei's SoC design used on their high-end smartphones. As we mentioned before, access to CoreSight components requires privileges in order to run our own kernel module. This requires the device to be unlocked and rooted. However, the Huawei Kirin-based prototype devices we purchased are all flashed with stock firmware which locks the bootloader by default. Unlocking Huawei's bootloader is non-trivial as it has not been officially supported by Huawei since 2018 [17]. Furthermore, the kernel source code released by Huawei does not contain any CoreSight memory address or topology information which makes the usage even more difficult.

ASLR in CoreSight Tracing CoreSight components are interconnected with the CPU core. As such, the trace addresses of executed instructions are virtual addresses. To mitigate memory vulnerabilities in the trusted world, vendors such as Qualcomm and Samsung have enabled Address-Space Layout Randomization (ASLR) to randomize the load address of trusted applications. This feature cannot be disabled and makes the collected execution trace inconsistent between different loading/unloading of the same TA. However, once the TA is loaded, the memory layout does not change until the TA is reloaded. To address this issue, we reverse-engineered the TA loading process of QSEE and found that QSEE always calls an initialization function, `tz_app_init`, after the trusted application is successfully loaded. This design makes the TA loading and initialization process deterministic although the recorded execution address is changed. We leverage this behavior to remove the address difference caused by ASLR by aligning our recorded execution trace with that function offset every time the fuzzed TA is reloaded. However, this approach is limited to rooted Qualcomm devices.

Scalability of CoreSight Configuration The process of inferring CoreSight component topology and subsequent configuration is time-consuming and must be done on a per-device basis. It is made more difficult due to the unavailability of the technical reference manual (TRM) of many of the SoCs used in our prototype devices. However, we found that we can successfully configure the CoreSight devices to collect the secure world trace on four different devices from two different vendors, as shown in Table 1. While we argue this is a significant achievement, scalability concerns regarding CoreSight-assisted fuzzing are still an issue if the solution is to be generic.

7 Related Work

Approaches related to TEE fuzzing can be organized into two categories based on fuzzing environment: firmware rehosting through emulation and black box fuzzing running on-device. Both of them overcome certain challenges of TrustZone fuzzing.

Rehosting-based approaches aim to provide black box software with the necessary dependencies for supervised execution. Costin et al. demonstrated the feasibility of automated vulnerability discovery in embedded devices via full system emulation [18]. Talebi et al. [19] enables dynamic analysis of Linux kernel drivers by running the target in an emulator environment and forwarding access to real hardware. However, these techniques used in REE cannot be applied to fuzz TrustZone vulnerabilities as they are dependent on software and hardware components that are not publicly documented. Besides, the manual reverse engineering effort for finding out the dependencies and interfaces is tremendous. However, Harrison et al. developed PartEMU [4], which provides necessary software and hardware dependencies for TEE vulnerabilities with minimum effort. The small selection of simulation components makes PartEMU the first and the most feasible design for rehosting a whole TEE system to an emulated environment. Despite its high fidelity in TEE fuzzing, PartEMU has three main shortcomings: (1) the low fuzzing coverage due to missing implementation of TA command interactions; (2) it does not simulate hardware root of trust and consequently fails to fuzz the TAs dependent on these proprietary hardwares; and (3) the fuzzing framework suffers major overhead due to the cross-architecture emulation.

Rehosting approaches also have some other inherent limitations. First, the program to be rehosted is not always accessible for researchers to reverse-engineer its format. For instance, pseudo-TAs of OP-TEE [20] are statically built into the OP-TEE core blob, which will take a lot of effort to analyze its code. Another scenario is when TAs are encrypted and only expose limited APIs to the REE, e.g., Huawei devices with TrustedCore. Second, the effectiveness of rehosting-based fuzzing depends on the accuracy of hardware emulation. Feng et al. proposed a novel tool, called P2IM [21], using abstracted hardware register patterns to generate hardware models automatically on-the-fly and applicable to fit diverse firmware implementations. [22] attempts using machine learning to generate hardware models. While these techniques show the feasibility of filling the gap between the emulation model and real hardware, they cannot be used directly on TA fuzzing.

Meanwhile, black box fuzzing is also studied on TrustZone fuzzing due to the nature of intransparency. Recently, Busch et al. proposed a fuzzing framework capable of effectively fuzzing TAs on COTS devices, TEEzz [5]. By leveraging observable information from REE, the fuzzer can infer the types and parameters of TAs API through their interaction to generate fuzzing templates and, therefore, achieve high-efficiency on-device fuzzing. Whereas TEEzz overcomes several limitations of TEE fuzzing, it requires fully rooted devices, multiple available CAlibs, and specified TEE platforms. The requirements of TEEzz together with its burdensome manual work for specified fuzzing template extractor make its fuzzing approach neither general nor efficient.

8 Conclusion

There exist too few frameworks for fuzzing TEEs considering the market share ARM TrustZone occupies. The two most robust solutions currently are PartEMU and TEEzz, each significant achievement that addresses this concern. However, they suffer from several issues that make them non-ideal candidates for rapidly and scalably fuzzing TAs. We attempt to solve this issue using COTS devices with ARM CoreSight-assisted fuzzing in which we evaluate using CoreSight-enabled prototype smartphones to fuzz TAs. We, therefore, implement CROWBAR and outline in detail the process by which CoreSight features are discovered, enabled, and configured on a range of prototype devices. This allows us to demonstrate CoreSight-assisted trace integration and evaluation by fuzzing TAs from multiple vendors. In total, we evaluate 5 TA and find 3 unique crashes. Finally, we provide a detailed discussion of the challenges faced in leveraging CoreSight-assisted fuzzing on prototype devices across a representative range of device vendors that will hopefully prove useful for future research in the area.

Statements and Declarations

Funding U.S. Department of Energy award number DE-SC0018430

Competing Interests There are no competing interests.

Author Contributions H.S, M.H., S.N, Y.L., and D.S wrote the main manuscript text. H.S and S.N prepared the figures. All authors reviews the manuscript.

Data Availability Data and materials are available upon request.

Ethical Approval Not applicable.

References

1. Ltd A. System IP. https://www.arm.com/products/silicon-ip-system. (Date last Access 28 July 2022)
2. Fasano A, Ballo T, Muench M, Leek T, Bulekov A, Dolan-Gavitt B, Egele M, Francillon A, Lu L, Gregory N et al (2021) Sok: Enabling security analyses of embedded systems via rehosting. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, pp. 687–701
3. Liang H, Pei X, Jia X, Shen W, Zhang J (2018) Fuzzing: State of the art. IEEE Trans Reliab 67(3):1199–1218
4. Harrison L, Vijayakumar H, Padhye R, Sen K, Grace M (2020) Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In: Proceedings of the 29th USENIX Conference on Security Symposium. SEC'20. USENIX Association, USA
5. Busch M, Machiry A, Spensky C, Vigna G, Kruegel C, Payer M (2023) Teezz: Fuzzing trusted applications on cots android devices. In: 2023 2023 IEEE Symposium on Security and Privacy (SP) (SP), pp. 220–235. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/SP46215.2023.00013. https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00013
6. ARM (2018) ARM® Embedded Trace Macrocell Architecture Specification: ETMv4.0 to ETMv4.6. ARM
7. Fioraldi A, Maier D, Eißfeldt H, Heuse M (2020) {AFL++}: Combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20)
8. ARM (2015) CoreSight SoC-400. ARM. (Date last Access 28 July 2022)
9. Pinto S, Santos N (2019) Demystifying arm trustzone: A comprehensive survey. ACM Comput Surv (CSUR) 51(6):1–36
10. Ning Z, Zhang F (2019) Understanding the security of arm debugging features. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 602–619. https://doi.org/10.1109/SP.2019.00061
11. Bicchierai L (2019) The prototype iPhones that hackers use to research Apple's most sensitive code. https://www.vice.com/en/article/gyakgw/the-prototype-dev-fused-iphones-that-hackers-use-to-research-apple-zero-days
12. https://github.com/ARM-software/CSAL. (Date last Access 2 Dec 2022)
13. ARM. Embedded Trace Macrocell Architecture Specification. https://developer.arm.com/documentation/ihi0014/q/preface/Additional-reading/The-ETM-documentation-suite. (Date last Access 14 July 2022)
14. Wikipedia: MurmurHash (2022) https://en.wikipedia.org/wiki/MurmurHash
15. Google. Google Pixel 7 Tech Specs. https://store.google.com/us/product/pixel_7_specs?hl=en-GB. (Date last Access 1 July 2022)
16. Yoo S-G, Park K-Y, Kim J (2012) Software architecture of jtag security system. WSEAS Transactions on Systems 11(8):398–408
17. Fedewa J (2018) Huawei will stop providing bootloader unlocking for all new devices. https://www.xda-developers.com/huawei-stop-providing-bootloader-unlock-codes/
18. Costin A, Zarras A, Francillon A (2016) Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 437–448
19. Talebi SMS, Tavakoli H, Zhang H, Zhang Z, Sani AA, Qian Z (2018) Charm: Facilitating dynamic analysis of device drivers of mobile systems. In: USENIX Security Symposium, pp. 291–307
20. Linaro (2018) Pseudo Trusted Applications. https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html#pseudo-trusted-applications. (Date last Access 10 Oct 2022)
21. Feng B, Mera A, Lu L (2020) P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: Proceedings of the 29th USENIX Conference on Security Symposium, pp. 1237–1254
22. Gustafson E (2019) Toward the analysis of embedded firmware through automated re-hosting. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses