Towards Comprehensive Fuzzing of TrustZone TAs

.HACK 2024

Juhyun Song
Korea University

About

Juhyun Song

https://juhyun167.github.io/about

Experiences

- **KAIST Hacking Lab**
  - Undergraduate Intern (2023.12~Present)
- **Samsung Electronics**
  - Intern, Conducted research on TrustZone Security (2023.03~06)
- **KITRI Best of the Best**
  - Hall of Fame, Conducted research on device drivers (2020.07~2021.02)

Motivation

“Have you heard of **TrustZone?**”

Motivation

“Have you heard of **TrustZone**?”

Motivation

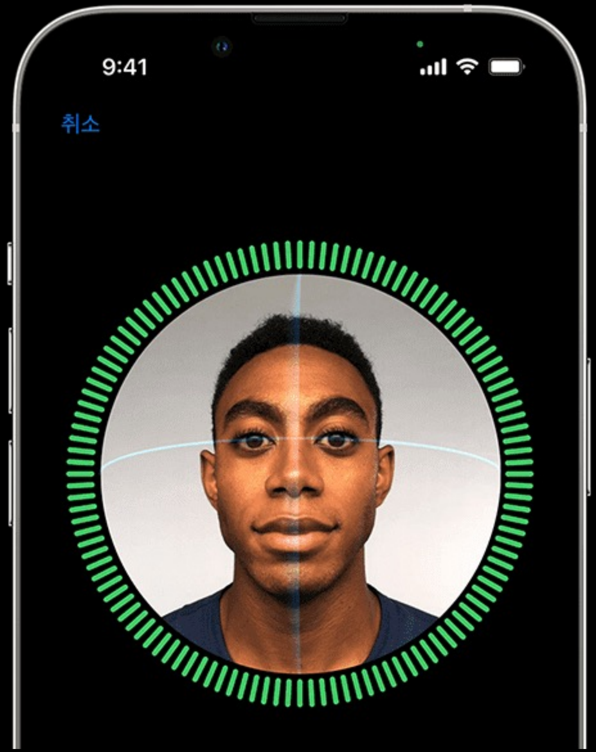
usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION
OP-TEE documentation
latest
Search docs
Getting started
OP-TEE Documentation
OP-TEE Documentation
This is the official location for OP-TEE documentation.
About OP-TEE
Coding standards
Contribute
Contact
GLOBALPLATFORM
GlobalPlatform
TEE Client API
Version 1.0
Public Release
July 2010
Document Reference:
GLOBALPLATFORM
THE STANDARD FOR SECURE DIGITAL SERVICES AND DEVICES
GlobalPlatform Technology
TEE Internal Core API Specification
Version 1.1.2.50 (Target v1.2)
Public Review
June 2018
Document Reference: GPD_SPE_010
SAMSUNG
SAMSUNG
SAMSUNG

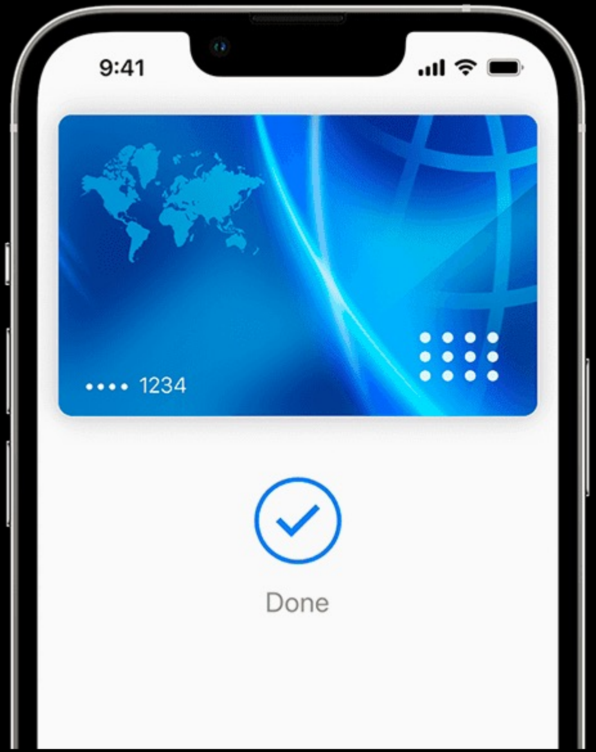
Topics

- Trusted Execution Environments and TrustZone
- Trusted Applications
- Challenges in TA fuzzing
- Our approach

Computing Ecosystem

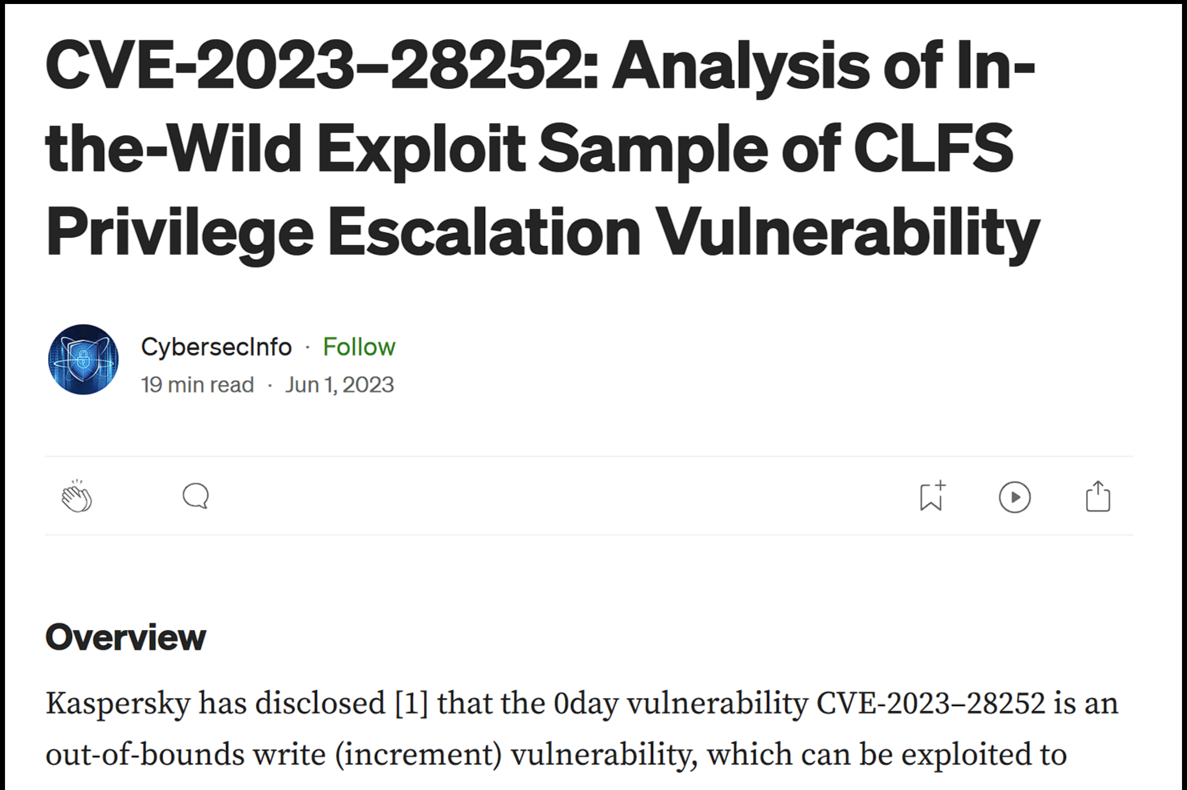
- Increasing number of services are being deployed on the cloud.
- Growing number of mobile devices are managing security-sensitive tasks.





Computing Ecosystem

- What would happen if these systems were **hacked?**
- Would the services and credentials remain secure, even in the face of privileged attackers?



CVE-2023–28252: Analysis of In-the-Wild Exploit Sample of CLFS Privilege Escalation Vulnerability

CybersecInfo · Follow
19 min read · Jun 1, 2023

Overview

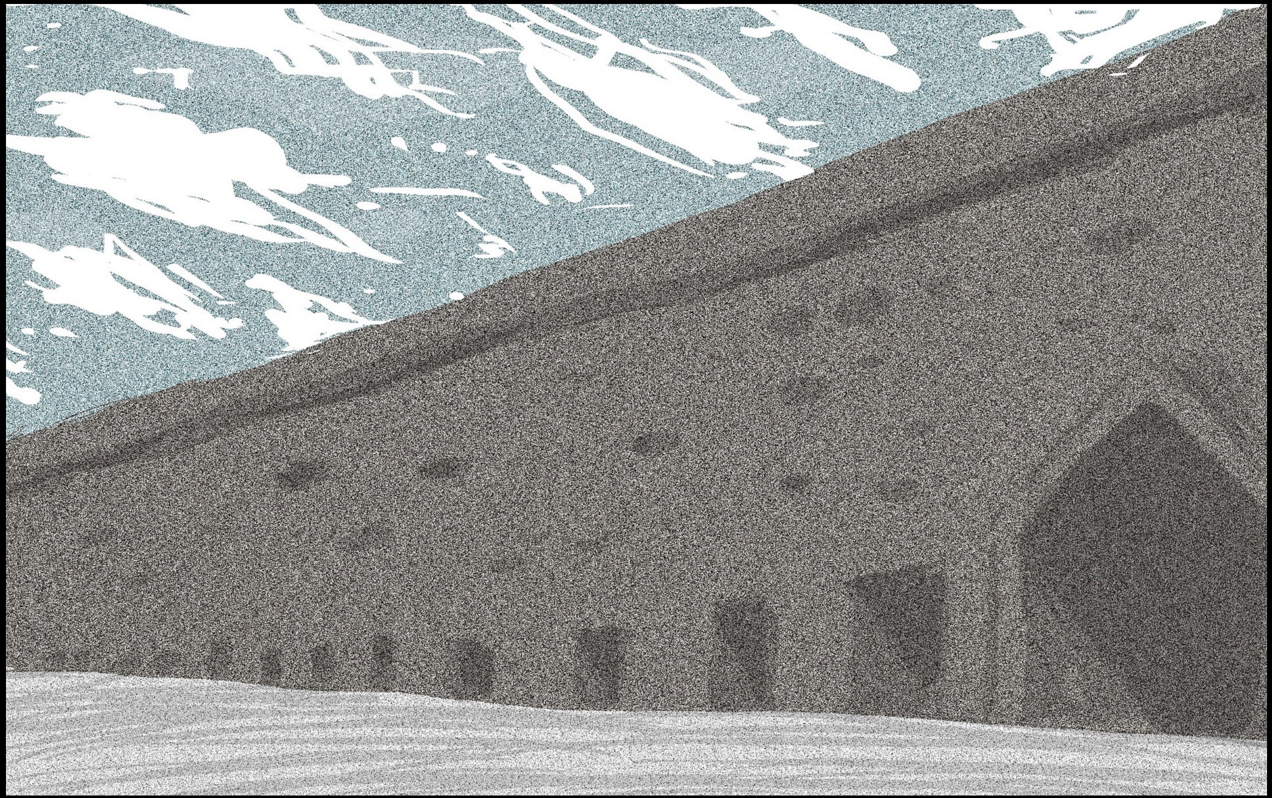
Kaspersky has disclosed [1] that the 0day vulnerability CVE-2023–28252 is an out-of-bounds write (increment) vulnerability, which can be exploited to

Trusted Execution Environments

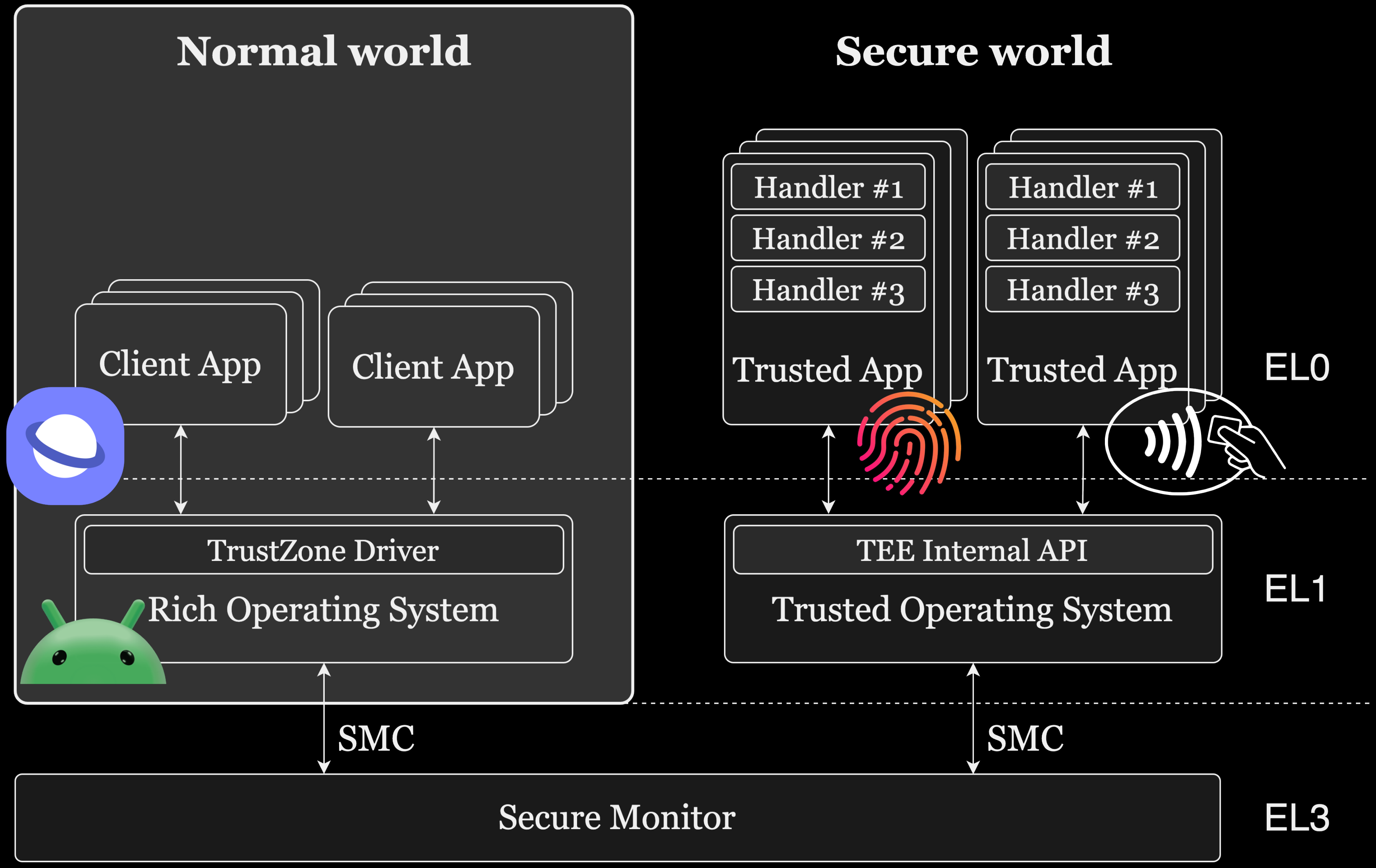
- **Trusted Execution Environments (TEEs)** significantly reduce the attack surface against powerful adversaries.
- TEEs guarantee that the code and data residing within the **secure region** of the main processor maintain both confidentiality and integrity.

TrustZone

- **TrustZone** is a security extension for ARM processors.
- It partitions the processor into two distant realms: the **secure world** and the **normal world.**

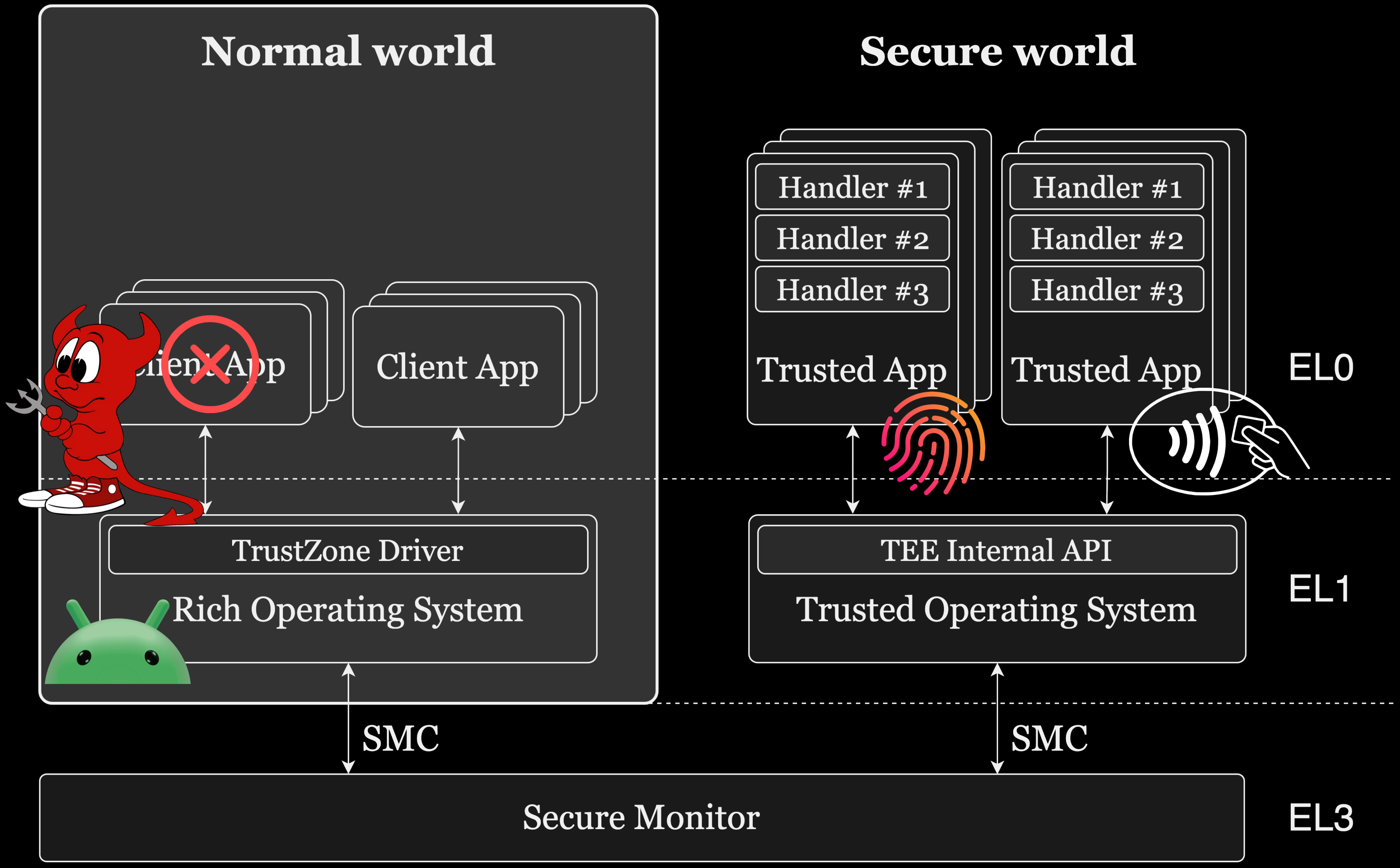


TrustZone



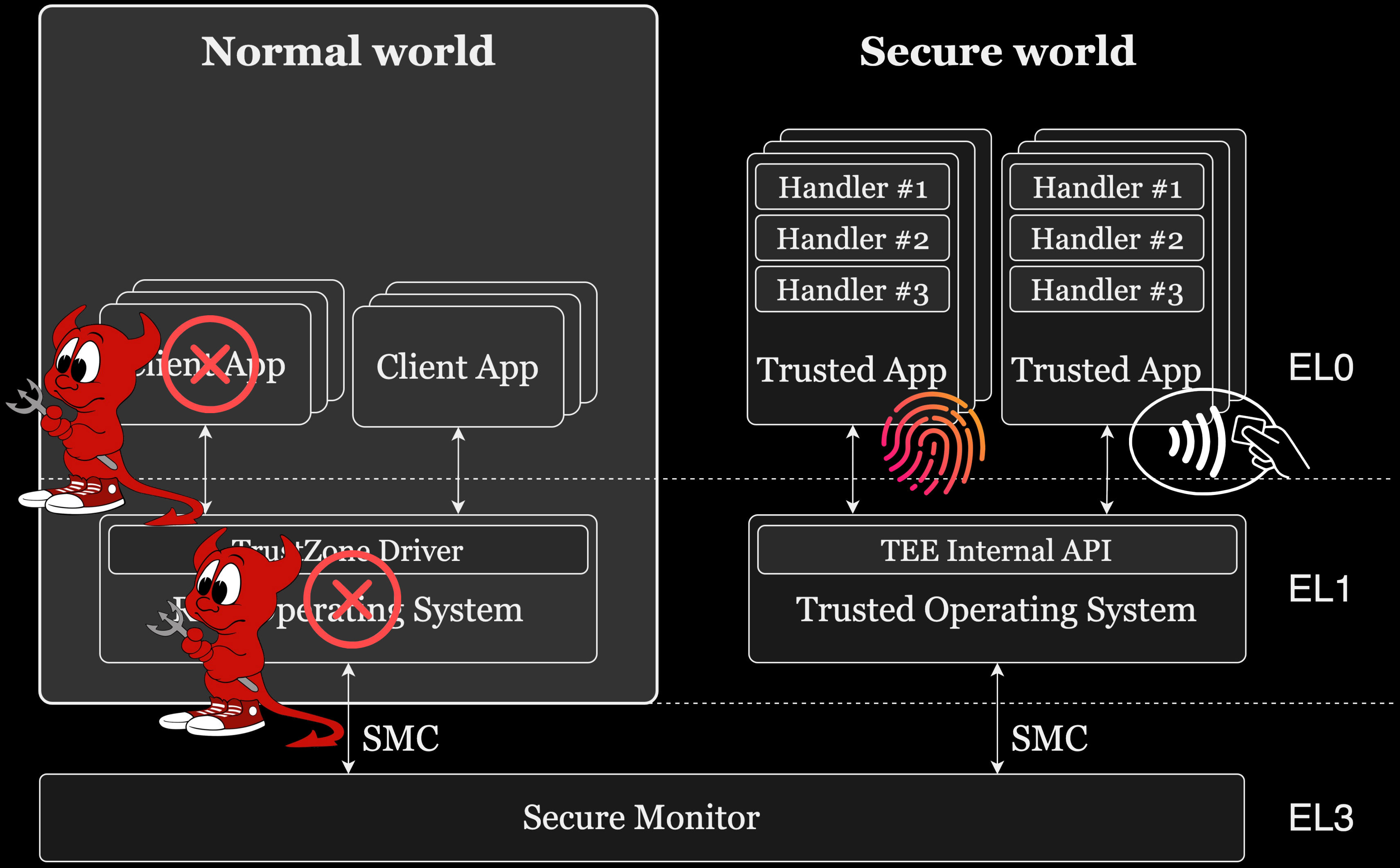
Normal world
Secure world
Client App
Client App
Handler #1
Handler #2
Handler #3
Trusted App
Handler #1
Handler #2
Handler #3
Trusted App
EL0
TrustZone Driver
Rich Operating System
TEE Internal API
Trusted Operating System
EL1
SMC
SMC
Secure Monitor
EL3

TrustZone

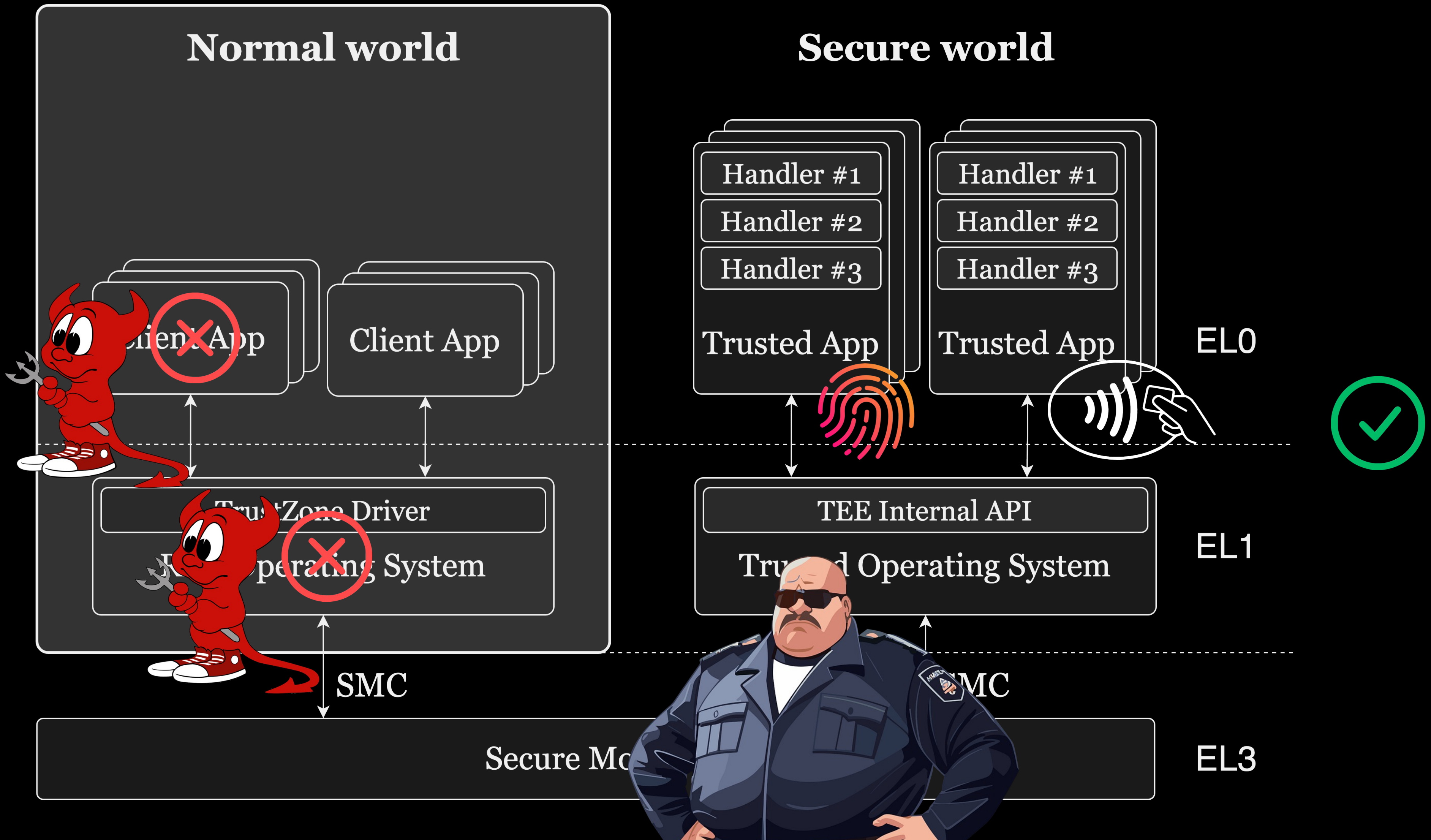


Normal world
Secure world
Client App
Client App
Handler #1
Handler #2
Handler #3
Trusted App
Handler #1
Handler #2
Handler #3
Trusted App
EL0
TrustZone Driver
Rich Operating System
TEE Internal API
Trusted Operating System
EL1
SMC
SMC
Secure Monitor
EL3

TrustZone

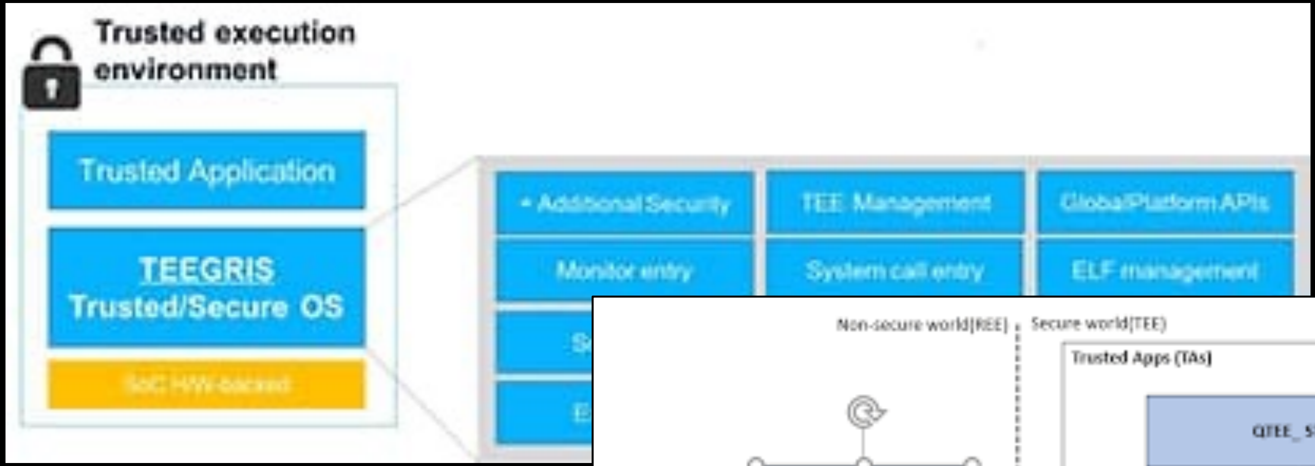

Normal world
Secure world
Handler #1
Handler #2
Handler #3
Trusted App
Handler #1
Handler #2
Handler #3
Trusted App
Client App
EL0
TEE Internal API
Trusted Operating System
EL1
SMC
SMC
Secure Monitor
EL3

TrustZone

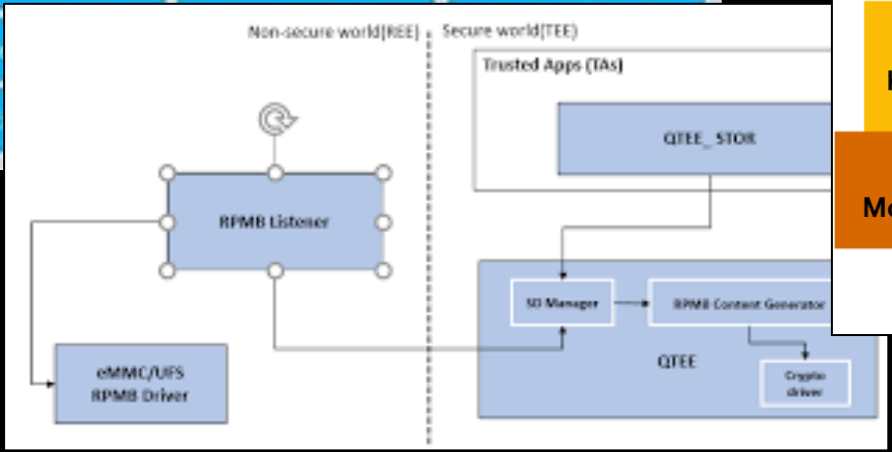

Normal world
Secure world
Client App
Client App
Handler #1
Handler #2
Handler #3
Trusted App
Handler #1
Handler #2
Handler #3
Trusted App
EL0
TEE Internal API
EL1
SMC
EL3

TrustZone

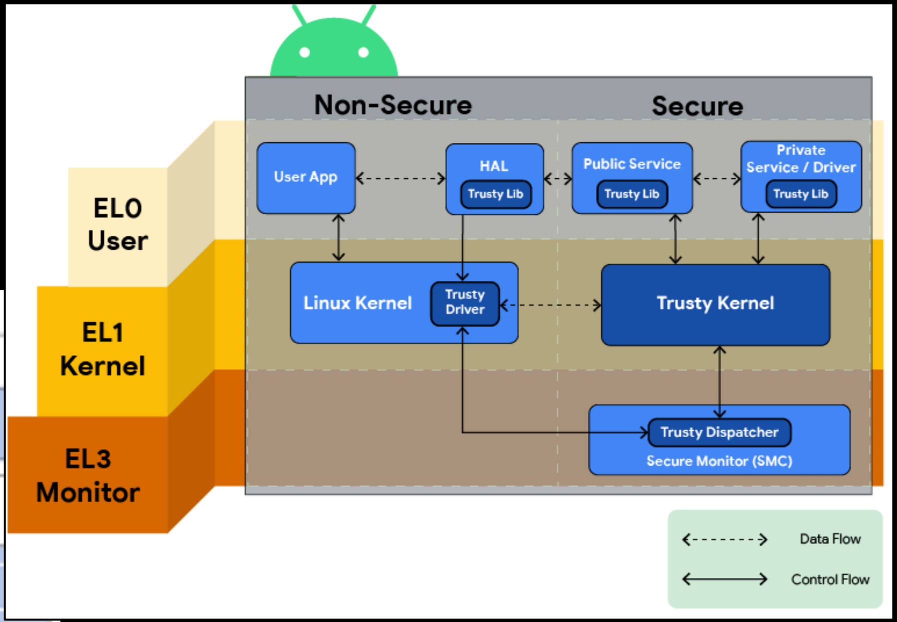
• Manufacturers construct TEEs based on TrustZone by implementing their unique software architectures.



SAMSUNG



Qualcomm

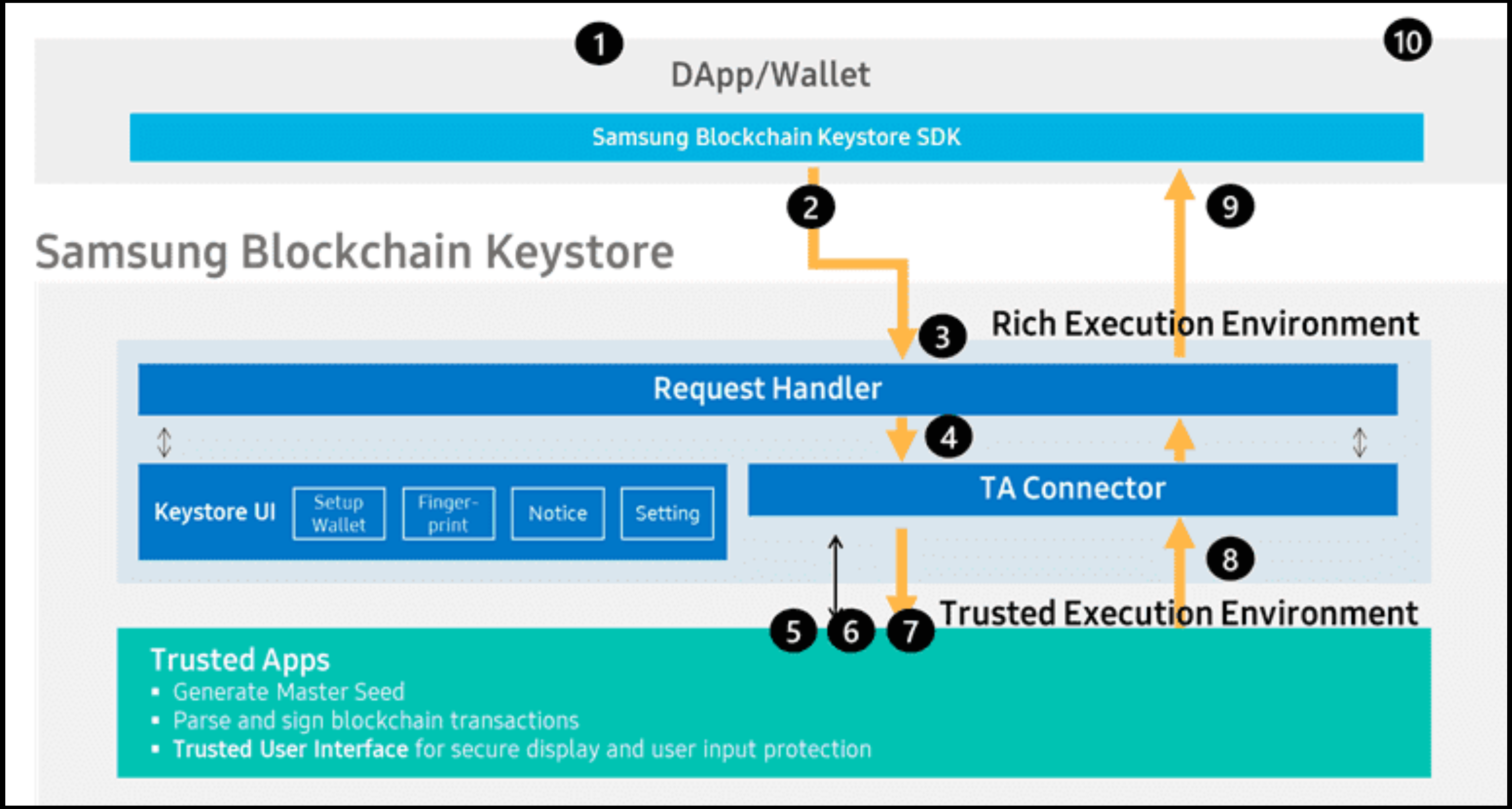


Android

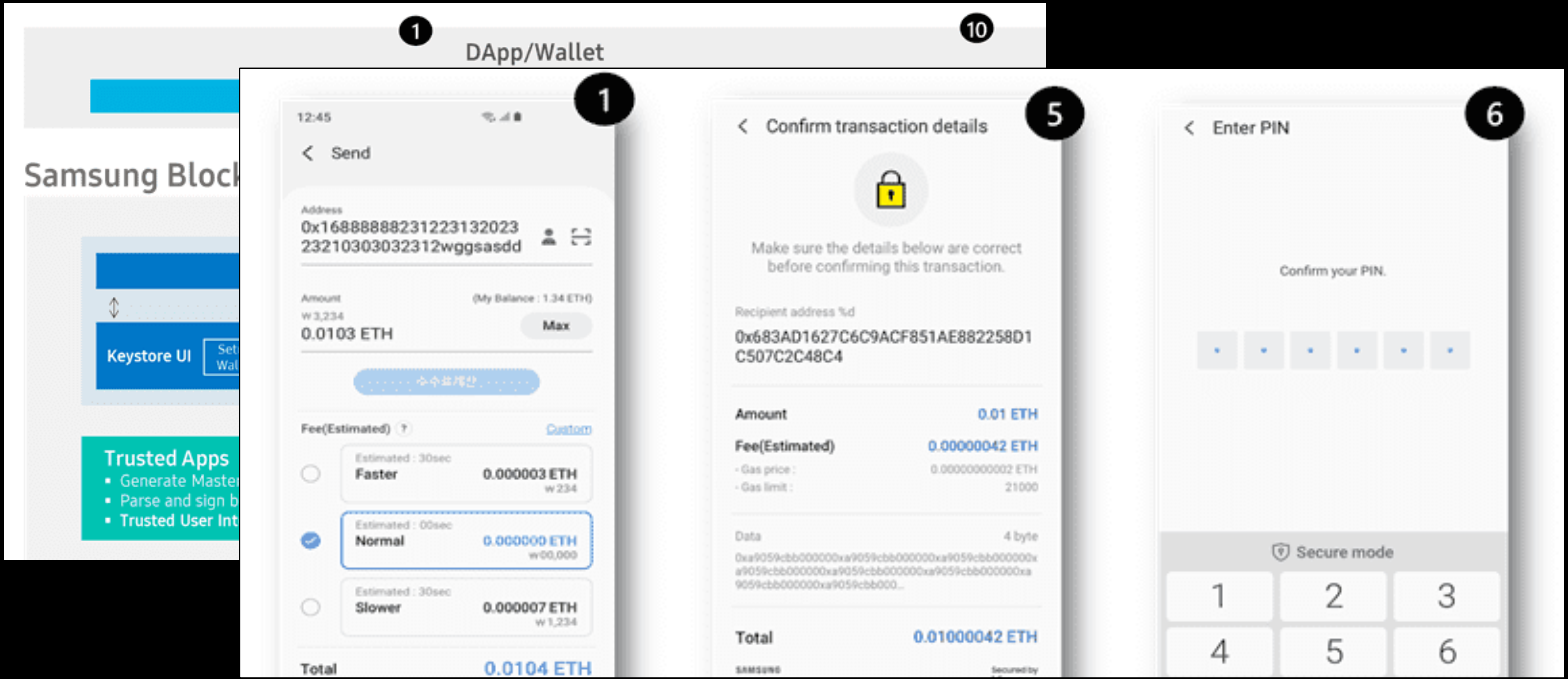
Trusted Applications

- **Trusted Applications (TAs)** are applications that operate within the TEE.
- TAs provide essential security feature through a secure interface.
  - e.g., mobile payments, cryptographic keystore, confidential computing

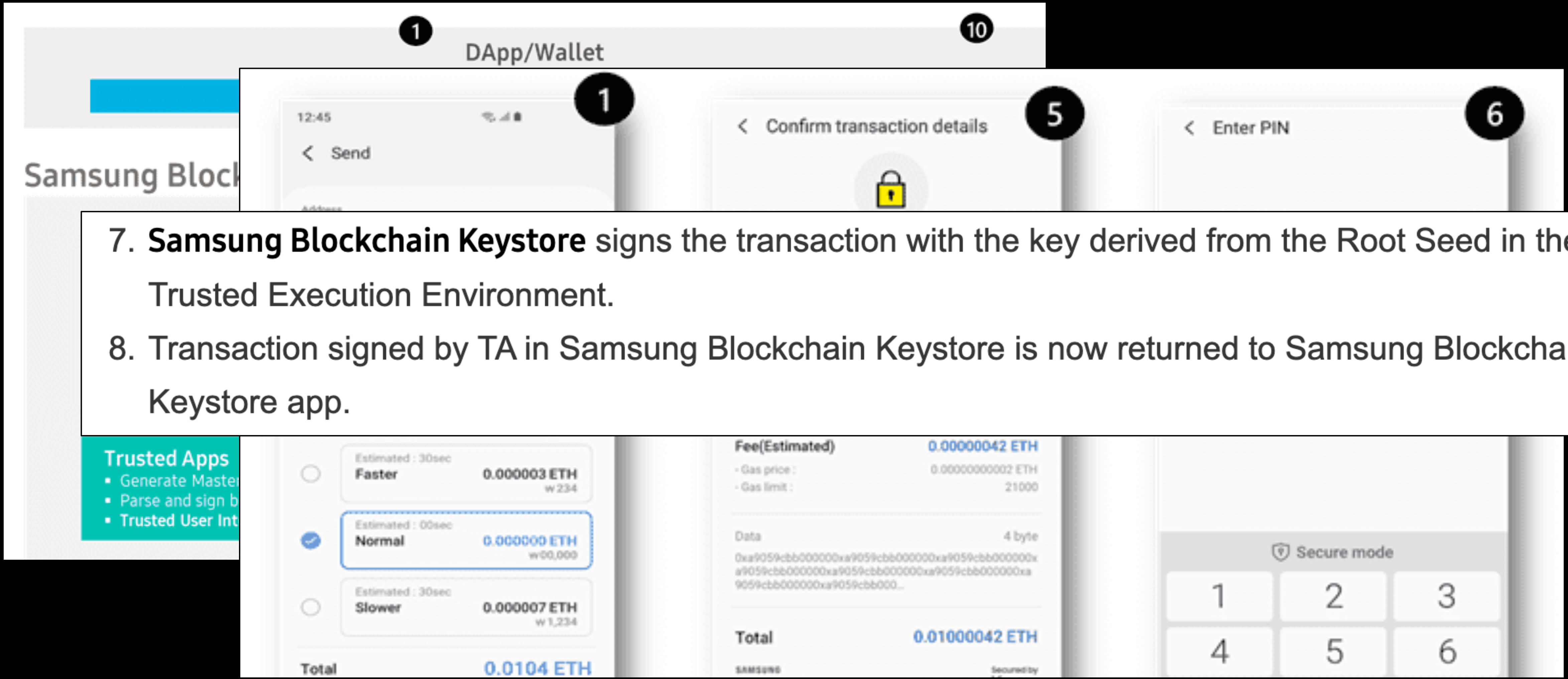
Trusted Applications



Trusted Applications



Trusted Applications



7. **Samsung Blockchain Keystore** signs the transaction with the key derived from the Root Seed in the Trusted Execution Environment.
8. Transaction signed by TA in Samsung Blockchain Keystore is now returned to Samsung Blockchain Keystore app.

Trusted Applications

- TAs implement each functionality as distinct command handlers.
  - analogous to device drivers and GUI programs.

```
TEE_Result TA_InvokeCommandEntryPoint(void __unused *session,
                                      uint32_t command,
                                      uint32_t param_types,
                                      TEE_Param params[4])
{
    switch (command) {
    case TA_SECURE_STORAGE_CMD_WRITE_RAW:
        return create_raw_object(param_types, params);
    case TA_SECURE_STORAGE_CMD_READ_RAW:
        return read_raw_object(param_types, params);
    case TA_SECURE_STORAGE_CMD_DELETE:
        return delete_object(param_types, params);
```

Trusted Applications

- **Client Applications (CAs)** are applications in the normal world that communicate with TAs.
- CAs are required to provide the command ID and parameters for the desired TA commands, and they receive a result code upon completion.

```
op.params[0].tmpref.buffer = id;
op.params[0].tmpref.size = id_len;

op.params[1].tmpref.buffer = data;
op.params[1].tmpref.size = data_len;

res = TEEC_InvokeCommand(&ctx->sess,
                TA_SECURE_STORAGE_CMD_READ_RAW,
                &op, &origin);
```

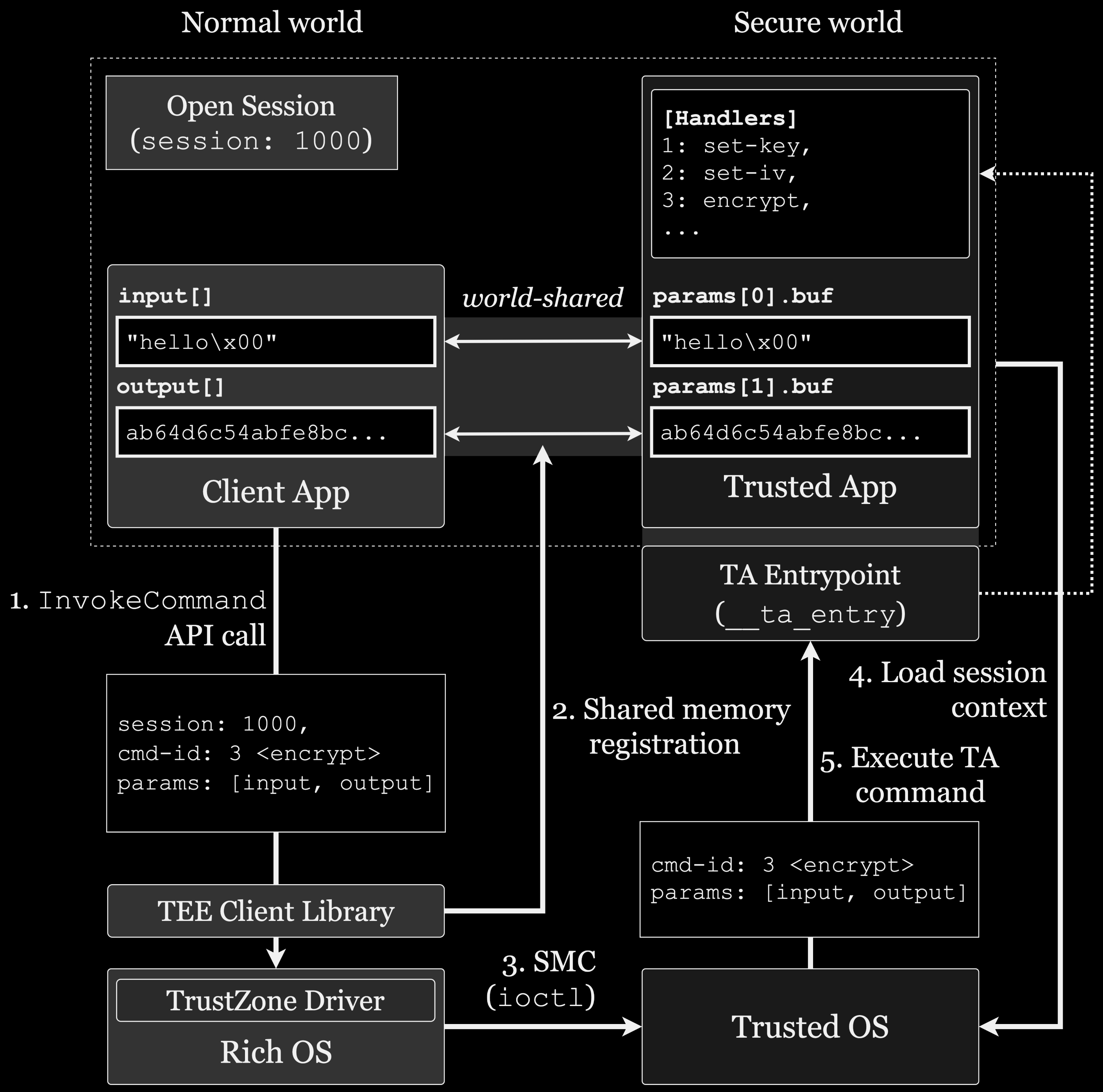


FIGURE 4. World switch procedure in OP-TEE based TAs. In this example, the CA is invoking the encrypt handler with two memory parameters.

TrustZone Security

- We now understand that TrustZone maintains its security even if the normal world OS is compromised.
- However, are we certain that TrustZone **itself** is secure?
  - Are there absolutely no vulnerabilities in TAs and TEEs?

TrustZone Security

- It turns out that TrustZone have been successfully attacked due to **security flaws** in recent years.
  - e.g., absent mitigations, validation bugs, map physical memory
- Some vulnerabilities could even be leveraged to compromise the normal world OS.

SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems

David Cerdeira, Centro Algoritmi, Universidade do Minho, david.cerdeira@dei.uminho.pt
Nuno Santos, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, nuno.santos@inesc-id.pt
Pedro Fonseca, Department of Computer Science, Purdue University, pfonseca@purdue.edu
Sandro Pinto, Centro Algoritmi, Universidade do Minho, sandro.pinto@dei.uminho.pt

***Abstract*—Hundreds of millions of mobile devices worldwide rely on Trusted Execution Environments (TEEs) built with Arm TrustZone for the protection of security-critical applications (e.g., DRM) and operating system (OS) components (e.g., Android keystore). TEEs are often assumed to be highly secure; however, over the past years, TEEs have been successfully attacked multiple times, with highly damaging impact across various platforms. Unfortunately, these attacks have been possible by the presence of security flaws in TEE systems. In this paper, we aim to understand which types of vulnerabilities and limitations affect existing TrustZone-assisted TEE systems, what are the main challenges to build them correctly, and what contributions can be borrowed from the research community to overcome them. To this end, we present a security analysis of popular TrustZone-assisted TEE systems (targeting Cortex-A processors) developed by Qualcomm, Trustonic, Huawei, Nvidia, and Linaro. By studying publicly documented exploits and vulnerabilities as well as by reverse engineering the TEE firmware, we identified several critical vulnerabilities across existing systems which makes it legitimate to raise reasonable concerns about the security of commercial TEE implementations.**

***Index Terms*—TEE, TrustZone, Security Vulnerabilities, Arm**

I. Introduction

Trusted Execution Environments (TEE) are a key mechanism to protect the integrity and applications. By leveraging dedicated h the execution of security-sensitive appl domains isolated from the platform Arm TrustZone [1] has become the ogy to implement TEEs in mobile employed in industrial control sy low-end devices [4]. In the future enabled IoT devices are expect provide secure environments fo

TrustZone-assisted TEEs are secure than modern OSes due to enforced by TrustZone technol Computing Base (TCB), which smaller than standard become widely ad malware [6–10] rate TrustZon operations i banking [1 systems h doubt on TEEs ca

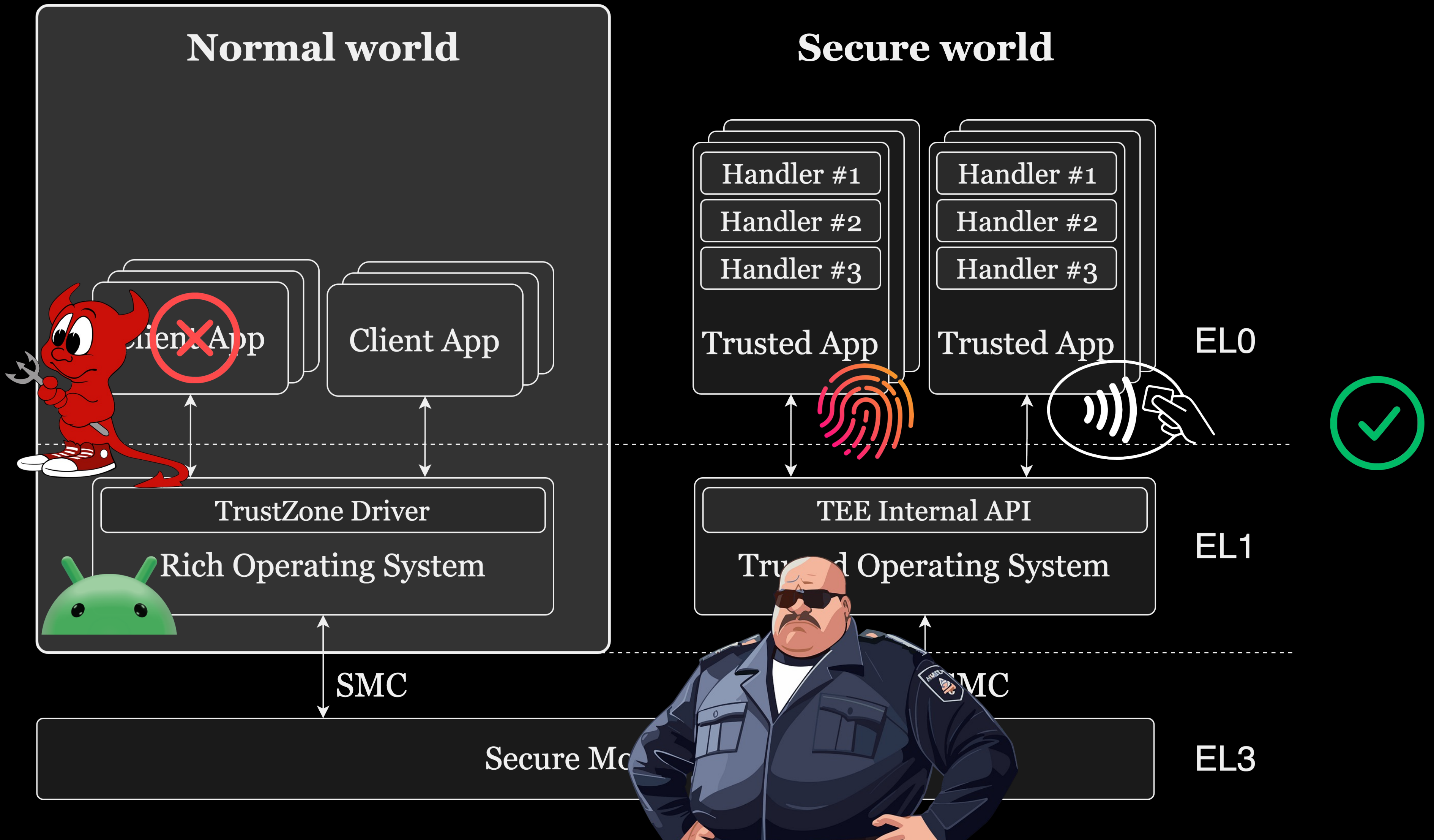
In this paper, we perform a systematic study of publicly disclosed vulnerabilities in commercial TrustZone-assisted TEEs for Arm Cortex-A devices. Despite the existence of multiple security reports affecting such systems, this information tends to be scattered and, in certain cases, unverified, which makes it difficult to obtain a comprehensive understanding of the prevailing vulnerabilities and overall security properties of these systems. To fill this gap, we analyzed 207 TEE bug reports spanning a nearly 5 years, from 2013 until mid-2018, focusing on widely deployed TEE systems developed for Arm-based devices by five major vendors: Qualcomm, Trustonic, Huawei, Nvidia, and Linaro. We examined and categorized numerous vulnerabilities, in particular, some of those that have been leveraged to carry out successful attacks. From our analysis, along with the manual inspection of TEE firmware, we have gained multiple insights about the extent and causes of existing vulnerabilities, and about potential solutions to mitigate them.

One first observation is that TEE systems have a long history of *critical implementation bugs*. Numerous bugs have been (and continue to be) found inside TEE applications – named Trusted Applications (TAs) – and inside the trusted kernel responsible for managing the TEE runtime. Many bugs involve input validation errors, such as buffer overflows. As multiple attacks, these bugs can be leveraged to id's Linux kernel or to entirely compromise the devices featuring TEEs by Qualcomm [14, 15], 7], or Huawei [18].

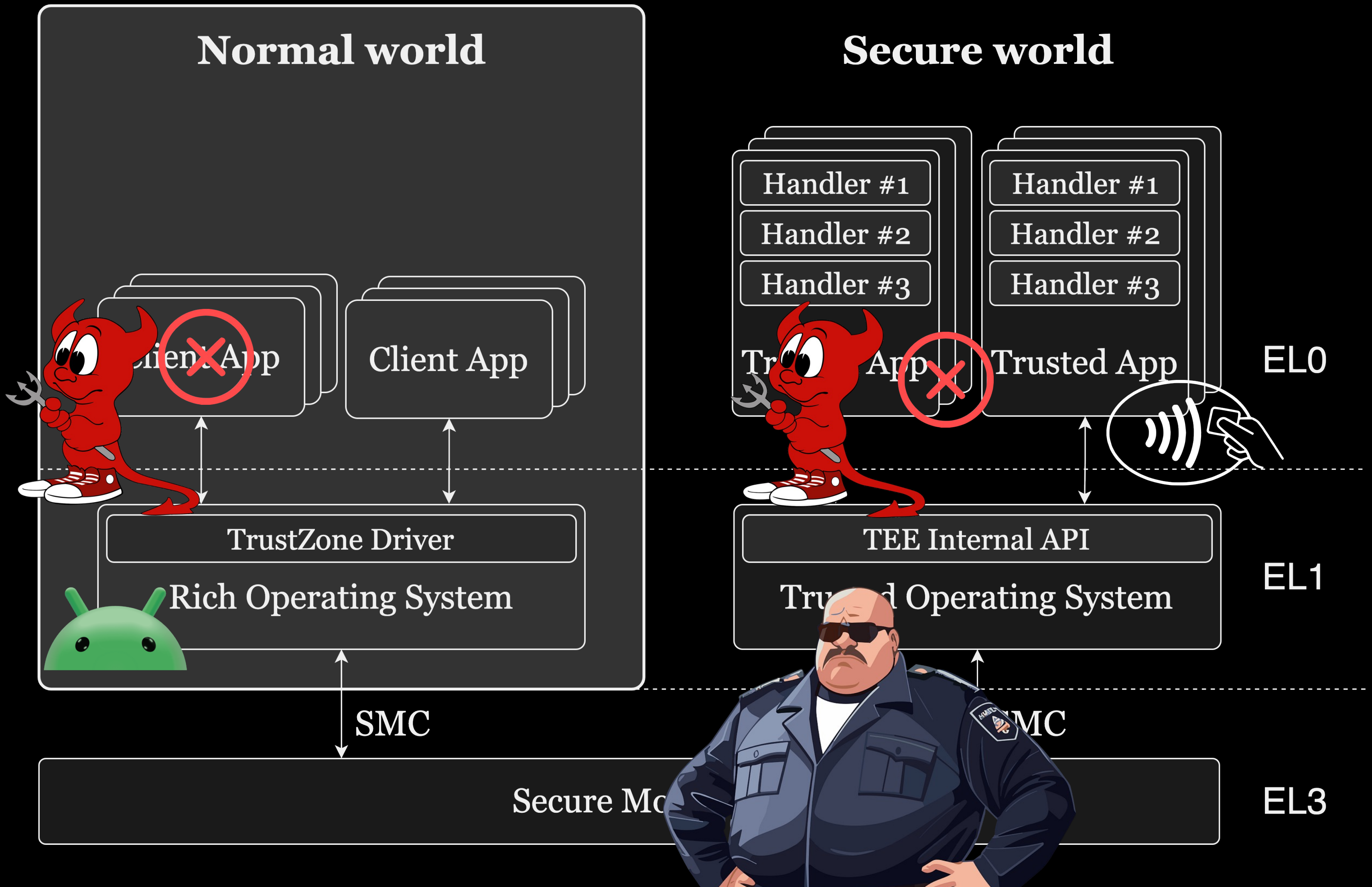
g vulnerable TAs is facilitated by the *al deficiencies* of TrustZone-assisted TEE e, the memory protection mechanisms modern OSes, e.g., ASLR or page guards, r ill-implemented in most analyzed systems. tend to expose a large attack surface, s TEE kernel system calls that can be r example, on Qualcomm's TEE, any TA regions of the host OS. As a result, by TA, e.g., leveraging a buffer overflow, trol Android [15].

roperties are overlooked in most ral and microarchitectural rity of the TEE. Some behavior of trusted tural side-channels components that TEE-restricted

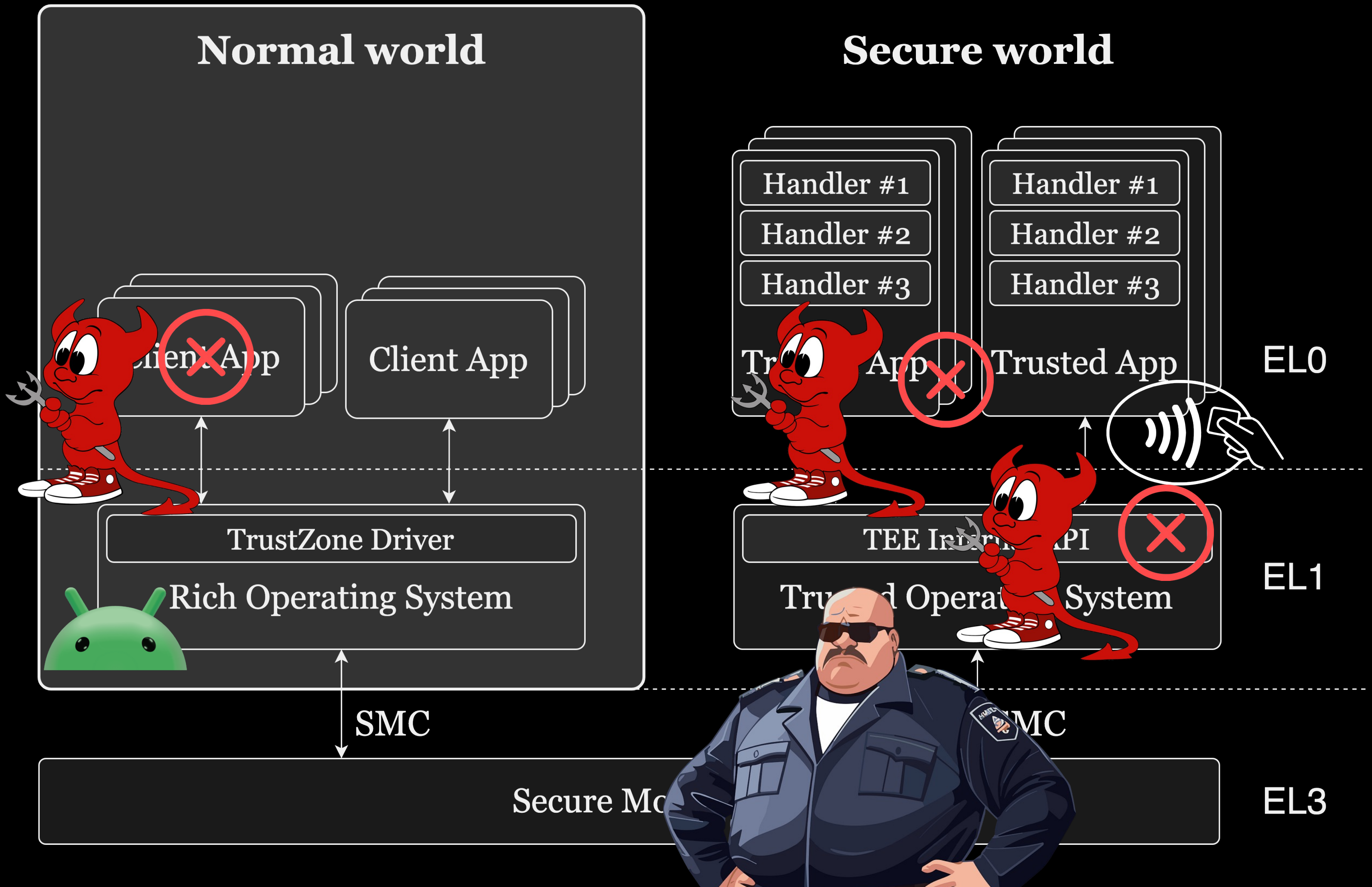
TrustZone Security



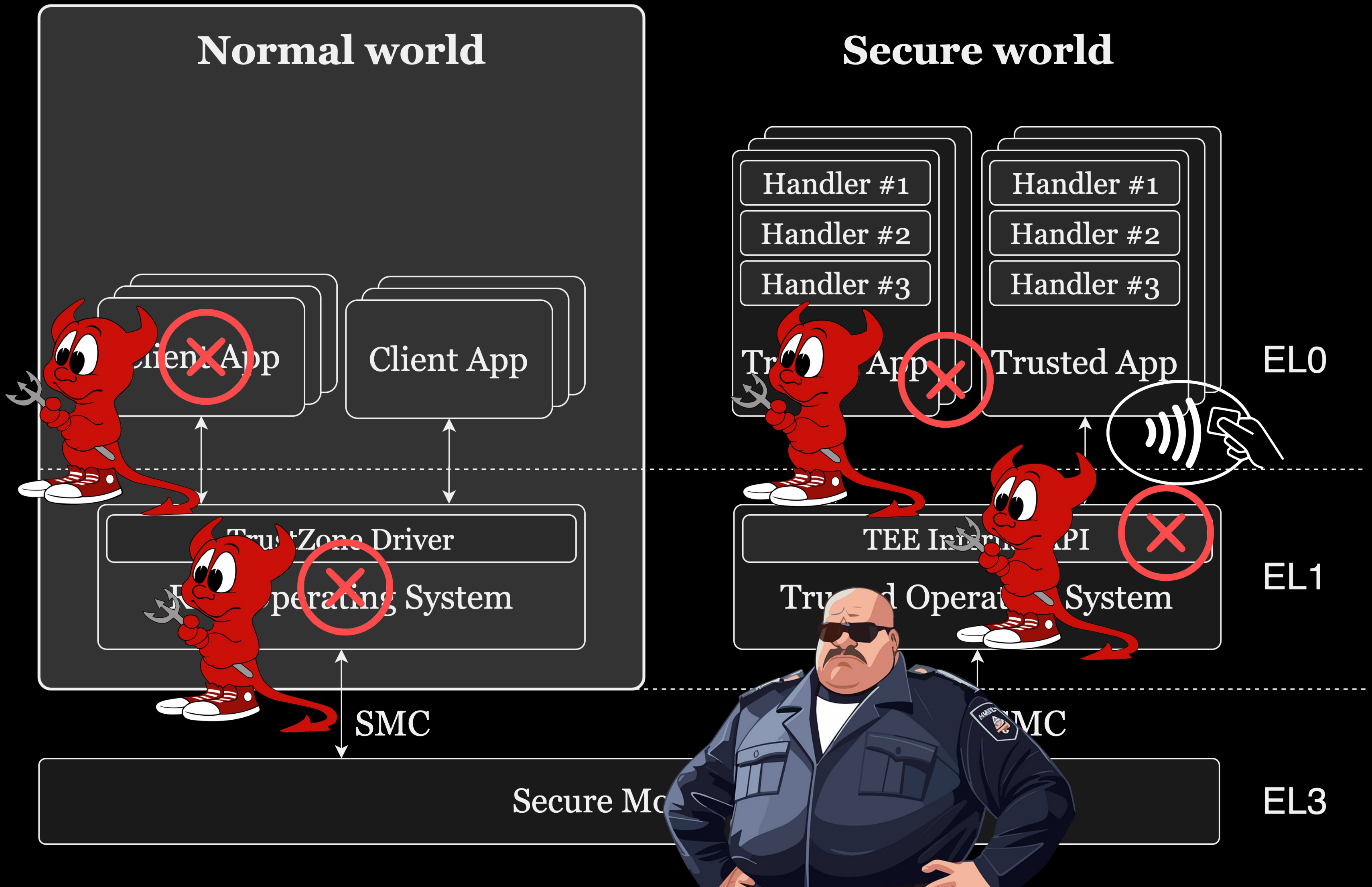
TrustZone Security



TrustZone Security



TrustZone Security



TrustZone Security

- Among the implementation issues in TrustZone TEEs, **validations bugs in TAs** constituted the largest portion. (33.16%)

Class	Subclass	# Bugs
Validation Bugs	Secure Monitor	2 (1.07%)
	Trusted Applications	62 (33.16%)
	Trusted Kernel	52 (27.81%)
	Secure Boot Loader	5 (2.67%)
Functional Bugs	Memory Protection	32 (17.11%)
	Peripheral Configuration	8 (4.28%)
	Security Mechanisms	11 (5.88%)
Extrinsic Bugs	Concurrency Bugs	11 (5.88%)
	Software Side Channels	4 (2.14%)

Table VI
Number of bug reports involving implementation issues.

Summary

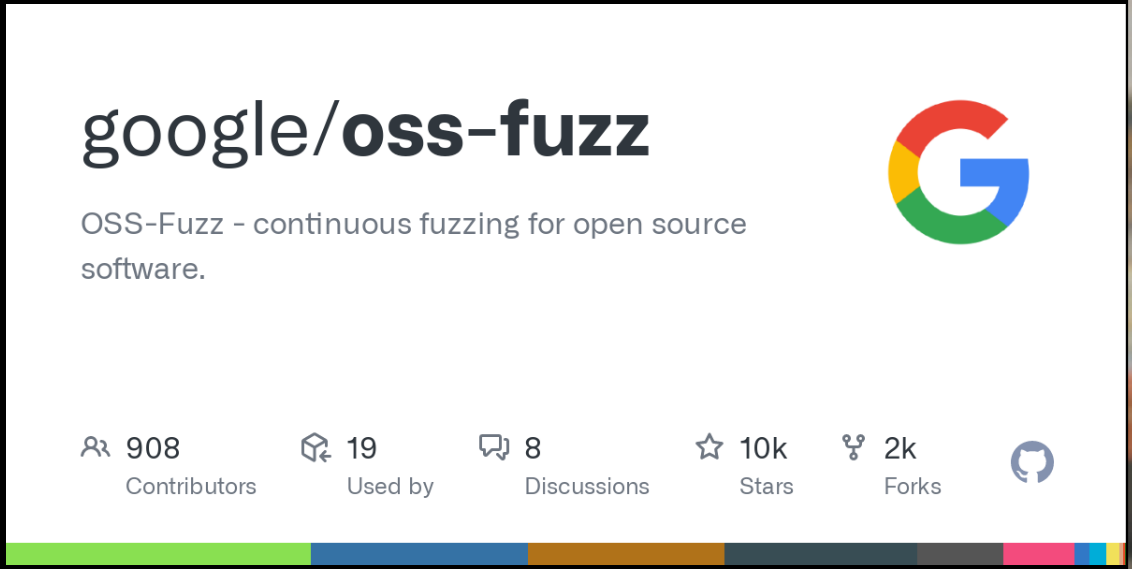
This is the third part of a blog series covering my security research into Samsung's TrustZone . Other parts in this series so far: 1, 2.

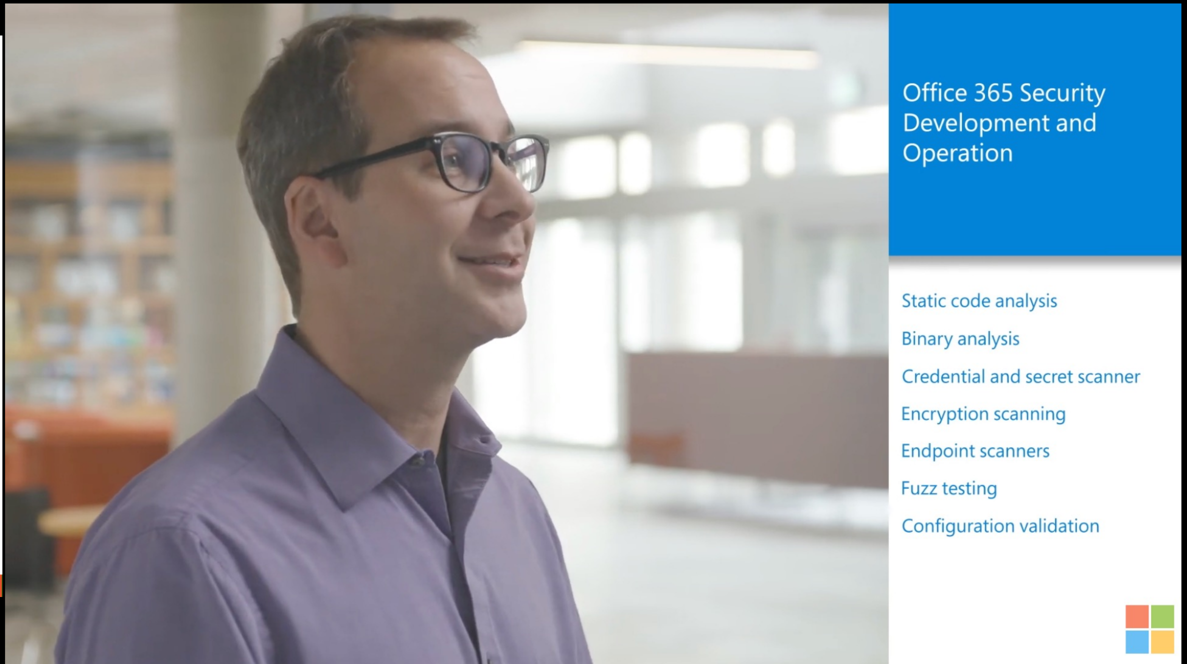
This post covers the following vulnerabilities that I have found:

- SVE-2017–8888: Authentication Bypass + Buffer overflow in tlc_server
- SVE-2017–8889: Stack buffer overflow in ESECOMM Trustlet
- SVE-2017–8890: Out-of-bounds memory read in ESECOMM Trustlet
- SVE-2017–8891: Stack buffer overflow in ESECOMM Trustlet
- SVE-2017–8892: Stack buffer overflow in ESECOMM Trustlet
- SVE-2017–8893: Arbitrary write in ESECOMM Trustlet

Fuzzing

- **Fuzzing** is a process of identifying security vulnerabilities by repeatedly testing a program with modified inputs.
- It has been widely accepted in the field of software security assessment.
  - e.g., Microsoft SDL, Google OSS-Fuzz





Challenges in TA Fuzzing

- Fuzzing TrustZone presents significant challenges due to its **black-box** operation.
    - Reading and modifying states in the secure world is not feasible.
    - Instrumenting the TA or trusted OS is **restricted** without appropriate access.
- The fuzzer faces extreme difficulty to gain meaningful insights about its target.

Previous Works

- **Finding 1-Day Vulnerabilities in Trusted Applications using Selective Symbolic Execution (NDSS 2020)**
    - emulates TA execution environemnts using selective symbolic execution.
    - necessitates a “patched” version of TA, resulting in limitations during production testing.

Finding 1-Day Vulnerabilities in Trusted Applications using Selective Symbolic Execution

Marcel Busch and Kalle Dirsch
{marcel.busch, kalle.dirsch}@fau.de
IT Security Infrastructures Lab
Department of Computer Science
Friedrich-Alexander University Erlangen-Nürnberg (FAU)

***Abstract*—Trusted Execution Environments (TEEs) constitute a major building block for modern mobile devices' security architectures. Yet, the analysis tools available to researchers seeking to examine these critical components are rudimentary compared to the vast range of sophisticated tools available for other execution contexts (*i.e.,* Linux or Windows userland). We see the primary reason for the lack of tools is originating from the closed-source nature of TEEs. Specifically, the analysis of Trusted Applications (*i.e.,* userland applications executed in a TEE) is of vital importance, since they account for the largest attack surface. However, hardware primitives (*i.e.,* ARM TrustZone) prevent access to this high-privileged context and thwart any form of dynamic analysis.**

In this paper, we present our approach to investigate 1-day vulnerabilities using selective symbolic execution of real-world Trusted Applications (TAs). Our system, SimTA, is based on angr and emulates the TA's execution environment. We build SimTA based on insights gained from manual static analysis of a commercially and widely deployed closed-source TEE by using an exploit on a physical device. In our evaluation, we elaborate on how SimTA facilitates the binary-diff-guided analysis by replicating the analysis of a known critical vulnerability. Additionally, we reveal two further issues, an authentication bypass and a heap-based buffer overflow, that have quietly been introduced by the vendor.

I. Introduction

In 2016, at an event called “GeekPwn”, Stephens [22] presented a chain of exploits that ultimately led to an arbitrary code execution within the TEE of Huawei [25]. Using these exploits, he could unlock the targeted device using the fingerprint sensor with a finger of *any* person or even a nose. His privilege escalation into the TEE is connected to CVE-2016-8764, which is an input validation vulnerability that an attacker can leverage to execute arbitrary shellcode within the TEE context.

A common way to investigate vulnerabilities similar to this is binary-diffing in combination with meticulous manual analysis. To extract the patch for the vulnerability in question, we refer to CVE-2016-8764's summary [19] and identify the latest affected version to compare it with its succeeding version. One problem that can arise while extracting the patch is that not only the vulnerable sequence of instructions appears in the binary-diff, but many others. For example, new features could have been introduced, or compiler flags might have changed, resulting in irrelevant sequences. In this case, indicators such as additional code accessing attacker-provided input, could be used to identify relevant sequences. Unfortunately, it is not possible to use dynamic analysis inside of the TEE to investigate the patches handling attacker-controlled input, because access is usually locked down by vendors. After finding a vulnerability, an analyst needs many parameters from the address space to replicate the exploit. However, the layout of the address space (*i.e.,* the location where code and data are mapped to), which is necessary for the replication, is not publicly disclosed.

In this work, we present our insights and techniques to face these challenges. We studied CVE-2016-8764 using manual analysis guided by binary-diffing and performed a dynamic analysis on the device, treating the TEE as a black-box. We were successful in replicating Stephens' exploit and gained insights into Huawei's TEE, Trusted Core (TC). Using this exploit, we acquired the address space layout of the targeted TA. Next, leveraging the runtime parameters observed from the device, we implemented an angr-based [21] prototype, SimTA, capable of emulating the execution environment. SimTA achieves a runtime behavior that is close to the normal execution of the TA on the device.

In addition to having an execution environment for the targeted TA, SimTA allows us to annotate the attacker-controlled input, thus, permitting us to filter patches dealing with attacker-controlled input from the binary-diff. Furthermore, we can even selectively introduce symbolic inputs to better understand the constraints introduced by a patch. As a result, we found a previously unknown 1-day heap-overflow vulnerability, an authentication bypass, and the already known type-confusion vulnerability underlying CVE-2016-8764. We elaborate on the analyses that led to these findings in our evaluation.

In summary, our contributions are the following:

- We share our insights for the interfaces, the abstraction layers, and the address space layout of one TA for the TC TEE. In order to get access to TEE internals and examine the runtime parameters of the TA, we implement and use an exploit for CVE-2016-8764 to collect the information from a real device.

Workshop on Binary Analysis Research (BAR) 2020
23 February 2020, San Diego, CA, USA
ISBN 1-891562-62-2
https://dx.doi.org/10.14722/bar.2020.23014
www.ndss-symposium.org

Previous Works

- **Finding 1-Day Vulnerabilities in Trusted Applications using Selective Symbolic Execution (NDSS 2020)**
  - emulates TA execution environemnts using selective symbolic execution.
  - necessitates a "patched" version of TA, resulting in limitations during production testing.
- **TEEFuzzer: A fuzzing framework for trusted execution environments with heuristic seed mutation (FGCS 2023)**
  - collect code coverage by instrumenting the trusted OS.
  - not feasible when TA developers are restricted to obtain such permissions.

Finding 1-Day Vulnerabilities in Trusted Applications using Selective Symbolic Execution

Marcel Busch and Kalle Dirsch
{marcel.busch, kalle.dirsch}@fau.de
IT Security Infrastructures Lab
Department of Computer Science
Friedrich-Alexander University Erlangen-Nürnberg (FAU)

Abstract—Trusted Execution Environments (TEEs) constitute a major building block for modern mobile devices' security

version. One problem that can arise while extracting the patch is that not only the vulnerable sequence of instructions appears

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

TEEFuzzer: A fuzzing framework for trusted execution environments with heuristic seed mutation

Guoyun Duan [a,b], Yuanzhi Fu [a], Boyang Zhang [a], Peiyao Deng [a], Jianhua Sun [a], Hao Chen [a], Zhiwen Chen [a,c,*]

[a] CSEE of Hunan University, No. 2 Lushan South Road, Changsha 410082, China
[b] Information and Network Center, Hunan University of Science and Engineering, Yongzhou Hunan, 425199, China
[c] School of Computer Science & School of Cyberspace Science, Xiangtan University, Xiangtan Hunan, 411105, China

ARTICLE INFO

Article history:
Received 18 October 2022
Received in revised form 28 February 2023
Accepted 4 March 2023
Available online 7 March 2023

Keywords:
Fuzzing
Trusted computing
Particle swarm
Heuristic seed mutation
TrustZone
Trusted execution environment

ABSTRACT

With the rapid development of the Internet, data security faces new challenges. As a bridge between the underlying hardware and upper layer applications, the operating system plays a critical role in securing sensitive data. The trusted execution environments (TEEs) are special operating systems aiming at preventing the illegal access and tampering of sensitive data. Thus, TEEs have much stricter security requirements than normal operating systems. Fuzzing is a promising technique that is widely used to identify vulnerabilities in operating systems and applications. However, existing fuzzing frameworks are not directly applicable to TEE-enabled devices due to the specific architecture of TEE-based systems.

In this paper, we present the design and implementation of a coverage-guided fuzzing framework for trusted execution environments. Specifically, we build TEEFuzzer, a system that can perform fuzz testing for the open portable trusted executive environment (OP-TEE), which is a widespread TrustZone operating system. Our system contains several purpose-build components, which include a seed generation module and a heuristic seed mutation module to achieve higher coverage, a coverage collection module, and an automatic bug-reproducing module to improve efficiency. With extensive evaluations, 38 crashes have been triggered in OP-TEE. In terms of performance, the average execution speed of TEEFuzzer is 79.4 test cases per second. In summary, we show that fuzzing is a feasible and effective approach to testing trusted execution environments.

© 2023 Elsevier B.V. All rights reserved.

1. Introduction

The rapid development of intelligent terminals and the popularization of the Internet of Things (IoTs) have put forward higher requirements for operating systems (OS) [1]. Besides simplicity and low performance overhead [2], these devices urgently require the operating systems to have the capability of providing high security protection for critical systems. By deploying a tamper-resistant security chip as the trusted root of the system [3], and constructing important operational steps or processes in the system into a chain of trust, we can obtain a security subsystem that can ensure the security of critical systems and user data. Trusted Execution Environment (TEE) proposed by Global Platform (GP) is a critical security component in many systems to guarantee the integrity and confidentiality of applications [4]. With the help of dedicated hardware, TEEs can execute security sensitive applications, like cryptographic key management and attestation, in protected domains isolated from the normal OS that coexists with TEEs. AMD, Intel, Google, Apple, Qualcomm, and other vendors and device manufacturers have added TEE modules to their products to enhance the security of their products [5,6]. ARM's TrustZone [7,8] is not only the hardware technology for implementing TEE in mobile environments, but also the security foundation for Android smartphones and IoT devices. In addition, it is widely deployed in servers [9] and low-end devices [10]. It is expected that trillions of TrustZone-enabled devices will be available in the market in the future.

It is well acknowledged that TrustZone-based TEEs are more secure than normal OSes because of the hardware-enforced separate execution environment and smaller Trusted Computing Base (TCB). Thus, many systems rely on TEEs to protect them from

Previous Works

- **PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation (USENIX 2020)**
  - emulate necessary HW & SW components for four widely-used TrustZone TEEs.
  - undisclosed due to industry involvements.

PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation

Lee Harrison*[1], Hayawardh Vijayakumar*[1], Rohan Padhye[2], Koushik Sen[2], and Michael Grace[1]

[1]Samsung Knox, Samsung Research America
{lee.harrison,h.vijayakuma,m1.grace}@samsung.com
[2]EECS Department, University of California, Berkeley
{rohanpadhye,ksen}@cs.berkeley.edu

Abstract

ARM's TrustZone technology is the basis for security of billions of devices worldwide, including Android smartphones and IoT devices. Because TrustZone has access to sensitive information such as cryptographic keys, access to TrustZone has been locked down on real-world devices: only code that is authenticated by a trusted party can run in TrustZone. A side-effect is that TrustZone software cannot be instrumented or monitored. Thus, recent advances in dynamic analysis techniques such as feedback-driven fuzz testing have not been applied to TrustZone software.

To address the above problem, this work builds an emulator that runs four widely-used, real-world TrustZone operating systems (TZOSes) - Qualcomm's QSEE, Trustonic's Kinibi, Samsung's TEEGRIS, and Linaro's OP-TEE - and the trusted applications (TAs) that run on them. The traditional challenge for this approach is that the emulation effort required is often impractical. However, we find that TZOSes depend only on a limited subset of hardware and software components. By carefully choosing a subset of components to emulate, we find we are able to make the effort practical. We implement our emulation on PARTEMU, a modular framework we develop on QEMU and PANDA. We show the utility of PARTEMU by integrating feedback-driven fuzz-testing using AFL and use it to perform a large-scale study of 194 unique TAs from 12 different Android smartphone vendors and a leading IoT vendor, finding previously unknown vulnerabilities in 48 TAs, several of which are exploitable. We identify patterns of developer mistakes unique to TrustZone development that cause some of these vulnerabilities, highlighting the need for TrustZone-specific developer education. We also demonstrate using PARTEMU to test the QSEE TZOS itself, finding crashes in code paths that would not normally be exercised on a real device. Our work shows that dynamic analysis of real-world TrustZone software through emulation is both feasible and beneficial.

* These authors contributed equally to this work.

1 Introduction

ARM's TrustZone technology [2] is the basis for security of billions of devices worldwide, including Android smartphones [51,54] and IoT devices [55]. TrustZone provides two isolated environments: a rich execution environment (REE or "normal world") for running normal applications, and a trusted execution environment (TEE or "secure world") for running trusted applications. Only the secure world has access to sensitive data such as cryptographic keys and biometrics information. The secure world runs security-critical "trusted applications" (TAs) for cryptographic key management, attestation [41], device integrity maintenance [4], and authentication on top of a TrustZone operating system (TZOS). It is the responsibility of the TAs and TZOS to protect access to such sensitive data even if the normal world is fully compromised, for example, due to malicious apps or users who "root" their smartphones [63]. A vulnerability in a TA or the TZOS leads to a breakdown of this protection. Therefore, it is critical to be able to analyze the security of TrustZone software.

In spite of TrustZone software's importance to security, dynamic analysis of real-world TrustZone software is limited by TrustZone's locked-down nature. In real-world TrustZone deployments, only code that is authenticated (i.e., signed) by a trusted party can run. This restriction maintains the security of data accessible only by the secure world. However, it comes at a cost: the inability to instrument or monitor code in the secure world. This rules out applying dynamic analysis techniques such as feedback-driven fuzz testing [9,12,40,61], concolic execution [13,48], taint analysis [17,58], or debugging, on TrustZone software on real devices.

As a result, approaches to analyze real-world TrustZone software have been limited. Approaches to find TA vulnerabilities include static reverse-engineering of binaries [7,8] and blind fuzzing without feedback [6] on real devices. Approaches that attempt to emulate software by forwarding requests to real hardware [28,31,49,59] through interfaces such as JTAG or USB are not applicable, since TrustZone hardware does not export such interfaces and its software is

Previous Works

- **PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation (USENIX 2020)**
  - emulate necessary HW & SW components for four widely-used TrustZone TEEs.
  - undisclosed due to industry involvements.
- **TEEzz: Fuzzing Trusted Applications on COTS Android Devices (S&P 2023)**
  - black-box fuzzing with type and state inference.
  - only provides a limited view of the target.

PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation

Lee Harrison*[1], Hayawardh Vijayakumar*[1], Rohan Padhye[2], Koushik Sen[2], and Michael Grace[1]

[1]Samsung Knox, Samsung Research America
{lee.harrison,h.vijayakuma,m1.grace}@samsung.com
[2]EECS Department, University of California, Berkeley

TEEzz: Fuzzing Trusted Applications on COTS Android Devices

Marcel Busch, EPFL; Aravind Machiry, Purdue University; Chad Spensky, Allthenticate; Giovanni Vigna, UC Santa Barbara; Christopher Kruegel, UC Santa Barbara; Mathias Payer, EPFL

***Abstract*—Security and privacy-sensitive smartphone applications use trusted execution environments (TEEs) to protect sensitive operations from malicious code. By design, TEEs have privileged access to the entire system but expose little to no insight into their inner workings. Moreover, real-world TEEs enforce strict format and protocol interactions when communicating with trusted applications (TAs), which prohibits effective automated testing.**

TEEzz is the first TEE-aware fuzzing framework capable of effectively fuzzing TAs *in situ* on production smartphones, i.e., the TA runs in the encrypted and protected TEE and the fuzzer may only observe interactions with the TA but has no control over the TA's code or data. Unlike traditional fuzzing techniques, which monitor the execution of a program being fuzzed and view its memory after a crash, TEEzz only requires a limited view of the target. TEEzz overcomes key limitations of TEE fuzzing (e.g., lack of visibility into the executed TAs, proprietary exchange formats, and value dependencies of interactions) by automatically attempting to infer the field types and message dependencies of the TA API through its interactions, designing state- and type-aware fuzzing mutators, and creating an in situ, on-device fuzzer.

Due to the limited availability of systematic fuzzing research for TAs on commercial-off-the-shelf (COTS) Android devices, we extensively examine existing solutions, explore their limitations, and demonstrate how TEEzz improves the state-of-the-art. First, we show that general-purpose kernel driver fuzzers are ineffective for fuzzing TAs. Then, we establish a baseline for fuzzing TAs using a ground-truth experiment. We show that TEEzz outperforms other blackbox fuzzers, can improve greybox approaches (if TAs source code is available), and even outperforms greybox approaches for stateful targets. We found 13 previously unknown bugs in the latest versions of OPTEE TAs in total, out of which TEEzz is the only fuzzer to trigger three. We also ran TEEzz on popular phones and found 40 unique bugs for which one CVE was assigned so far.

***Index Terms*—Fuzzing, Android, TEE, ARM TrustZone**

I. Introduction

Smartphones operate on private user data and perform sensitive functionality, e.g., financial transactions [31], user

only the application, a vulnerability in a TA compromises the security of the entire system [88], potentially even the secure boot process [66].

While the security of these TAs is foundational to the security of the device, performing effective testing (e.g., fuzzing) remains an open challenge. Smartphones ship with the trusted OS (tOS) and numerous pre-installed TAs, prohibiting the normal world (e.g., Android) from inspecting their code at runtime. TA interactions are stateful and use complex proprietary message formats [39]. The entities in the secure world (TEE and TAs) are often encrypted and get decrypted in secure memory at runtime, prohibiting the use of static analysis-based vulnerability detection techniques. Dynamic analysis, i.e., fuzzing, is an effective alternative.

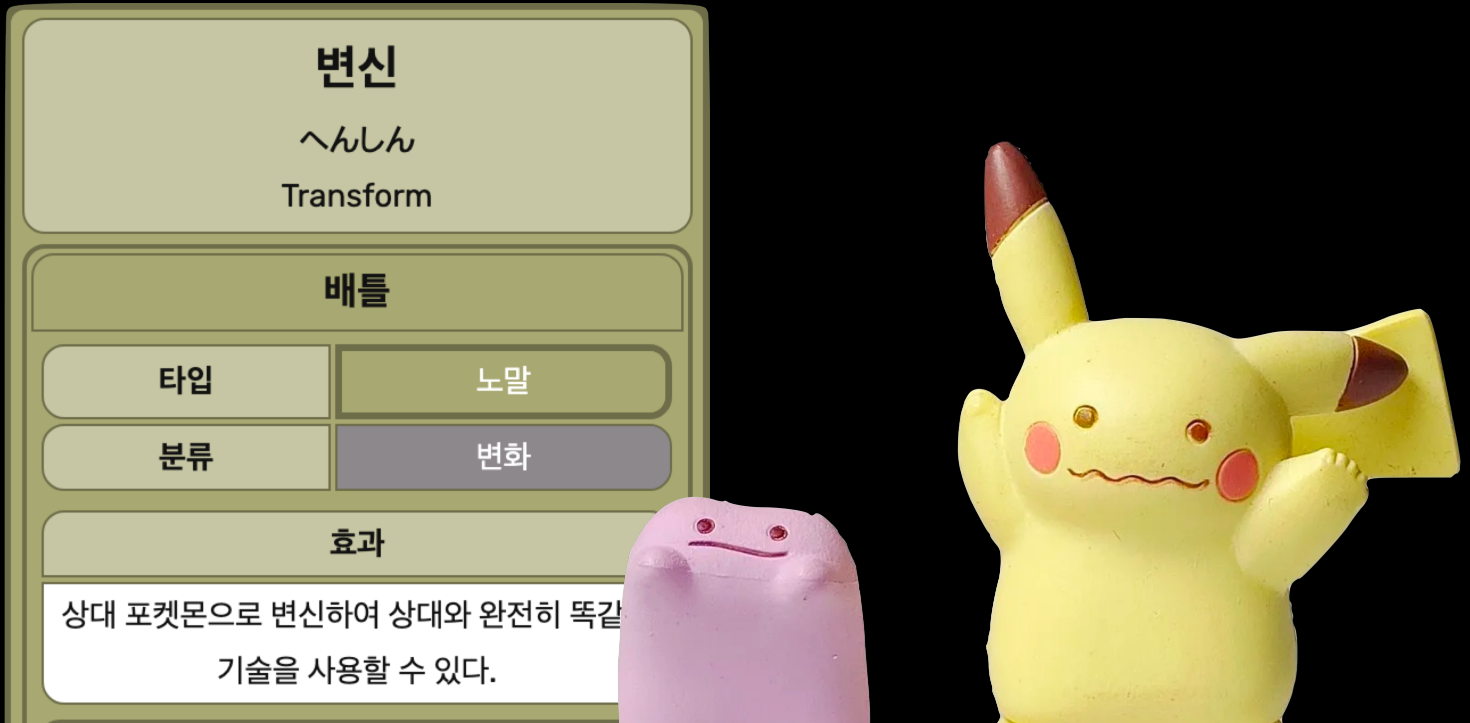
There are two principled approaches for fuzzing TAs: *rehosting through emulation* or *on-device instrumentation*.

Rehosting the TEE in an emulated environment overcomes the inaccessibility of the TEE's internal state. PartEmu [39] rehosts Samsung's proprietary TEE software stacks. They rehost the tOS and its TAs, to an emulated system-on-a-chip (SoC), gaining unrestricted access to the TEE's internal state. Limitations to this approach are (1) the reverse engineering and implementation effort for emulated software and hardware components, (2) the inaccuracy of these implementations, (3) the lack of public data sheets, and (4) industry involvement leading to non-disclosure agreements for existing solutions. Especially the last limitation deserves further emphasis. PartEmu is the only existing rehosting solution targeting multiple TEEs. The prototype validates the feasibility of rehosting proprietary software stacks deployed on Samsung devices and is not publicly available.

The second approach, on-device fuzzing, mitigates these limitations and inaccuracies of emulation approaches. However, it lacks access to the TEE's internal state and must fall back to blackbox fuzzing techniques. Unlike typical fuzzing

Our Approach

- Despite residing in the secure world, TAs are essentially just instructions.
- If we create a **duplicate** of a TA in the normal world, could we just fuzz test it with standard fuzzers? (e.g., AFL++)



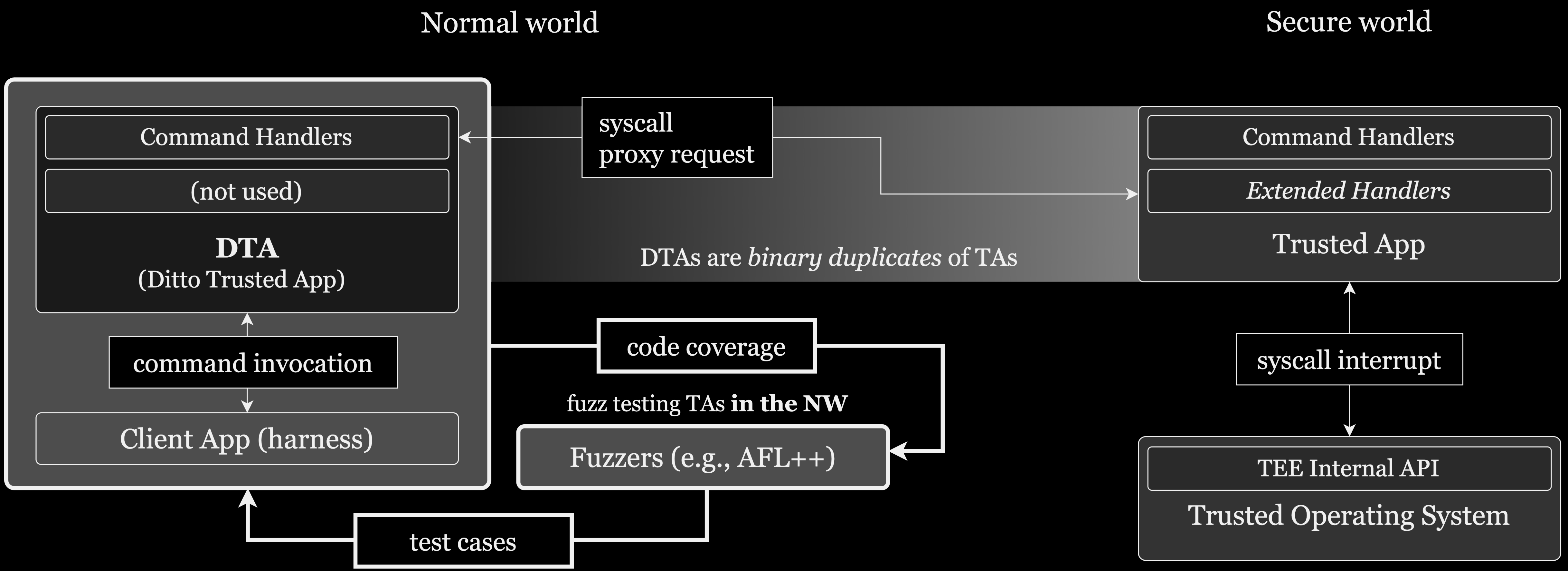


FIGURE 2. Software architecture of Ditto Trusted Applications (DTAs).

Key Challenges

- **Locating DTA**
  - The memory map of TAs must be preserved on DTAs.
- **Redirecting controls to DTA**
  - When invoking DTA command handlers, the register and memory sets should reflect the context of the original TA with precision.
- **Delegating system calls**
  - Secure world employs a completely different set of system calls.

Key Challenges

- We assume a TA vendor wants to add fuzzing to development cycle, but restricted to modify trusted OS.
- We modified TAs to include the following four **extended handlers.**

```
switch (cmd_id) {
case CMD_FETCH_TA_ADDRS:
    return func_fetch_ta_addrs(param_types, params);
case CMD_SET_DITTO_PRINTF:
    return func_set_ditto_printf(param_types, params);
case CMD_COPY_FROM_TA:
    return func_copy_from_ta(param_types, params);
case CMD_COPY_TO_TA:
    return func_copy_to_ta(param_types, params);
case CMD_PROXY_SYSCALL:
    return func_proxy_syscall(param_types, params);
```

TABLE 1. Extended handlers and corresponding usages.

Extended Handlers	Description
`fetch_ta_addrs`	Transmit runtime data from the secure world that is necessary for DTA creation, such as (1) TA page addresses, (2) session ID, (3) stack pointer (SP), (4) TA entrypoint function (`__ta_entry`) address, and (5) pseudo-shared memory address to the normal world.
`copy_to_ta`	Transfer data from the buffer in the normal world to the secure world.
`copy_from_ta`	Transfer data from the buffer in the secure world to the normal world. This handler, along with `copy_to_ta`, is used

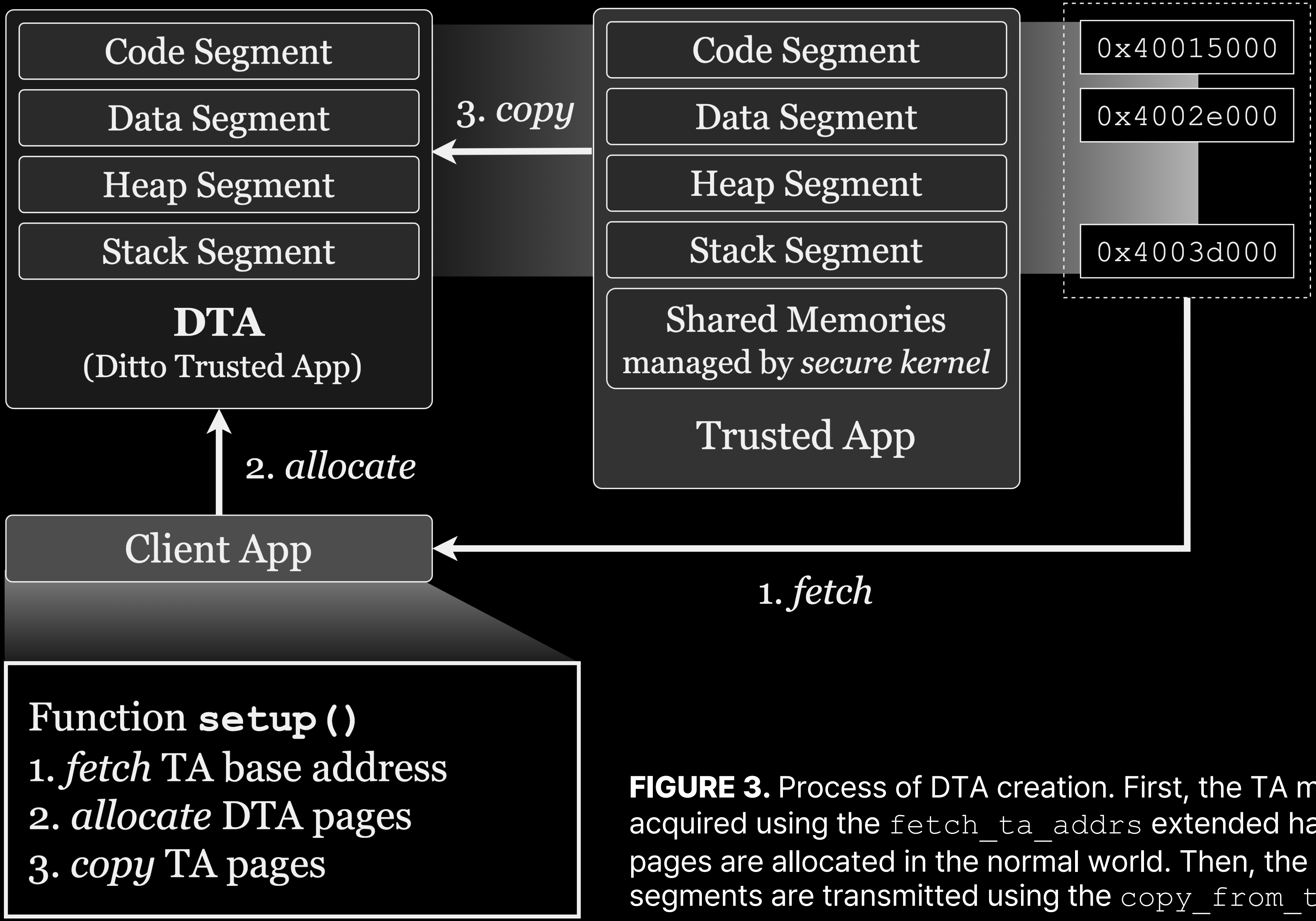


FIGURE 3. Process of DTA creation. First, the TA memory map is acquired using the `fetch_ta_addrs` extended handler, and identical pages are allocated in the normal world. Then, the contents of the TA segments are transmitted using the `copy_from_ta` extended handler.

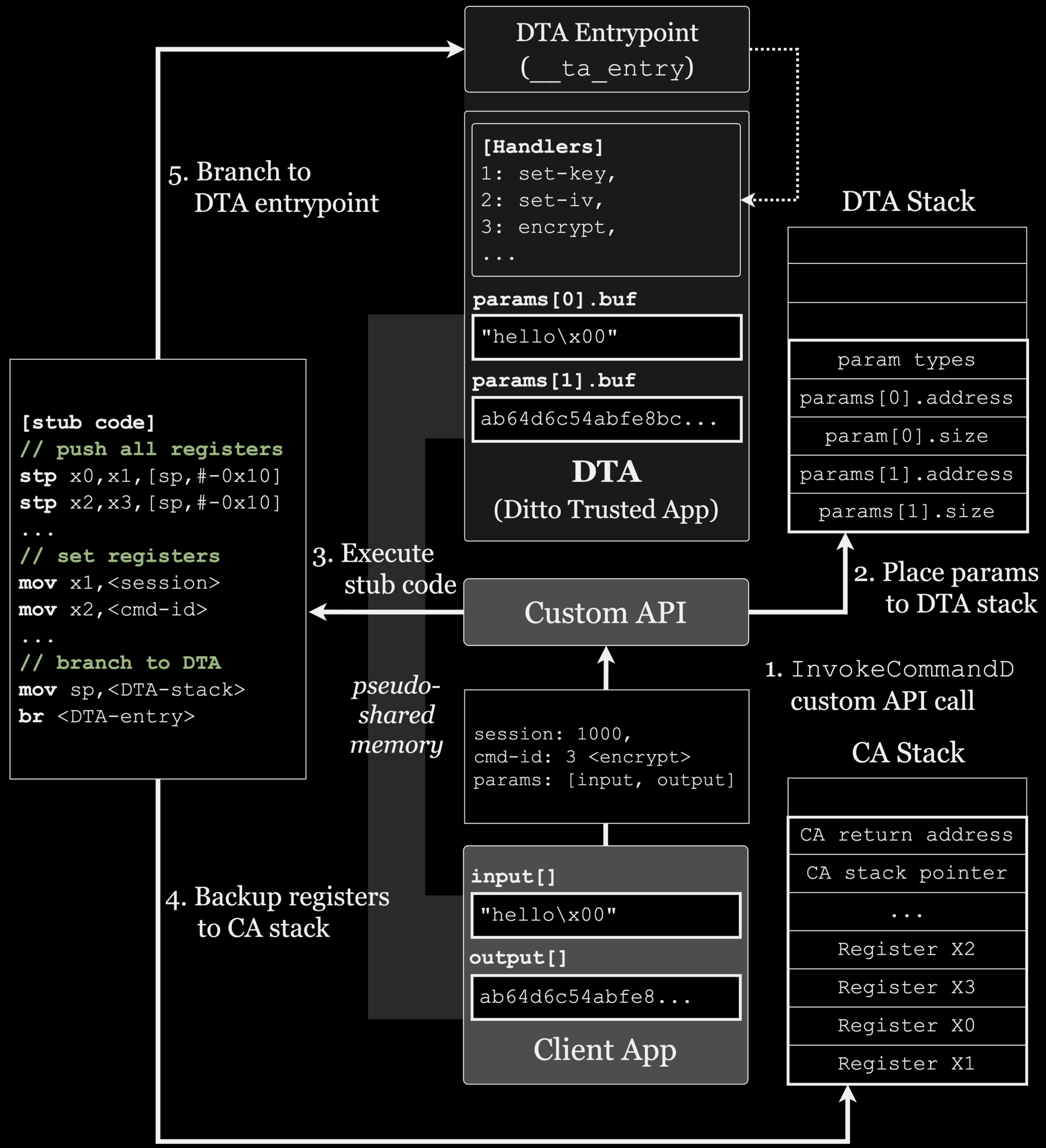


FIGURE 5. Transition of control flow during the execution of DTA command handlers.

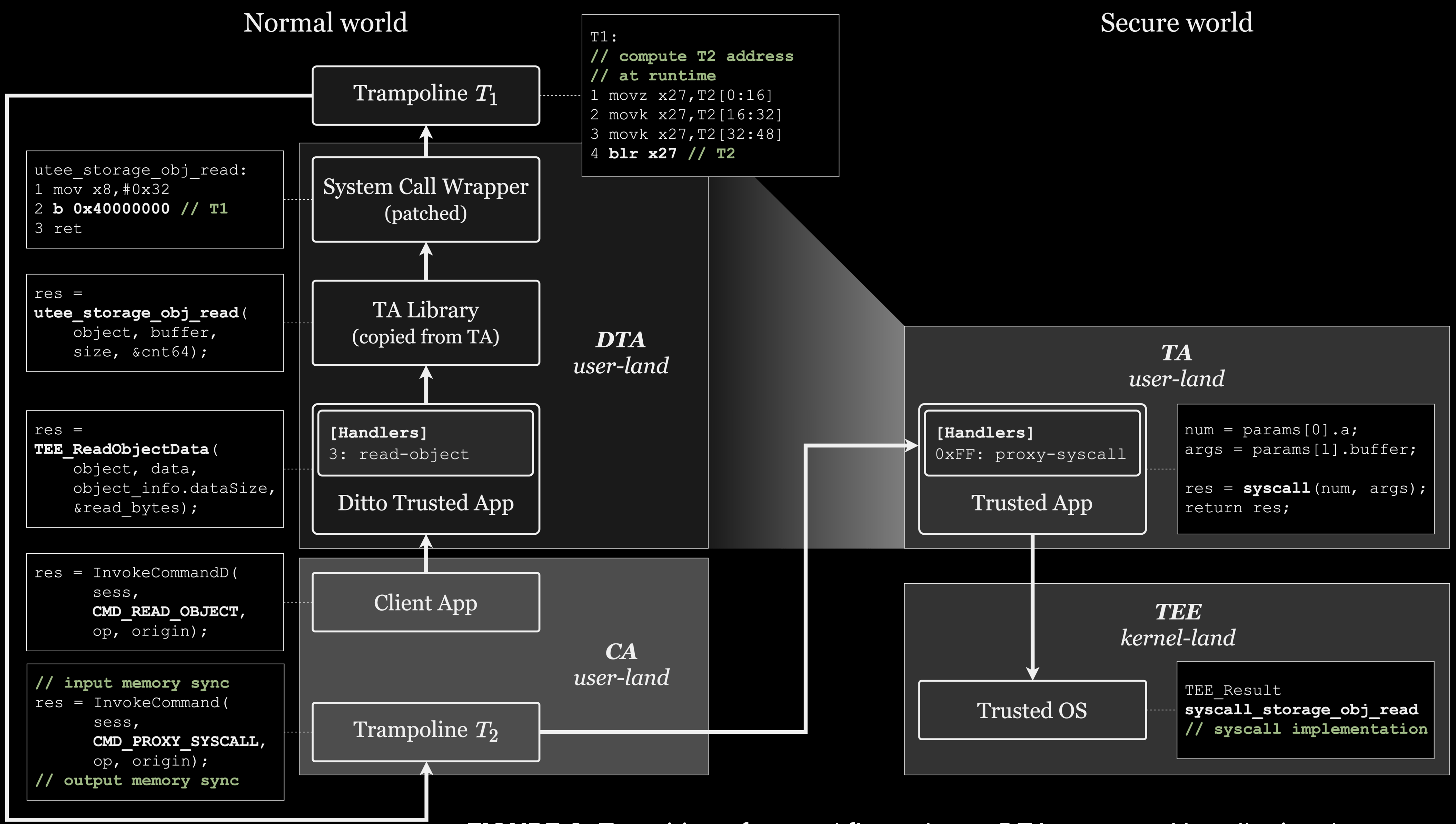


FIGURE 6. Transition of control flow when a DTA command handler invokes system calls.

Delegating System Calls

- We rewrite the system call wrappers at runtime to direct the control flow towards the trampolines.

```
utee_log:
mov x8,#0x1       // system call number
- svc #0
+ b 0x40000000    // trampoline T1
ret
...
utee_cache_operation:
mov x8,#0x46      // system call number
- svc #0
+ b 0x40000000    // trampoline T1
ret
```

```
const size_t near_addr = 0x40000000;
uint32_t *ditto_syscall_entry_addr;

ditto_syscall_entry_addr =
    mmap((void *) near_addr, PAGE_SIZE, PROT_READ
| PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
...
ditto_syscall_entry_addr[0] = 0xa9bf7bfd;
// stp x29,x30,[sp,#-0x10]!
ditto_syscall_entry_addr[1] = 0xa9bf73fb;
// stp x27,x28,[sp,#-0x10]!
ditto_syscall_entry_addr[2] = (0b110100101) << 23
| (bit0_16) << 5 | 27;
// movz x27,ditto_syscall_func_addr[0:16]
```

Evaluation

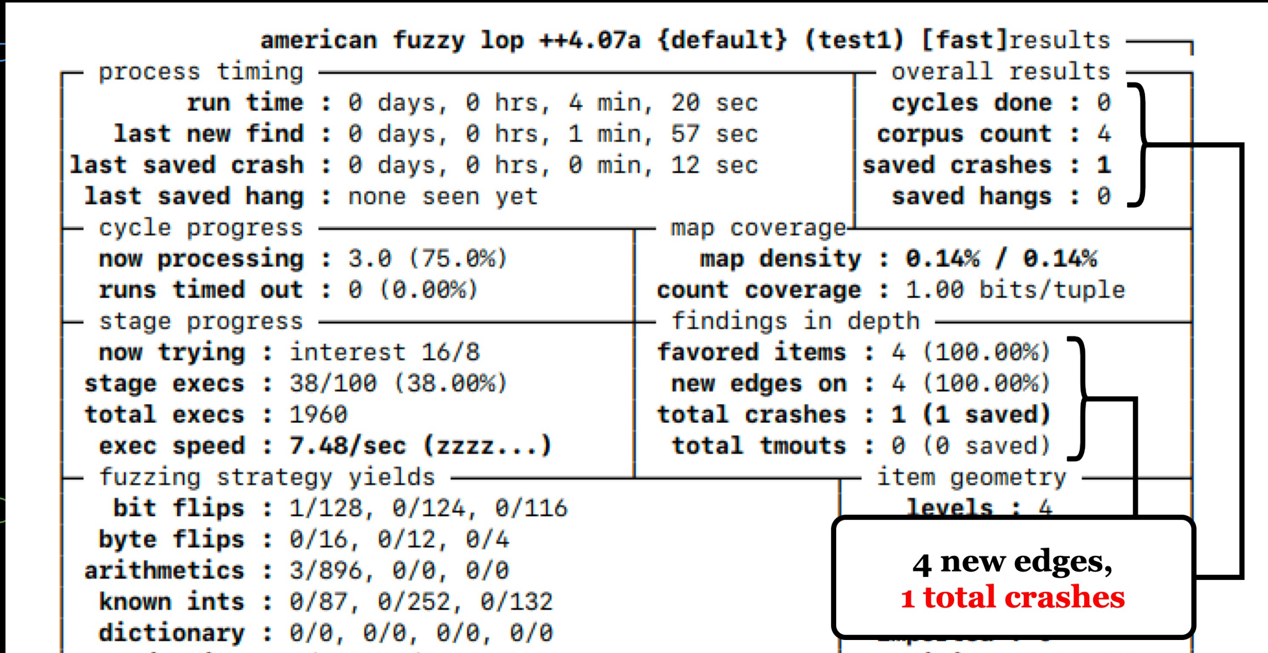
- We successfully identified vulnerable sites in a sample TA using AFL++ Frida mode.
- Additionally, we were able to visualize collected coverage data within binary analysis platforms (e.g., Lighthouse).

```
TEE_Result func_crashme(uint32_t param_types, TEE_Param params[4])
{
    ...
    if (buf[0] != 'A' && buf[0] != 'a') goto out;
    if (buf[1] != 'B' && buf[1] != 'b') goto out;
    if (buf[2] != 'C' && buf[2] != 'c') goto out;
    if (buf[3] != 'D' && buf[3] != 'd') goto out;

    int *addr = (int *) 0;
    *addr = 0xdeadbeef;      /* CRASH */
```

Evaluation

- We successfully identified vulnerable sites in a sample TA using AFL++ Frida mode.
- Additionally, we were able to visualize collected coverage data within binary analysis platforms (e.g., Lighthouse).

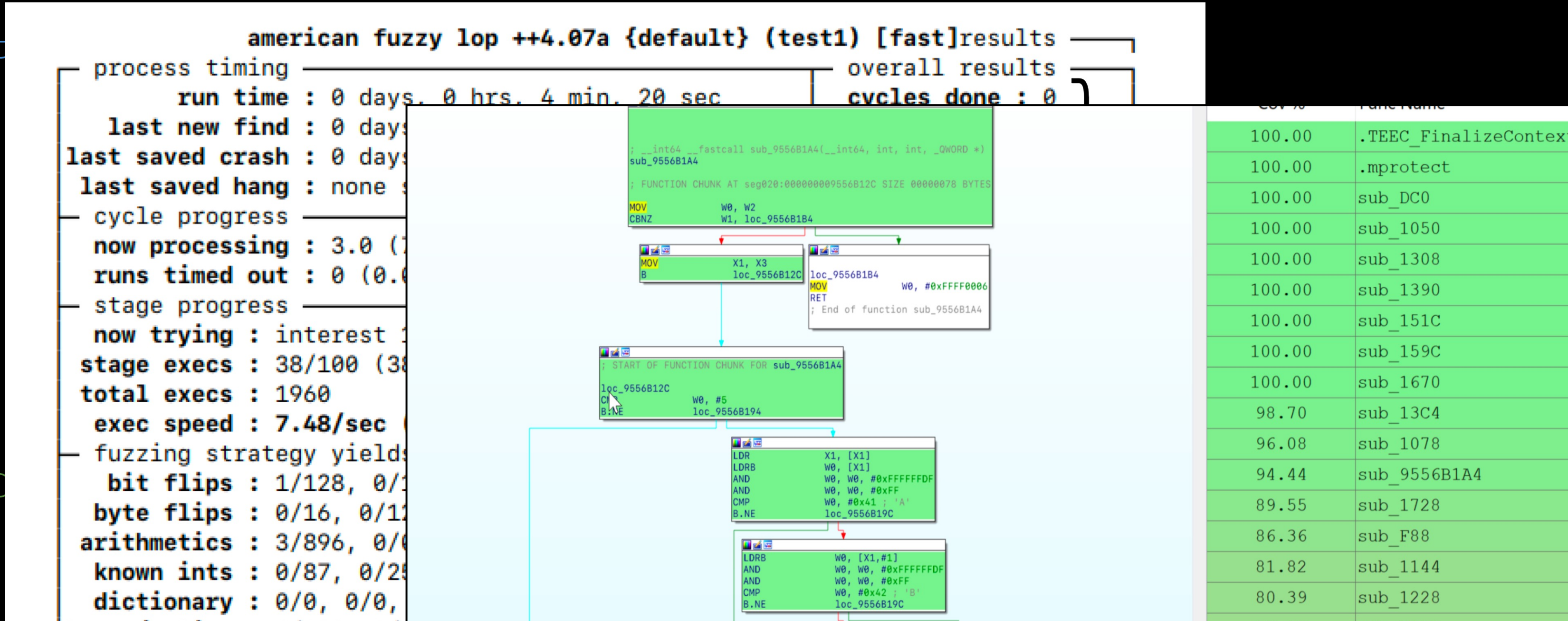
```
TEE_Result func_crashme(uint32_t
{
    ...
    if (buf[0] != 'A' && buf[0]
    if (buf[1] != 'B' && buf[1]
    if (buf[2] != 'C' && buf[2]
    if (buf[3] != 'D' && buf[3]

    int *addr = (int *) 0;
    *addr = 0xdeadbeef;     /* C
```



Evaluation

- We successfully identified vulnerable sites in a sample TA using AFL++ Frida mode.
- Additionally, we were able to visualize collected coverage data within binary analysis platforms (e.g., Lighthouse).

```
TEE_Result func_crashme(uint32_t
{
    ...
    if (buf[0] != 'A' && buf[0]
    if (buf[1] != 'B' && buf[1]
    if (buf[2] != 'C' && buf[2]
    if (buf[3] != 'D' && buf[3]

    int *addr = (int *) 0;
    *addr = 0xdeadbeef;     /* C
```



Evaluation

- The overhead of DTA comprises initialization, command invocations, and system call proxies.
- The actual overhead for system calls varied depending on the specific system call that has been invoked.

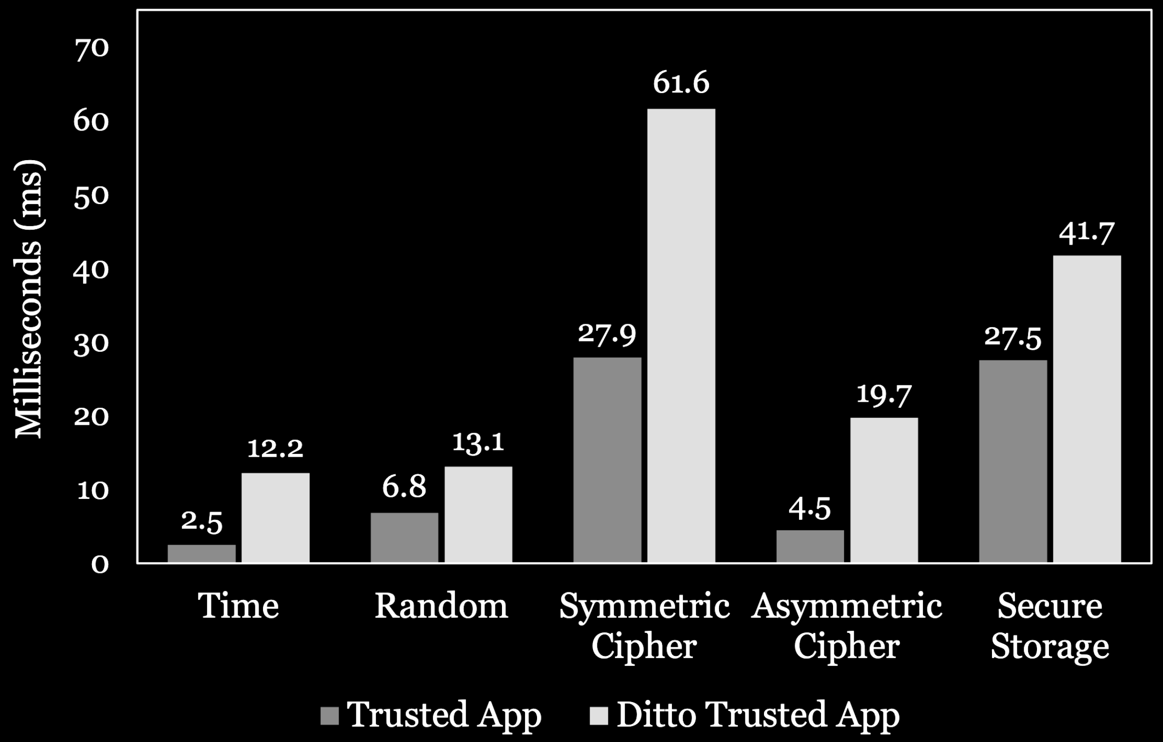


FIGURE 11. Execution time of system call sets.

Evaluation

- Fuzzing with DTA resulted in a performance gap of less than 15 exec/sec in most TA operations.

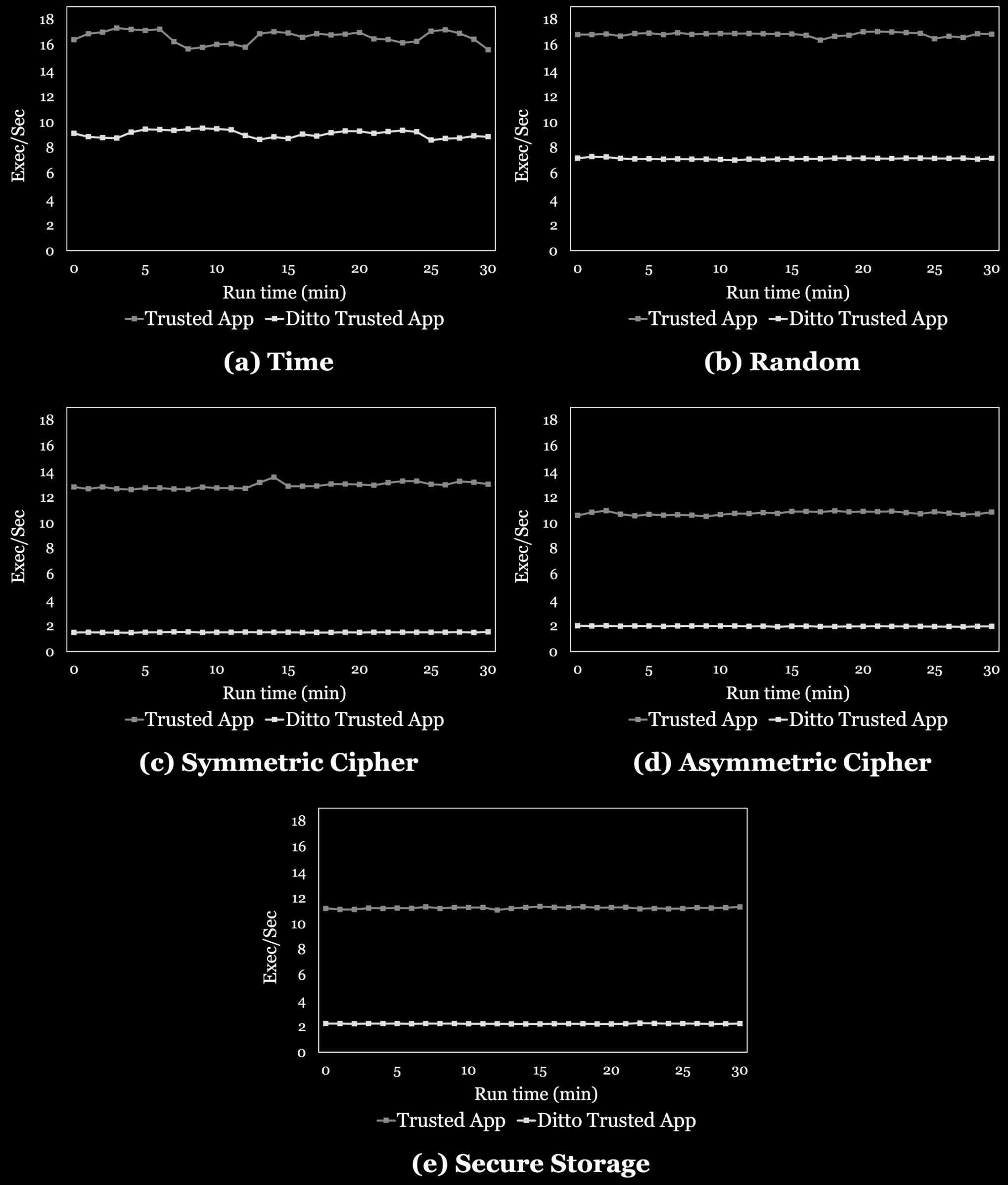
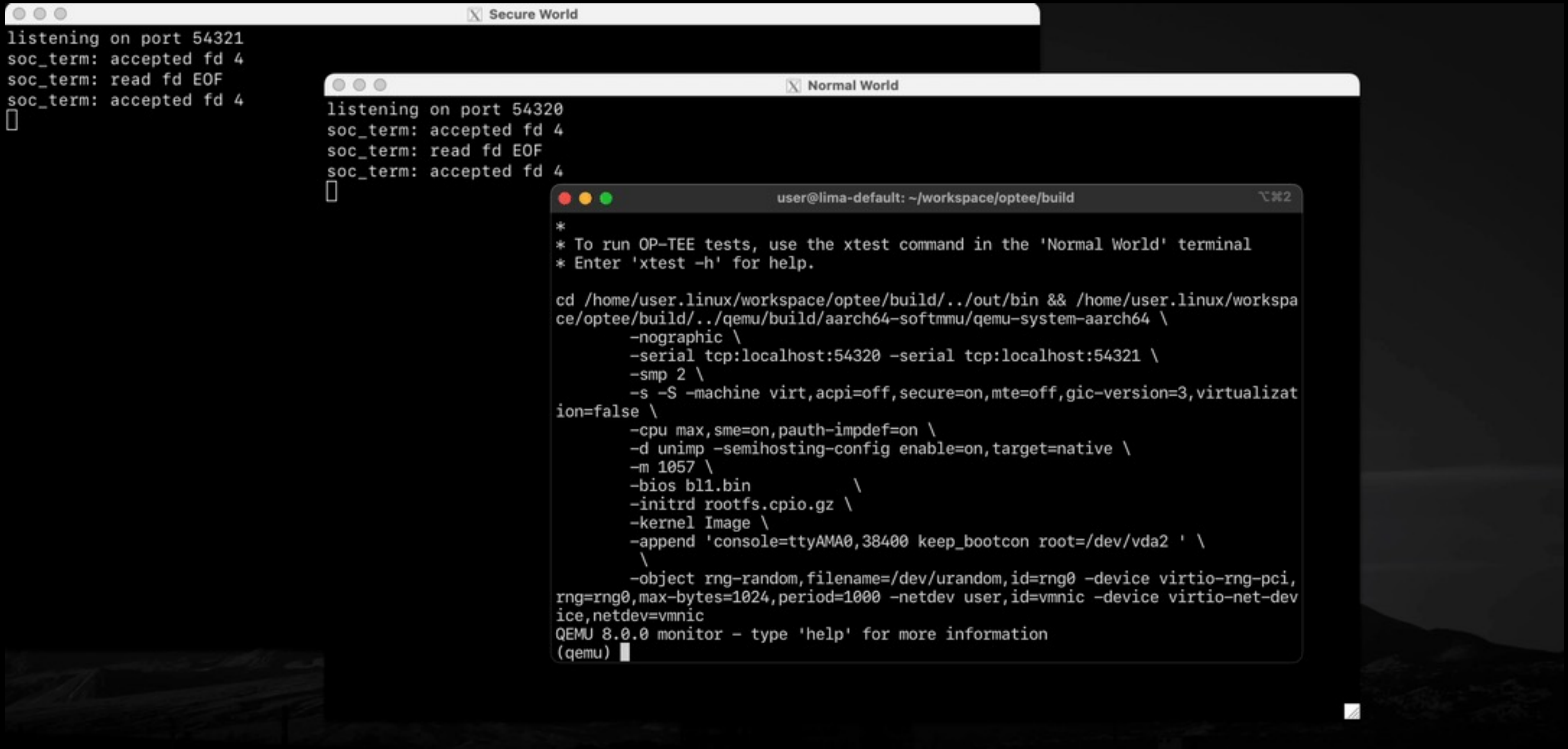


FIGURE 12. Execution rates of AFL++ measured in executions per second (exec/sec).


Secure World
listening on port 54321
soc_term: accepted fd 4
soc_term: read fd EOF
soc_term: accepted fd 4
Normal World
listening on port 54320
soc_term: accepted fd 4
soc_term: read fd EOF
soc_term: accepted fd 4
user@lima-default: ~/workspace/optee/build
*
* To run OP-TEE tests, use the xtest command in the 'Normal World' terminal
* Enter 'xtest -h' for help.
cd /home/user.linux/workspace/optee/build/../out/bin && /home/user.linux/workspace/optee/build/../qemu/build/aarch64-softmmu/qemu-system-aarch64 \
-nographic \
-serial tcp:localhost:54320 -serial tcp:localhost:54321 \
-smp 2 \
-s -S -machine virt,acpi=off,secure=on,mte=off,gic-version=3,virtualization=false \
-cpu max,sme=on,pauth-impdef=on \
-d unimp -semihosting-config enable=on,target=native \
-m 1057 \
-bios bl1.bin \
-initrd rootfs.cpio.gz \
-kernel Image \
-append 'console=ttyAMA0,38400 keep_bootcon root=/dev/vda2 ' \
\
-object rng-random,filename=/dev/urandom,id=rng0 -device virtio-rng-pci,rng=rng0,max-bytes=1024,period=1000 -netdev user,id=vmnic -device virtio-net-device,netdev=vmnic
QEMU 8.0.0 monitor - type 'help' for more information
(qemu)

Conclusion

- Current methods for fuzzing TrustZone require extensive reverse engineering and implementation efforts.
- We present DTA, a framework designed to facilitate TA fuzzing by executing TAs outside the secure world.
- We have made DTA available at https://github.com/juhyun167/dta

Conclusion

```
@article{song2024dta,
  title={DTA: Run TrustZone TAs Outside the Secure World for Security Testing},
  author={Song, Juhyun and Jo, Eunji and Kim, Jaehyu},
  journal={IEEE Access},
  year={2024},
  publisher={IEEE}
}
```

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 31 December 2023, accepted 21 January 2024, date of publication 25 January 2024, date of current version 5 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3358612

RESEARCH ARTICLE

DTA: Run TrustZone TAs Outside the Secure World for Security Testing

JUHYUN SONG[1], **EUNJI JO**[2], **AND JAEHYU KIM**[2]
[1]Department of Computer Science and Engineering, Korea University, Seoul 02841, South Korea
[2]Samsung Electronics, Hwasung-si, Gyeonggi-do 18448, South Korea
Corresponding author: Juhyun Song (thegreatsonga@korea.ac.kr)

ABSTRACT As mobile devices increasingly handle security-sensitive tasks, Trusted Execution Environments (TEEs) have become essential for providing secure enclaves. TrustZone, a popular technology for creating TEEs, allows Trusted Applications (TAs) to run with highly restricted communication interfaces. However, the isolated nature of TrustZone makes it challenging to test TA security, which is a crucial task given that TA vulnerabilities could compromise the entire system. Existing TrustZone fuzzing methods require substantial reverse engineering and implementation efforts, making them difficult to integrate into the development process. In this paper, we introduce DTA, a framework that enables the use of existing fuzzers for TA fuzzing. DTA's design includes procedures for relocating TAs outside the secure world, implementing an alternative context switch mechanism, and delegating secure world system calls to a proxy handler. Our approach has proven effective in identifying crashes in vulnerable TAs using AFL++, and we provide an evaluation of the overhead breakdown and a comparison with other methods. In conclusion, DTA offers a more comprehensive solution for incorporating fuzz testing into the TA development cycle.

INDEX TERMS Trusted application (TA), trusted execution environment (TEE), fuzzing, OP-TEE.

I. INTRODUCTION

ARM TrustZone is a security technology that has been widely deployed on billions of embedded devices around the world [1]. TrustZone security operates on the fundamental principle of separating the system into two distinct domains: the normal world and the secure world [2]. Essential security functions, including authentication, encryption, and digital rights management (DRM) are executed exclusively within the secure world. Interactions between the normal world and the secure world are permitted through restricted interfaces, enabling controlled access to the secure world and providing

implementation, and hardware itself, highlighting the need for enhancements in overall system security.

A. CHALLENGES IN TA FUZZING

Enhancing the security of TrustZone presents a unique challenge due to its structural attributes such as world separation, access control, and information blocking, which impede security analysis. Dynamic security analysis is essential for many security activities, including the identification, analysis, and mitigation of vulnerabilities [5]. Fuzzing, a technique that tests dynamic behavior of programs, has

thank you.

juhyun.a7@gmail.com