SANCTUARY: ARMing TrustZone with User-space Enclaves

Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, Emmanuel Stapf
Technische Universität Darmstadt, Germany
{ferdinand.brasser, david.gens, patrick.jauernig, ahmad.sadeghi, emmanuel.stapf}@trust.tu-darmstadt.de

***Abstract*—ARM TrustZone is one of the most widely deployed security architecture providing Trusted Execution Environments (TEEs). Unfortunately, its usage and potential benefits for application developers and end users are largely limited due to restricted deployment policies imposed by device vendors. Restriction is enforced since every Trusted App (TA) increases the TEE's attack surface: any vulnerable or malicious TA can compromise the system's security. Hence, deploying a TA requires mutual trust between device vendor and application developer, incurring high costs for both. Vendors work around this by offering interfaces to selected TEE functionalities, however, these are not sufficient to securely implement advanced mobile services like banking. Extensive discussion of Intel's SGX technology in academia and industry has unveiled the demand for an unrestricted use of TEEs, yet no comparable security architecture for mobile devices exists to this day.**

We propose SANCTUARY, the first security architecture which allows unconstrained use of TEEs in the TrustZone ecosystem without relying on virtualization. SANCTUARY enables execution of security-sensitive apps within strongly isolated compartments in TrustZone's *normal world* comparable to SGX's user-space enclaves. In particular, we leverage TrustZone's versatile Address-Space Controller available in current ARM System-on-Chip reference designs, to enforce two-way hardware-level isolation: (i) security-sensitive apps are shielded against a compromised normal-world OS, while (ii) the system is also protected from potentially malicious apps in isolated compartments. Moreover, moving security-sensitive apps from the TrustZone's *secure world* to isolated compartments minimizes the TEE's attack surface. Thus, mutual trust relationships between device vendors and developers become obsolete: the full potential of TEEs can be leveraged.

We demonstrate practicality and real-world benefits of SANCTUARY by thoroughly evaluating our prototype on a HiKey 960 development board with microbenchmarks and a use case for one-time password generation in two-factor authentication.

I. INTRODUCTION

Mobile devices have already changed our daily lives in various ways. Their success can mainly be attributed to the ecosystem that evolved around them. The increasing computing and storage capabilities, the vast number and variety of apps available on app stores and markets, as well as the connectivity to cloud services make mobile devices convenient replacements for traditional computing platforms, and the de-facto standard way of accessing the Internet [40].

Despite all benefits, today's mobile devices provide a large attack surface imposing many security and privacy challenges on their system design to be able to protect sensitive applications such as mobile banking, payments, and eID services.

The TrustZone security architecture was motivated mainly by the need for *secure mobile services* [5], when introduced in 2008 as part of an industry effort. TrustZone introduces the notion of a *normal world* and a *secure world*. While the normal world runs the Legacy OS (LOS) and user-level applications, security-sensitive applications can be executed (partially or entirely) within the secure world which represents a Trusted Execution Environment (TEE) on top of the TrustZone kernel and hardware.

Problems of TrustZone. Despite TrustZone's implementation and wide-spread deployment, TrustZone-based TEEs are mainly used by the vendors for own purposes, and hence a flourishing landscape of secure mobile services is largely missing even more than a decade after TrustZone was initially released [17]. One root cause for the lack of progress in TrustZone-based TEEs' adoption is that each installed Trusted App (TA) increases the potential for security-critical vulnerabilities, allowing attackers to exploit bugs and escalate privileges, exfiltrate private data, or gain complete control over the entire device. In practice, this means that bugs in TrustZone-enabled applications expose a large number of devices to real-world security threats, as continuously demonstrated by security researchers across device families and hardware vendors (e.g. in 2014 [15], [32], in 2015 [19], [47], in 2016 [20], [49], and in 2017 [52], [44]). Google's ProjectZero [45] recently summarized the main flaws of the current design of TrustZone as follows: it combines (i) weak isolation between TAs in the TEE, with (ii) Trusted Computing Base (TCB) expansion, and (iii) highly privileged access to the platform, making TrustZone a high-value target for attackers. Thus, vendors often aim to control and restrict access to the TEE. Thorough security assessments are needed to build a trust relationship between device vendor and app developer. Furthermore, deploying TAs to a TEE produces a large management overhead [23]. For smaller developers, the emerging costs pose a significant investment severely limiting the development of secure mobile services in practice. Device vendors try to circumvent these problems by offering some TEE functionalities, e.g. key storage, over interfaces to normal-world apps. However, this approach does not allow developers to protect own security-sensitive code and data. Hence, the provided TEE services are not sufficient to implement feature-rich secure mobile services.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23448
www.ndss-symposium.org

Existing Security Architectures. A number of ARM-based security architectures have been proposed previously [28], [10], [18]. However, they rely on virtual memory for isolation, using the same isolation mechanism proven insufficient for isolating TAs within ARM TrustZone's secure world [45]. Approaches that rely solely on *temporal isolation* – i.e., suspending the entire system to provide protection for TA execution – are not suitable for today's multi-core platforms [38], [51], since they effectively disable multitasking and parallel execution for the entire platform which imposes severe restrictions that directly affect user experience.

Goals and Contributions. Our main goal is to tackle the aforementioned problems and enable the full potential of TEEs for third-party application developers without requiring any hardware changes.

To this end, we present SANCTUARY, a novel security architecture for Trusted Execution Environments (TEEs) based on the latest ARM System-on-Chip (SoC) reference designs. SANCTUARY inherently de-privileges TrustZone-enabled apps by moving them from the secure-world TEE to an isolated normal-world compartment, thereby reducing the code base in the secure world. We call these security-sensitive apps, which are comparable to SGX's user-space enclaves, Sanctuary Apps (SAs). SANCTUARY achieves SA isolation by dynamically partitioning and re-allocating system resources: CPU cores and physical memory are temporarily reserved for the isolated compartments to execute SAs without suspending the rest of the system. In particular, we leverage TrustZone's Address-Space Controller (TZASC) to ensure a hardware-enforced, two-way isolation between SAs and all other system components. This enables an SGX-like usage of TrustZone without requiring any hardware modifications.

Building SANCTUARY comes with a number of interesting challenges: first, the Legacy OS normally assumes full control over all available CPUs. To support dynamic re-allocation of cores we have to claim, initialize, and boot individual cores dynamically at run time. Second, enforcing a strict separation between normal world, SAs, and secure world necessitates communication channels between them, e.g., to relay I/O or shared data. Third, SANCTUARY must provide security services, such as *remote attestation* and *sealing* of SAs (similar to SGX), and provide secure ways for SAs to access them. Finally, to offer tangible improvements in real-world scenarios, SANCTUARY must provide adequate performance, e.g., in authentication for mobile banking applications, without affecting user experience. Our design of SANCTUARY tackles all of these challenges to support SGX-like usage of TrustZone-enabled applications.

To summarize, our main contributions are as follows:

- We present the design of SANCTUARY, a novel security architecture building on existing TrustZone's hardware and software components while enabling enclave-like usage in the form of de-privileged normal-world execution environments that are completely isolated from the rest of the system.
- Our proof-of-concept implementation of SANCTUARY uses the HiKey 960 development board, and Linaro's open-source software OP-TEE on top of TrustZone.
- We analyze and discuss the security properties of SANCTUARY in a strong adversary setting that includes malicious SAs.
- We extensively evaluate SANCTUARY with respect to its setup and communication overhead. Additionally, we demonstrate real-world benefits of SANCTUARY in a detailed one-time password and key-generation use case for two-factor authentication, which is highly relevant for many security-sensitive applications such as mobile payment. Our results show that SANCTUARY supports low latency and does not affect user experience, hence, offering practical performance characteristics.

II. BACKGROUND

The core principle of TEEs is isolation of code and data to protect their integrity and confidentiality.

TEEs have been developed by both, academic community and industry. First, we present ARM TrustZone [5] which is available on most ARM-based systems and which is the basis for our novel security architecture SANCTUARY. Second, we explain the TrustZone Address Space Controller (TZASC) that enforces memory access control in TrustZone and plays a key role for our hardware-based isolation in SANCTUARY.

We discuss TEE research proposals as well as other related approaches in detail in Section VIII.

A. ARM TrustZone

TrustZone represents a set of security enhancements to processor designs and SoCs that are based on the ARM architecture. TrustZone enhances the processor, memory (including caches), and peripherals. A TrustZone-enabled processor can execute instructions in four different privilege levels (*Exception Levels* – EL0-EL3) and, additionally, two security modes at any given time (cf., *normal world* and *secure world* in Figure 1). To facilitate switching between normal and secure world, and to provide a clean interface, EL3 (also called *monitor mode*) runs the ARM Trusted Firmware (TF). On top of the Trusted Firmware (TF), the secure and normal world both manage their own address spaces using the remaining privilege levels for separation: EL2 is optionally used for a hypervisor, EL1 for the OS kernel, and EL0 (lowest execution privilege) is used for execution of application code.

The processor can switch from normal to secure world via an instruction called the secure monitor call (`smc`). When an `smc` instruction is invoked from normal world, the processor-core performs a context switch to the secure world (via the monitor mode) and freezes its normal-world execution. All other CPU cores of a multi-core system can independently remain in normal-world mode.

TrustZone can separate physical memory into two partitions, with one partition being exclusively accessible by the secure world. This isolation is enforced by the memory controller (TZASC), which is discussed in Section II-B. While the normal world cannot access memory assigned to the secure world, the secure world can access normal-world memory.

A device running ARM TrustZone boots up in the secure world. After the secure world finished its initial setup by

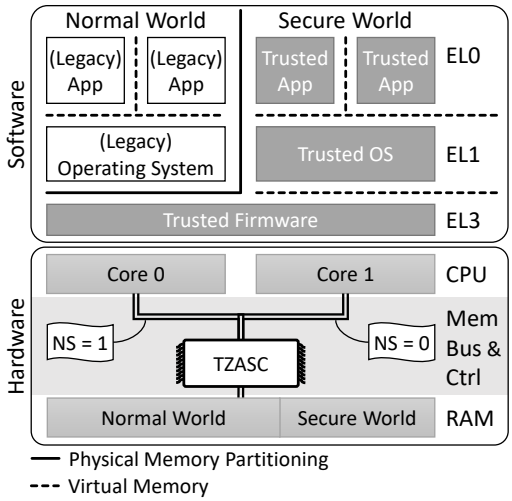

Figure 1: TrustZone software and hardware components. Software can be executed in *normal world* or in *secure world*. Isolation between these two worlds is enforced by the memory controller (TZASC) that checks for each memory access which world it originates from.

booting the Trusted OS (TOS), it switches to the normal world and boots the LOS. Most TrustZone-enabled devices are configured to use *secure boot*, i.e., the boot loader cryptographically checks the TOS prior to execution [5]. In fact, many vendors lock their devices against end-user modification via secure boot, to ensure integrity of the secure world. This allows them to make the secure world part of their TCB.

B. TrustZone Address Space Controller

With TrustZone, secure-world memory is isolated from the untrusted normal-world memory through physical memory partitioning. This is enforced in hardware by the TrustZone Address Space Controller (TZASC) which resides between the system bus and the memory chip (see Figure 1). It supports multiple memory regions and access-control settings based on several bus transaction characteristics. Originally, this only included two types of memory accesses: *non-secure* access ($\texttt{NS} = 1$), or *secure* access ($\texttt{NS} = 0$). A CPU core in *secure* mode can perform accesses of the type *secure* and *non-secure*, whereas CPU cores in *normal* mode can only perform *non-secure* accesses. The first TZASC reference implementation from ARM, the TZC-380, was published in 2010 [6]. Its successor, the TZC-400 [8], was introduced 2013 and can utilize additional characteristics of a bus transaction to separate the protected memory regions – this feature is called *identity-based filtering*. Thus, in current ARM reference designs, every device that can act as a bus master (e.g., CPU, GPU, DMA controller) is assigned a *bus-master ID* in hardware, which is appended to its memory bus transactions. This can be used to assign memory regions to specific bus masters for non-secure accesses. ARM advertises the identity-based filtering feature in context of their TrustZone Media Protection Architecture (TZMP) [4], which is used for media protection by exclusively assigning memory, e.g. the frame buffer, to the GPU.

III. Adversary Model and Requirements

A. Adversary Model

Our threat model adheres to that of TrustZone and makes the same underlying assumptions [5]. In particular, the attacker can corrupt all normal-world software, including all privilege levels up to an optional hypervisor (EL2), via remote or local software attacks. Additionally, an adversary can conduct passive physical attacks. However, the adversary cannot compromise the secure-world software and the monitor mode.

Invasive physical attacks that tamper with hardware, e.g., to inject faults at run time are out of scope. Similar to TrustZone, we do not consider Denial-of-Service (DoS) attacks,i.e., SANCTUARY does not provide availability guarantees.

Our detailed standard assumptions are derived from the related work [22], [10], [9], [12], [16], [28]:

- Applications in normal world are considered untrusted.
- The Legacy OS (LOS) in the normal world is untrusted.
- Isolation between different privilege levels is enforced by hardware through virtual memory.
- All existing architectural defenses, such as Execute Never (XN), Unprivileged Execute Never (UXN), Privileged Execute Never (PXN), and Privileged Access Never (PAN) are deployed and active.
- Secure and normal world are isolated by the TrustZone hardware extensions [5].
- Software in the secure world, including the boot loader and EL3 firmware (monitor mode), is trusted.

In this setting, SANCTUARY can be used to minimize the amount of software required in the secure world as it allows to outsource *all* Trusted Apps (TAs) to Sanctuary Apps (SAs) which execute in isolated compartments in the normal world.

B. Requirements Analysis

To enable practical and secure Sanctuary Apps (SAs) on ARM TrustZone-based platforms, a number of requirements must be fulfilled. We show that SANCTUARY fulfills these security requirements in Section VI, and demonstrate that SANCTUARY meets the functional requirements in Section VII.

1) **Code and data integrity.** The integrity of the code and data of an SA must be preserved. This can be achieved by (i) isolation during SA execution and (ii) attestation of the SA code when loaded into the isolated compartment.
2) **Data confidentiality.** Confidentiality of data processed in an SA must be preserved. This can be achieved by (i) a secure channel for provisioning the data, (ii) spatial isolation during execution, and (iii) temporal isolation to prevent that sensitive information becomes accessible after SA execution has finished.
3) **Secure channel to secure world.** An SA needs a secure channel to utilize security services provided by

the secure world. This can be realized by an *exclusive* shared memory, i.e., accessible only by the SA and the secure world but not by untrusted normal-world software.

4) **Protection from malicious SAs.** To enable unrestricted usage models for SAs, malicious SAs must be tolerated. Protecting the platform from malicious SAs can be achieved by limiting the access privileges of SAs to a minimum (i.e., EL0) and preventing them from accessing normal-world memory.
5) **Hardware-enforced resource partitioning.** To ensure strict isolation spatial *and* temporal isolation are needed.
6) **Minimal software changes.** Leveraging existing interfaces of the secure-world OS and the normal-world OS prevents extensive modification of the software stack.
7) **Positive user experience.** Assigning a single CPU core for limited time to SA execution leads to low impact on the overall system performance for most usage scenarios on todays commonly available multi-core architectures. Latency can be kept low by minimizing the SA run-time environment.

IV. SANCTUARY DESIGN

The goal of the SANCTUARY architecture is to enable secure and widespread use of Trusted Execution Environments (TEEs) (e.g., through third-party developers) on ARM based devices. SANCTUARY allows the creation of multiple parallel isolated compartments on ARM devices in the normal world which are strictly isolated from the LOS and Legacy Apps (LAs). The isolated compartments, which we call SANCTUARY Instances, run security-sensitive apps called Sanctuary Apps (SAs). Every SANCTUARY Instance executes only one SA at a time. Since all SANCTUARY instances are independent and separated from each other, also the SAs become strongly isolated. Additionally, all SANCTUARY instances are isolated from the existing TrustZone secure world.

Spatial isolation of a SANCTUARY Instance is achieved by (i) partitioning the physical memory using the TZC-400 memory controller, (ii) dedicating a CPU core to the SANCTUARY Instance, and (iii) excluding the SANCTUARY Instance's memory from shared caches. Temporal isolation is ensured by launching the SANCTUARY CPU core from a trustworthy state (ARM Trusted Firmware (TF)) and erasing all sensitive information from memory and caches before it exits.

We designed SANCTUARY in such a way that the required changes to the existing software ecosystem are *minimal*: in fact, SANCTUARY can extend existing TEE architectures without affecting the functionality of already deployed software in both the normal world and the secure world.

Figure 2 shows an abstract view of SANCTUARY's design. In the following, we describe SANCTUARY's isolation mechanism, its initialization, and its security services.

A. SANCTUARY *Isolation*

In addition to the existing security boundary between TrustZone's secure world and normal world, SANCTUARY enables isolation *within* the normal world. A dedicated memory region

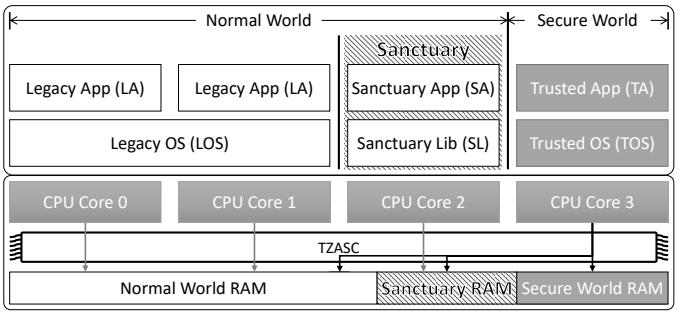

Figure 2: SANCTUARY design overview. Within the normal world, one core is reserved for SANCTUARY. The TCB, marked in gray, includes the hardware and the secure-world software that is involved in the initialization of an SA.

is made exclusively accessible by one CPU core by leveraging ARM's new memory access controller TZC-400. Details on how the controller needs to be configured to achieve this physical memory partitioning are given in Section V-E. As a result, all software executing on that CPU core is protected from untrusted software executing on the remaining CPU cores of the system. In Figure 2, CPU core 2 running a SANCTUARY Instance is configured to have exclusive access to the SANCTUARY *RAM* partition, as depicted by the arrows. The untrusted normal-world software – executing on CPU cores 0 and 1 – can only access the normal world memory. Furthermore, the CPU core assigned to the SANCTUARY Instance is *not* allowed to access normal-world memory, achieving a two-way isolation which allows SANCTUARY to tolerate potentially malicious SAs. However, SANCTUARY does support shared memory between normal world and SA for efficient communication as well as shared memory between secure world and SA to establish a secure channel. This enables scenarios like secure UI over TAs. SANCTUARY's handling of shared memory is explained in detail in Section V-E.

The secure-world software is trusted and therefore allowed to access all memory, including normal-world memory, SANCTUARY memory, and secure-world memory (black arrows in Figure 2).

Multi-SA isolation: SANCTUARY instances are either executed consecutively on the same CPU core, or execute on separate, mutually isolated cores with dedicated memory partitions. After SA execution finished, the system returns to its original state (see Section IV-B) and the next SANCTUARY instance can be launched. This ensures strong isolation between SAs: all SAs are executed completely independently of each other.

Privilege isolation: SAs are limited to execute in user-mode. The privileged mode of a CPU core used by SANCTUARY is occupied by the Sanctuary Library (SL). Important to note is that the SL is *not* part of the TCB, but instead is only needed to provide two main functionalities: (i) initializing an execution environment for the SA, and (ii) providing service interfaces to the SA, e.g., for accessing SANCTUARY's security services.

B. SANCTUARY *Initialization*

SANCTUARY's isolation does protect the integrity and confidentiality of an SA while it is executing on the dedicated CPU

core. However, since the SA code is loaded by the untrusted LOS, its integrity must be verified. The initialization process of SANCTUARY provides the necessary verification mechanism.

For better resource utilization, SANCTUARY does not dedicate one CPU core for executing SAs permanently. If a new SANCTUARY instance is created, one CPU core is shut down and removed from the resources available to the LOS executing in the normal world. All remaining CPU cores stay under control of the LOS. Hence, the LOS can continue execution of normal-world tasks preserving the system's availability, i.e., the user does not notice negative effects from the creation of a SANCTUARY Instance and the execution of an SA.

Next, the code to be executed on the SANCTUARY core, i.e., SL and SA, is loaded into a separate memory section. After the memory isolation has been activated, the loaded code is validated using digital signatures. The signature for the SL is provided by the device vendor, whereas the signature for the SA is provided by the SA developer. The detailed verification process is described in Section V. After a successful verification, the dedicated CPU core is restarted. The SANCTUARY core starts from a defined initial state, boots the SL and executes the SA.

After an SA has finished, the dedicated core removes all information from the memory, invalidates all cached data, and shuts down. The isolation for the wiped memory is deactivated, making the memory available to the LOS again. The CPU core is restarted and reassigned to the LOS.

C. SANCTUARY *Security Services*

The initial content of an SA is loaded from unprotected memory, hence, it can be manipulated and cannot contain confidential data. Therefore, SANCTUARY needs to provide a mechanism to provision confidential data to an SA over a secure channel *after* it has been created. However, to ensure that secret data is not sent to a malicious (or maliciously modified) SA, the integrity and authenticity of an SA needs to be verified before provisioning secret data. To enable secure provisioning of secret data to an SA and secure storage of secret data, SANCTUARY provides a set of security services implemented as TAs supplied by the device vendor (called vendor TAs throughout the remaining paper). These TAs run within the secure-world Trusted OS (TOS) (see Figure 2).

Remote attestation allows an SA to establish a secure channel to an external entity. Through the platform identity feature of TrustZone, the integrity measurement of SANCTUARY can be authentically reported to a third party. Linking the authentic integrity report with the establishment of a secure channel to the SA creates a secure and authenticated channel through which confidential data can be provisioned.

Sealing allows SAs to store sensitive data such that only instances of the originating SA can accesses the data. SANCTUARY provides each SA with a unique encryption key that is derived from the hash value computed over the SA binary. The key can be used to encrypt data, e.g., before writing it to persistent storage.

Further security services, like monotonic counters, secure timers, secure randomness, etc. can be provided by TrustZone's secure world, as well. Similar security services are commonly available in commercial TEE implementations, for instance Intel SGX [31], [39], [25], [2] and can be implemented similarly in SANCTUARY. In addition, secure user interfaces for SAs can easily be provided by TAs, as secure I/O is already provided by TrustZone.

D. SANCTUARY *Software Model*

With SANCTUARY, every application developer is able to utilize TEE functionalities, i.e., every developer can deploy an SA. Each SA belongs to an untrusted LA. This allows straightforward deployment through existing app markets: SAs come as part of LAs using the standard installation routine.

Additionally, by coupling each SA with an LA, the functionalities of the SL can be minimized. In particular, the LA acts as a proxy and allows the SA to make use of all functionalities provided by the LOS, like file system access. The LA and SA can efficiently exchange information and interact with each other via shared memory. When an SA wants to provide sensitive data to the LA, e.g., for persistent storage, the SA can use the sealing service (see Section IV-C) to encrypt the data before sending it to the LA.

How to partition an application into security-critical and uncritical parts is an orthogonal problem.

V. IMPLEMENTATION

System Setup. We implemented SANCTUARY on a HiKey 960 development board, as it provides a recent ARMv8 SoC design that is commonly used on modern mobile devices. Moreover, the HiKey 960 is one of the few development boards which gives developers the possibility to deploy own software in the secure world. The HiKey 960 is based on an octa-core ARM big.LITTLE processor architecture with four ARM Cortex-A73 and four Cortex-A53 cores.

SANCTUARY Software Components. An overview of our SANCTUARY implementation is shown in Figure 3. For the secure-world Trusted OS (TOS) we use OP-TEE [1] which currently is the most developed open-source TOS. The SANCTUARY design is not limited to a particular TOS and can also be implemented using a TOS which provides a less rich feature set. OP-TEE comes bundled with a recent Linux distribution which we use as the normal-world Legacy OS (LOS). We implement a custom kernel module (KM) as part of the LOS which manages the SANCTUARY Instances from the normal world. In OP-TEE, we implement two vendor Trusted Apps (TAs), the *Proxy TA* and the *Sealing TA*. They provide the basic security services for SANCTUARY, namely remote attestation and sealing. A SANCTUARY Instance consists of the Sanctuary Library (SL) and a SA. In our prototype implementation, we use the Zircon micro kernel [24] as the basis for our SL. Besides adding two vendor TAs, we only make small one-time changes to the Trusted Computing Base (TCB), i.e. OP-TEE and the ARM TF. The custom Static Trusted App (STA) which we add to OP-TEE manages the SANCTUARY Instances from the secure world. The Lines of Code (LOC) added to the TCB add up to 1313. The two vendor TAs make up more than half of the added lines. In total however, the TCB gets reduced because all TAs from third-party developers are removed from the secure world.

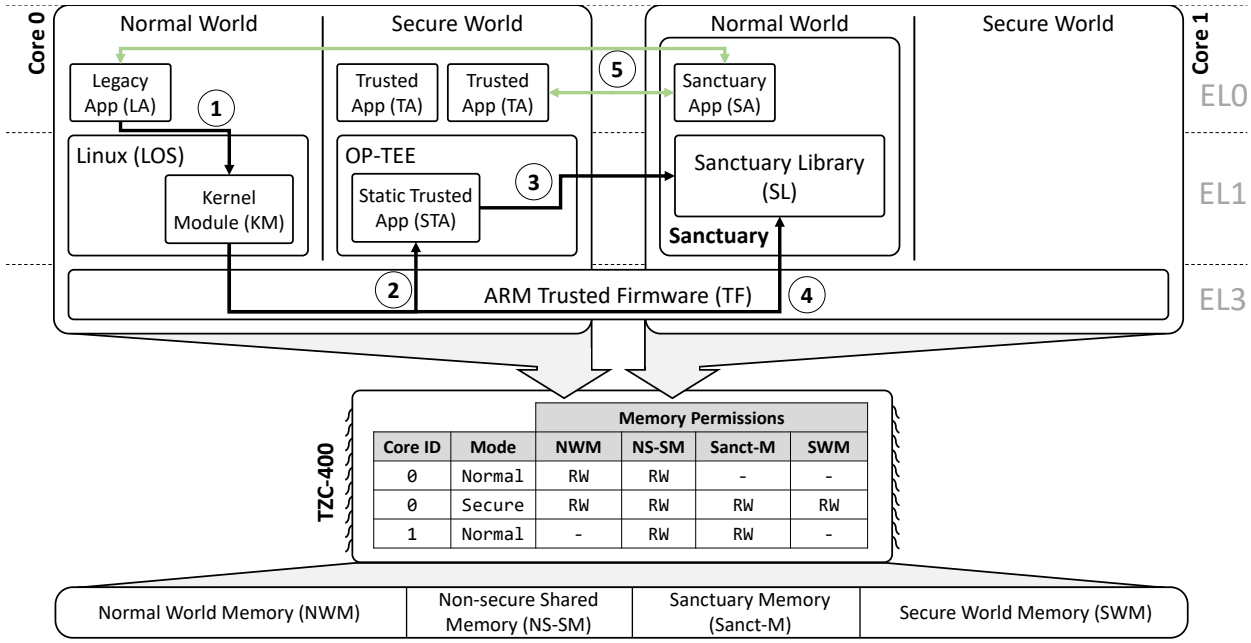

Figure 3: Implementation overview of SANCTUARY.

Since no source code of third-party TAs used on current devices is publicly available, we refer to Huang et al. [29] for average TA sizes. They implemented a mobile payment and chat TA consisting of 900 LOC and 200 LOC, respectively, which can be seen as a lower limit for implementing a useful TA. This shows that removing all third-party TAs from the secure world outweighs additions made to OP-TEE and the Trusted Firmware (TF) by an order of magnitude in terms of LOC. The number of added or modified LOC for all components are shown in Table I.

SANCTUARY Hardware Components. In SANCTUARY, we utilize the fact that unique master IDs can be assigned to every CPU core and therefore also to every memory transaction performed by a core. These transaction IDs can then be used to filter memory accesses on a hardware level. As such, memory regions can be made core-exclusive. The filtering and permission enforcement is performed by the TZC-400 memory controller. The TZC-400 allows or denies access to memory regions depending on two properties: (i) the type of the access transaction performed by the core running the code (*secure* or *non-secure*), and (ii) the bus master ID of the core which executes the SANCTUARY Instance. Enforced access permissions are shown in Figure 3 in the *Memory Permissions* table, and are described in detail in Section V-E.

SANCTUARY Usage. The high-level SANCTUARY life cycle works as follows: when a LA wants to execute sensitive code in form of an SA inside a SANCTUARY Instance, the LA requests execution of its bundled SA from the KM ①. The KM initiates the setup of the SANCTUARY Instance by loading the SANCTUARY binaries (SL and SA). Next, the KM removes one CPU core from the LOS and hands over control to the STA ② to perform all security-related steps, such as the verification of the SA ③. After successfully setting up of the SANCTUARY Instance, the KM triggers the SANCTUARY boot ④. When the boot process is finished the SA can execute the sensitive code, and communicate with its LA as well as with the TAs in the secure world ⑤.

Component	World	Added LOC	Modified LOC
Kernel Module	normal	713	-
Zircon Micro Kernel	normal	166	45
OP-TEE	secure	56	2
Static Trusted App	secure	472	-
ARM Trusted Firmware	secure	92	-
Proxy TA	secure	287	-
Sealing TA	secure	406	-

Table I: Modifications for Sanctuary Components.

In the following, we explain each component of SANCTUARY and its life cycle in detail.

A. Legacy OS

With SANCTUARY, the resource management remains in the LOS. We implement the required functionalities in a custom loadable kernel module (KM). The KM manages all resources needed for a SANCTUARY Instance. It is able to remove a core from the LOS and also to hand the core back to the LOS after a SANCTUARY Instance finished execution. Since we use Linux as the LOS in our prototype, we utilize the Linux CPU hotplug mechanism [11] for that purpose. Furthermore, the KM dynamically allocates memory for SANCTUARY Instances and their associated communication channels from SA to LA and from SA to TAs. Before a SANCTUARY Instance can be

started, the KM has to load the SANCTUARY binaries (SL and SA) into RAM which will be exclusively assigned to the SANCTUARY core afterwards. The OP-TEE driver facilitates the communication between LOS and OP-TEE.

B. Security Services

We keep the traditional structure of the secure world: Trusted Apps run on SEL0 (secure-world user space), while OP-TEE runs on SEL1 (secure-world kernel space). The TAs offer relevant security services. In our proof-of-concept implementation, we implemented a *Proxy TA* and a *Sealing TA*. The *Proxy TA* is used to establish a secure communication channel from an SA to remote servers. All data sent through the *Proxy TA* is authenticated with the platform key and bound to the identity of sender SA, i.e., the Proxy TA provides **remote attestation**. The *Sealing TA* provides **sealing** functionality which allows to bind data to a specific SA and to store it permanently on the device. For each SA an individual key is used.
A Static Trusted App (STA) represents a kernel module in OP-TEE. In our prototype, the STA verifies the SL using a pre-configured signature, sets up the SANCTUARY Instances, and tears them down. Moreover, the STA provides functionalities to TAs which can be used to, e.g., find out which SA is currently running in a SANCTUARY Instance, or to compute a hash over an SA binary.
All aforementioned security services rely on the Trusted Firmware (TF) as a trust anchor which is responsible for context switches between normal and secure world and low-level platform services. In our prototype, the TF was extended to verify several security-relevant steps during the SANCTUARY life cycle which is explained in detail in Section V-E.

C. Sanctuary

We implemented isolated code execution in SANCTUARY by running SANCTUARY Instances in the normal world on dedicated CPU cores. This isolates SANCTUARY Instances from untrusted LOS and TAs running on the remaining cores. A SANCTUARY Instance consists of two parts: the SL and an SA. The SL provides basic process and memory management functionalities for running an SA. In our implementation, we chose the Zircon micro kernel [24] as SL due to its small size (approx. 1MB) and versatility. After Zircon boots, it prepares the environment for the SA by configuring the CPU core, setting up the memory mappings and a basic execution environment. Then, the SA is started as a normal-world user process by the Zircon micro kernel. During execution, an SA can communicate with its corresponding LA and also with TAs in the secure world to utilize their provided security services (e.g., sealing or remote attestation). To achieve this, we extend the Zircon micro kernel with new system calls.
As required by SANCTUARY, the STA prevents simultaneous execution of SAs in one SANCTUARY Instance because this could lead to sensitive information leakage between SAs.

D. Memory Isolation Unit

In addition to isolating SANCTUARY execution through dedicated CPU cores, we protect SANCTUARY memory against normal-world accesses from other cores by leveraging the ARM TrustZone Address Space Controller (TZASC). As described in Section II, its recent implementation, the ARM TZC-400, allows setting memory-access permissions based on bus master IDs. Traditionally on ARMv8 architectures, all cores already have uniquely-assigned multi-processor-IDs (MPID register [3]). For all transactions sent to the system bus, multi-processor IDs are then translated to bus master IDs by a dedicated labeling component. Currently, as on the HiKey960 development board, transactions from all cores are labeled with the same bus master ID. For Sanctuary, only the mapping policy needs to be changed such that the bus transactions of cores are labeled with unique bus master IDs. No hardware modifications have to be made to the processor-core. We implemented the modified labeling ID-mapping policy using the ARM Fast Models virtualization tools. From software, we can now configure the TZC-400 such that memory regions can be exclusively assigned to single cores by filtering the bus transactions for the buster master ID labels. Details on how the TZC-400 needs to be configured are given in Section V-E.The performance overhead for configuring the TZC-400 is negligible compared to the rest of the Sanctuary startup, it only consists of a few register writes. It is important to mention that the assignment of bus master ID labels to transactions is already performed on all transactions by default. We only enforce the labeling of unique IDs. This means on the hardware level, SANCTUARY produces zero performance overhead. Therefore, evaluation on a Hikey 960 board gives realistic performance measurements.

If not enough unused bus master IDs are available to distinguish all core transactions, only a subset of the cores can run Sanctuary instances. This does not limit the general applicability of Sanctuary as long as at least two free bus master IDs are present.
On systems with the TZC-400, no additional hardware components are needed to implement SANCTUARY. Some device vendors already license the TZASC IP from ARM since it provides an industry-ready solution (e.g. Samsung on the Exynos chips [13]). Unfortunately, public information regarding the deployment of the TZC-400 on current platforms is limited.

E. Execution Life Cycle

In our prototype, a typical SANCTUARY life cycle consists of four phases: *(a) Sanctuary Setup*, *(b) Sanctuary Boot*, *(c) SA Execution*, and *(d) Sanctuary Teardown*, which we will explain in the following. In our prototype, we assume that a signature of the SL binary is already stored in the secure world. However, integrity and authenticity of the SL can generally also be established through certificates. Remote attestation of the SA can be achieved by leveraging the Proxy TA. However, alternative schemes, like Intel EPID, could be implemented as well. The implementation details of such a scheme are out of scope for this paper, thus, we refer the reader to Intel's documentation for a possible outline [36].

Sanctuary Setup. The SANCTUARY setup phase is performed by the KM in the normal world and the STA in the secure world. The KM manages system resources, whereas the STA performs all security relevant steps. The setup of a SANCTUARY Instance is triggered by the LA that requests execution of its sensitive code in the corresponding SA. Subsequently, the SL and SA binaries are loaded from the file system and handed over to the KM using *procfs*. The SL binary can also be loaded only once during system boot and remain in memory until the system is shut down. We implemented the binary loading

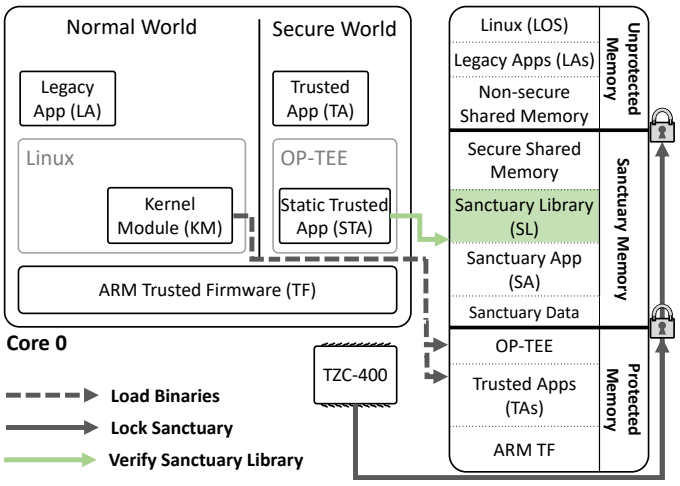

Figure 4: Memory layout after setting up a SANCTUARY. As before, **Core 0** is responsible for initializing and managing the Sanctuary App, which runs on **Core 1** (omitted for clarity).

in the normal world since OP-TEE cannot directly access the file system. The KM reserves additional memory for the SANCTUARY Instance which is used for memory allocations during SA run time. We call this memory area *Sanctuary Data*. Additional memory is reserved for the SANCTUARY Instance's communication channels. The communication between the LA and its SA is performed over non-secure (i.e. accessible by untrusted software) shared memory. In contrast, secure shared memory is used for communication between SA and TAs. The final memory layout after the setup is depicted in Figure 4.

After loading the binaries, the KM selects a CPU core to run the SANCTUARY Instance. The KM always selects the CPU core with the least load. Next, the Linux hotplug mechanism is used to shut down the selected core. If successful, the KM calls the STA and provides the ID of the selected core as an argument. This call traps into monitor mode where the TF checks that the selected core is indeed shut down before performing a world switch and handing over control to the STA. The STA then locks the SANCTUARY memory by configuring the TZC-400.

We assume that unique IDs `0-7` are assigned in hardware to 8 CPU cores, and that the selected SANCTUARY core has the ID `7`. Moreover, for the sake of simplicity, we assume that no other bus master than the CPU needs access to memory. Then, one of the up to 9 memory regions the TZC-400 can separate is configured to exactly cover the contiguous memory area in which the SL and SA binaries, the Sanctuary Data and the secure shared memory resides. We assume that region `1` is used for that purpose. The lowest address covered by region `1` is set using the *REGION_BASE_LOW_1* and *REGION_BASE_HIGH_1* registers. The highest address covered is set using the *REGION_TOP_LOW_1* and *REGION_TOP_HIGH_1* registers. Subsequently, the configured memory region `1` is solely assigned to the SANCTUARY core using the *REGION_ID_ACCESS_1* register. The bit assignments of the region ID access register is shown in Figure 5. The upper 16 bits of the register define the non-secure write access permissions (nsaid_wr_en), the lower 16 bits the non-

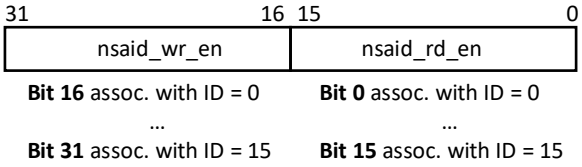

Figure 5: Region ID Access Register.

secure read access permissions (nsaid_rd_en). Every bit is associated with one bus master ID. This means, if e.g. bit 0 and bit 16 of *REGION_ID_ACCESS_1* are set to `1` and all other bits to `0`, only the bus master with the associated ID `0` is allowed to perform write or read access on region `1`. In our scenario, the SANCTUARY memory is assigned to the SANCTUARY core by setting *REGION_ID_ACCESS_1* to the value `0x800080`. Then, non-secure access is only allowed for the core with ID `7`, which is the SANCTUARY core in our example. For the memory regions that cover all of the normal-world memory except the non-secure shared memory, the bits are set to `0x7F007F` in the corresponding *REGION_ID_ACCESS* registers. Thus, permission to access the normal-world memory is granted to all cores except the SANCTUARY core. This is crucial for implementing two-way isolation. The region covering the non-secure shared memory is configured with the value `0xFF00FF` since the cores running the normal world and also the SANCTUARY core need access to it. As a last step, the regions covering the secure-world memory are configured with the bit value `0x0` such that no core can perform a non-secure access on the memory.

The resulting memory permissions are listed in Figure 3 for the different memory regions, core IDs, and execution modes. In the following verification step, the STA verifies the SL binary using the stored digital signature. For this purpose, the STA uses the RSASSA-PKCS1-v1_5 scheme together with SHA-256 which are provided by OP-TEE. After successful verification, the ARM TF is informed that the SANCTUARY is locked, verified, and ready to be booted.

Sanctuary Boot. After successful SANCTUARY setup, the KM calls the TF to boot the SANCTUARY core. Before starting the core, the TF checks that the SANCTUARY Instance was correctly locked and verified. After receiving the boot signal, the SANCTUARY core first executes the TF in EL3. During initialization of the TF, exception handlers needed for calling the TF from the SL are set up. The TF needs to be callable from the SL to shut the SANCTUARY core down in the teardown phase. After TF initialization, the core switches to EL1 and jumps to the entry point of the SL. We slightly modified the Zircon boot sequence to prevent information leakage from the SANCTUARY Instance We slightly modified the Zircon boot sequence to prevent information leakage from the SANCTUARY Instance by excluding all SANCTUARY memory from being cached in the shared L2 cache. Moreover, the external interrupts are configured using the core's CPU interface of the General Interrupt Controller (GIC) which cannot be accessed by other cores. This blocks external interrupts triggered by other cores, while allowing to receive interrupts requested by the SANCTUARY core, e.g. timer interrupts.

SA execution. While executing sensitive code, the SA may establish communication channels. The SA is able to commu-

nicate with its corresponding LA over the non-secure shared memory and with vendor TAs over the secure shared memory through OP-TEE. All data sent over the non-secure shared memory is accessible to the normal world, and hence, it is not part of the SANCTUARY memory partition. Communication is facilitated by the KM of the LOS in the normal world and by custom Zircon system calls on the SANCTUARY side. When the SA requires security services from vendor TAs, it communicates with the TA over the secure shared memory channel. On the secure-world side, this communication is facilitated by the STA. Since all data shared between the SA and a TA is sensitive, this data is solely exchanged over secure shared memory which is part of the protected SANCTUARY memory region assigned to the SANCTUARY core.

SANCTUARY allows two different implementation variants for SA to TA communication: (i) the OP-TEE driver is included in Zircon and the world switch to the secure world is performed by the SANCTUARY core itself, and (ii) the connection to the secure world is triggered by the SA's corresponding LA. This means an SA first has to communicate with the normal world before it can communicate with a TA. However, the data exchanged between TA and SA remains unaccessible to the LA. We implement the second variant in our prototype since it requires less modifications to the Zircon kernel.

Sanctuary Teardown. The three-step teardown of the SANCTUARY is triggered by the LA. The first step is to shut down the SANCTUARY core when the LA signals to the SA that its services are not needed anymore. Subsequently, the SA saves its state (if needed) using e.g. the sealing services. Next, internal clean up actions bring the Zircon kernel back in its original state and invalidate the L1 cache to prevent data leakage. Then, the SL signals the STA that it successfully performed the clean up action. Subsequently, the TF is used to shut down the core. The second step is to unlock the SANCTUARY memory. Analogously to the locking in the setup phase this is performed by the STA. Again, the modified TF checks that the SANCTUARY core is indeed shut down before performing the world switch and handing control over to the STA. The STA checks if the SA was able to perform its clean up action. Then, the secure shared memory and Sanctuary Data memory are zeroed to prevent leakage of SA data. Finally, the configuration of the TZC-400 is reverted such that the SANCTUARY memory region and the SANCTUARY core are freed. In the third step of the teardown process, the KM uses the Linux hotplug mechanism to reclaim the available core.

VI. SECURITY ANALYSIS

The goal of SANCTUARY is to protect against a strong attacker, as described by our adversary model (see Section III-A). We also derived the requirements for our design of SANCTUARY in Section III-B. For a systematic analysis of SANCTUARY, we will now look at all possible attack vectors that are available to an adversary in our threat model. In particular, we can see from Figure 2 that attacks can originate from three different locations on the platform: (i) the normal-world user space, (ii) the normal-world OS, and (iii) a malicious SA. In all three cases, the goal of the attacker is to compromise the integrity or data confidentiality of a victim SA or gain control over the LOS. This can happen at any point in time during the life-cycle of an SA (i.e., setup, boot, execution, or tear-down), and hence, we will discuss each case in the following. In particular, malicious code in the normal world can aim at either manipulating the SL and SA binaries before they are loaded (Section VI-A), overcome the isolation of SANCTUARY (Section VI-B), manipulate persistently stored data of an SA (Section VI-C), or extract information from an SA via the cache (Section VI-D). We discuss the case of malicious SAs in Section VI-E. As we will show, an adversary cannot compromise the security of SANCTUARY in any of those cases, and does not gain any advantage from executing code inside an SA over regular normal-world execution.

A. Binary Integrity

SL's and SAs's binaries are saved unencrypted in normal-world memory. Nevertheless, SANCTUARY ensures integrity of these binaries by using local attestation and by providing functionalities for remote attestation, respectively. SANCTUARY stores a signature of the SL in the secure-world memory. Before a SANCTUARY Instance is started, the STA performs a local attestation by measuring the SL binary and by verifying it against the stored signature. If verification fails, SANCTUARY's setup is aborted and the modified code therefore never executed. Developers can verify an SA's integrity using remote attestation. Whenever an SA connects to a server, TEE functionalities are used to establish a secure connection to the server. Moreover, the STA creates a signature of the SA which is also send to the server. Thus, the server can check if the SA is in a valid state before provisioning sensitive data to it.
These properties, together with the properties in Section VI-B, fulfill security requirement 1: *Code and data integrity*.

B. Code and Data Isolation

The SANCTUARY design provides strong hardware-enforced isolation of code and data. The SANCTUARY memory isolation is enforced by TrustZone before the integrity of the SL is verified. Once the SANCTUARY memory is locked, no core except the selected SANCTUARY core can perform non-secure reads or writes on the SANCTUARY memory region. The selected SANCTUARY core always boots in the TF and then jumps to an address in the SL which is set as a constant in the TF. During the boot process of the SANCTUARY Instance, the SL ensures that all interrupts from the system-wide interrupt controller, triggered from other cores than the SANCTUARY core, are disabled. Only a core itself can configure its interface to the GIC. Therefore, the execution of a SANCTUARY Instance cannot be interrupted by another core. Moreover, only the SANCTUARY core can shut itself down. During runtime, the SANCTUARY design makes sure that sensitive data is only passed to and received from a locked SANCTUARY Instance. When performing a world switch to the secure world, the TF verifies that the call was issued from the SANCTUARY core. Access from all other cores to the trusted functionalities in the TEE will be blocked. If the call was issued from the SANCTUARY core, the vendor TAs in the TEE use the STA to check if the SANCTUARY Instance is in correct state before reading or writing any data to the memory shared between secure world and SA. SANCTUARY also prevents the injection of data into the free SANCTUARY memory space before a SANCTUARY Instance is locked and the extraction of data after a SANCTUARY Instance is unlocked. The secure-world

STA overwrites SANCTUARY memory not reserved for either SL or SA with a fixed value after a SANCTUARY Instance is locked and before it is unlocked, including the secure shared memory. Besides, the SL is reset to its original state during shutdown, hence, it will not contain last executed SA's data. These properties fulfill security requirement 1: *Code and data integrity* in combination with the properties in Section VI-A. Additionally, in combination with the properties in Section VI-C, security requirement 2: *Data confidentiality* is fulfilled. Moreover, SANCTUARY's temporal and spatial hardware-enforced isolation fulfills security requirement 5: *Hardware-enforced resource partitioning* (in combination with the properties in Section VI-D). Finally, the exclusive shared memory between a SANCTUARY Instance and the secure world fulfills security requirement 3: *Secure channel to secure world*.

C. Secure Storage

SANCTUARY allows the secure and persistent storage of SA data using the STA and security services from the secure world. SANCTUARY ensures that the data is sealed to a SA entity using keys that are derived from a hash value computed over the SA binary. As a result, only an unmodified SA can successfully decrypt its own data. For the persistent storage of the sealed data, a SANCTUARY Instance uses the functionalities provided by the TEE. Depending on the TEE implementation, this might also allow the SA to bind its data to the device or to save it in roll-back protected memory. These properties fulfill security requirement 2: *Data confidentiality* (in combination with the properties from Section VI-B).

D. Cache Attack Resilience

As shown by recent Spectre [33] attacks, cache-based attacks can be very powerful. An attacker could, for instance, try to mount a software side-channel attack to extract data from cache lines used by a SANCTUARY Instance. Thus, these attacks are considered in SANCTUARY's design and implementation. As usual on ARMv8 platforms, we assume presence of first-level cache (L1) and second level cache (L2). On these platforms, first-level caches (L1) are core-exclusive, while the L2 cache is shared. This configuration allows two attack scenarios: direct attacks, and side-channel attacks.

Direct Attacks. A privileged attacker in the normal world could map the SANCTUARY memory region into an attacker-controlled memory space. This could potentially give an attacker *direct* access to the cached data of a SANCTUARY Instance, even without the permission to read the main memory for this physical address. For the L1 cache, we prevent this by running a SANCTUARY Instance on its own core and by invalidating the L1 cache before a SANCTUARY Instance is shutdown and unlocked. For the L2 cache, there are two ways to prevent direct attacks. One way is to configure the SANCTUARY memory region as `outer non-cacheable`, whereas the outer domain is represented by all caches outside of a particular CPU core. As a result, the SANCTUARY memory is never cached in the shared L2 cache. In Section VII, we show that this still gives practical performance. Alternatively, changes to the caches could be made on the hardware level to extend the enforcement of identity-based filtering to the L2. This prevents an attacker from directly accessing cache lines uses by a SANCTUARY Instance. In both cases, an attacker could also not inject own malicious data into the L2 cache. On ARMv8, data caches are normally either *Physically Indexed, Physically Tagged* (PIPT) or *Virtually Indexed, Physically Tagged* (VIPT) [3]. This means cache lines are tagged using physical addresses in both configurations. Since the attacker cannot write to or read from the physical addresses of the SANCTUARY memory, the attacker can also not fill the cache for those addresses.

Side-Channel Attacks. An unprivileged attacker could mount side-channel attacks like Prime+Probe [43] or Flush+Reload [53] to leak data from L1 or L2 caches. For the L1 cache, this is prevented by running a SANCTUARY Instance on its own core, i.e. the attacker cannot measure accesses to the SANCTUARY core's L1 cache while it is running. To prevent measurements after shutdown, a SANCTUARY Instance invalidates its L1 cache before it is shut down and unlocked. For the L2 cache, implementing the identity-based filtering does not solve the cache-side channel issue. Thus, cache partitioning (or a similar approach) is needed to prevent leakage. We prevent side-channel attacks on the L2 cache by excluding SANCTUARY memory from L2, which yields practical performance (cf. Section VII).

E. Malicious Sanctuary App

One strength of SANCTUARY is that third-party developers can easily create and deploy own SAs. This, however, also allows attackers to create malicious SAs. If an user is tricked into installing such an SA, it will be executed as a valid SA in a SANCTUARY Instance. The attacker could then try to attack the normal world or secure world from such a malicious SA, hence, SANCTUARY must protect against malicious SAs. With a malicious SA, an attacker might attack the LOS and LAs running in the normal world. Yet, an SA only has user privileges (EL0), EL1 is controlled by the device vendor providing the SL. If the attacker is able to successfully perform a privilege-escalation attack and compromise the SL, the secure-world memory is still not accessible for the attacker since the SA runs in normal world. In particular, since only the SANCTUARY memory is assigned to the SANCTUARY core, remaining normal-world memory could still not be accessed. Only the non-secure shared memory to the LA (developed by the attacker anyway) would be affected. An attacker could try to use a malicious SA to leak data from either other SAs or from TAs. However, since the SANCTUARY design dedicates CPU cores to SAs one at a time, unintended information flow between SAs is prevented. These properties fulfill security requirement 4: *Protection from malicious SAs*.

VII. EVALUATION

We evaluate SANCTUARY by implementing a real-world use-case in our prototype and by thoroughly measuring the performance of all SANCTUARY components. As mentioned in Section V, the overall implementation minimizes TCB changes by adding less than 1400 LOC. Thus, SANCTUARY fulfills functional requirement 6: *Minimal software changes* from Section III-B. The evaluation was performed on the HiKey 960 development board. The HiKey 960 provides an ARMv8 SoC design with an ARM big.LITTLE processor architecture equipped with four ARM Cortex-A73 and four Cortex-A53 cores. Every Cortex-A73 core has 64KB L1 instruction caches

Measurement	with L2 (us)	without L2 (us)
LA to STA	98	[88]
LA to TA	123	[120]
LA to SA	150	249
SA to TA	310	353

Table II: Performance Sanctuary Communication, square brackets indicate that deactivating L2 for the SANCTUARY Instance had no effect.

Measurement	with L2 (ms)	without L2 (ms)
Load Sanctuary binaries	7	[7]
Shut down core	113	[109]
Lock & Verify	13	[12]
Start Sanctuary:	59	311
Early core initialization	37	36
Set up kernel space env.	18	130
Set up user space env.	4	145

Table III: Performance Sanctuary Setup.

and 64KB L1 data caches. Moreover, all Cortex-A73 cores share a unified L2 with a size of 2MB. The energy-efficient Cortex-A53 cores share a unified L2 cache of 512KB. Besides, every Cortex-A53 core has exclusively access to 32KB L1I and 32KB L1D caches.

A. Microbenchmarks

We evaluated the performance of SANCTUARY by measuring the run time of the individual components and operations of our prototype. We performed the evaluation for the SANCTUARY configurations with both, active and deactivated L2 cache for the SANCTUARY core (other cores are unaffected by this). For an active L2 cache, we consider a weaker attacker model, that is similar to the one of Intel SGX, i.e., side-channel attacks are out of scope; orthogonal approaches like cache partitioning are needed. Furthermore, we assume that the identity-based filtering is also implemented in shared L2.
When not caching the SANCTUARY memory in L2, we can consider a stronger adversary that leverages software side-channel attacks. Square brackets in the results for the configuration without L2 highlight that these measurements are not influenced by the SANCTUARY L2 cache configuration. The shown deviations can be attributed to the complexity of modern processors which causes timing differences between consecutive runs. We used the generic timer available on ARM-based architectures to perform all our measurements. Moreover, we computed the relative standard deviation of our measurements to assess SANCTUARY's stability. The presented results are averaged over 100 runs per configuration. Based on these numbers, we conclude that latency introduced by SANCTUARY is practical in real-world applications.

1) Sanctuary Communication: Table II contains measurements for the different communication channels that exist in the SANCTUARY design. The first two measurements, *LA to STA* and *LA to TA* show how long it takes to perform a call from an LA to a TA or a STA. These measurements are completely independent from a SANCTUARY Instance but can be used to assess the performance of SANCTUARY's communication channels. The time required to perform a call from an LA to its SA with L2 cache is comparable to regular TrustZone communication. Hence, SANCTUARY does not introduce a large communication latency. The higher overhead for the communication between SA and TA is caused by the fact that the context switch is not performed by the SANCTUARY core but is triggered by the corresponding LA. This means the SA first has to communicate with the normal world before it can communicate with the secure world. As mentioned in Section V, the OP-TEE driver could also be included into the SL. Then, the SANCTUARY core could switch directly to the secure world. In this case performance similar to that of a call from LA to TA can be expected. When the L2 cache is deactivated for SANCTUARY memory, the duration of a call from LA to SA increases by a factor of 1.66, however the overall performance is still practical. The relative standard deviation of the communication measurements is low with 28%-34% for the configuration with L2 activated and 20%-32% for the configuration with L2 deactivated.

2) Sanctuary Setup: The primary difference in running SANCTUARY Instance compared to TAs lies in the setup time needed to isolate a CPU core. The bare execution speed of SAs and TAs is the same as they run on the same hardware. Table III breaks down the single steps performed starting the LA, requesting a SANCTUARY Instance initialization, up to execution of the SA. In the *Load Sanctuary binaries* step, both the SL and SA binaries are loaded in 7ms. In the next step, the Linux hotplug mechanism is used to shut down the core. With L2 cache enabled for this core, this represents the most expensive step of the SANCTUARY setup process with 113ms. Next, the SANCTUARY is locked and verified (cf. Section V). Subsequently, the Zircon kernel is booted (*Start Sanctuary* step). We measured the boot process in three phases. The first phase covers early initialization of the core. In the second phase, the platform components are initialized and the kernel environment is set up. In the last phase, the user space environment is set up, it ends with the execution of the SA. The results show that the boot overhead is higher if the L2 cache is not active. In the second boot step, the boot time increases by a factor of 7, in the third step even by a factor of 36. However, even without using the L2 cache for the SANCTUARY core, the complete SANCTUARY setup can still be performed in around 450ms. If the identity-based filtering feature is implemented in the cache, a setup time around 200ms can be achieved. Further optimizations could be achieved by reducing the SL. The relative standard deviation of the measurements with L2 activated range from 27% to 38%. With deactivated L2 the relative standard deviation values range from 26% to 44%.

3) Sanctuary Teardown: Table IV shows the performance evaluation of the Sanctuary teardown. In the *Sanctuary shutdown* step, the L1 cache is invalidated and the Zircon kernel brought into its original state. In the *Unlock Sanctuary* step, the SANCTUARY memory is zeroed which takes up most of the time. The complete teardown of the SANCTUARY can be

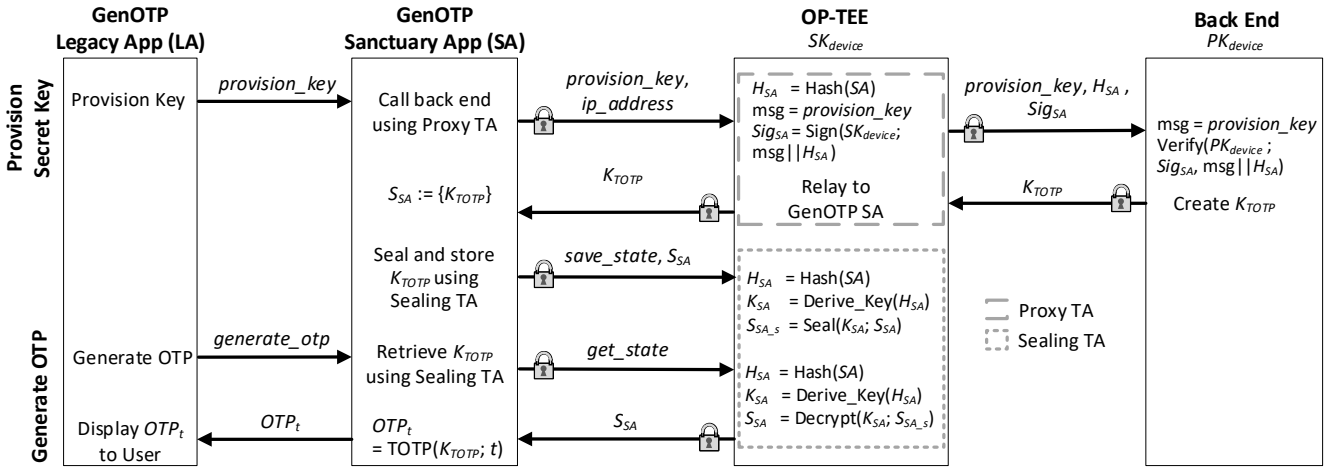

Figure 6: GenOTP app protocols for secret key provisioning and OTP generation. A lock symbol indicates TrustZone-protected communication channels.

Measurement	with L2 (ms)	without L2 (ms)
Sanctuary shutdown	1	[1]
Unlock Sanctuary	45	58
Restart core	53	[54]

Table IV: Performance Sanctuary Teardown.

performed very fast in around 100ms with and without L2 cache. For the former case, the relative standard deviation ranges from 15% to 40% and for the later from 14% to 25%. The measurements further emphasize the practicability of SANCTUARY, as setup and teardown induce a total run time overhead of approximately 340ms with L2, respectively approximately 600ms without L2.

B. Use-Case: OTP Generation for Two-Factor Authentication

To illustrate the practicability of SANCTUARY in real-world applications, we implemented a One-time Password (OTP) generator app on top of our prototype, which we call *GenOTP*. The *GenOTP* app, which we will now describe in more detail, consists of an LA and an SA. It can be used to seal a secret key to the SA and restore it at a later point in time to generate a fresh OTP. With SANCTUARY, every service provider can develop a custom app that protects the secret key without the need of an own TA in the TEE.

1) Scenario Description: Two-factor authentication schemes are often used for authenticating users on websites. The first factor, the knowledge factor, is usually represented by an username and a password. The second factor, the possession factor, is represented by a hardware token or a mobile device that creates fresh OTPs. The OTPs are created from a secret key shared between the user's device and the verification server. In a Time-based One-time Password Algorithm (TOTP) [30], the secret key is then used together with a fresh timestamp to generate an OTP. The secret key must be securely stored on the device and the TOTP code protected during execution.

In our scenario, an online retailer wants to offer two-factor authentication for its online shop. We assume that a customer's mobile device contains a TEE and supports SANCTUARY. The online retailer implemented the *GenOTP* app consisting of non-sensitive code in an LA and security-sensitive code in an SA. We further assume that the *GenOTP* app is already installed on the user's device and that the TEE contains an unique asymmetric key pair $(SK_{device}, PK_{device})$, brought onto the device during production. During the installation of *GenOTP*, PK_{device} was sent to the retailer's back end. We assume that *Proxy TA* and *Sealing TA* are present on the device.

2) Provision Secret Key: For generating OTPs on the mobile device, a secret key K_{TOTP} needs to be provisioned to it. The process for receiving the key from the retailer's back end is shown in Figure 6. The customer selects the option to provision a key in the *GenOTP* LA. A SANCTUARY Instance is started and executes the *GenOTP* SA. The SA then hands over the IP address of the server it wants to communicate with and the message it wants to send to the *Proxy TA*. In this case, the message only contains the information $provision\_key$. The *Proxy TA* now calls the STA to get the hash value of the SA binary running in the SANCTUARY Instance and creates a signature Sig_{SA} over the hash value and message using the device unique private key SK_{device}.
After the *Proxy TA* created the signature Sig_{SA}, it sends it to the retailer's back end, together with the hash H_{SA} calculated over the SA binary and the message $provision\_key$. In particular, the signed and thus protected message and H_{SA} is passed to the network stack in the normal world to be forwarded to the server. The involvement of the normal world is omitted in Figure 6 for lucidity as it is only providing non-secure functionalities. The retailer's back end, which has PK_{device}, can now verify if the SA was correctly loaded in a SANCTUARY Instance, since only then a valid signature is created by the *Proxy TA*. If verification succeeds, a secret key K_{TOTP} is created and returned via the *Proxy TA* to the SA.
The SA now needs to store the received key s.t. fresh OTPs can be generated anytime, even without Internet connection. For this, the *Sealing TA* is used. The SA collects all data it wants to seal in a state object S_{SA} and forwards it to the *Sealing TA*.

In our scenario, S_{SA} only contains the secret key K_{TOTP}. In general, any data that needs to be stored persistently can be incorporated into S_{SA}. When the *Sealing TA* receives S_{SA}, it calls the STA to get the SA binary hash. From the hash a symmetric key K_{SA} unique to the SA is derived. K_{SA} is then used to seal S_{SA} to the specific SA, producing the cipher S_{SA_S}. Finally, the data is sealed to the *Sealing TA* using functionality provided by OP-TEE.

3) Generate OTP: When the customer later wants to generate a fresh OTP for login into the retailer's online shop, he selects the OTP generation option from the LA. After a SANCTUARY Instance is started, the SA uses get_state of the *Sealing TA* to retrieve its saved data. The *Sealing TA* first restores the sealed SA state S_{SA_S} using the functionality provided by OP-TEE. Next, a hash computed over the SA binary is received from the STA and used to derive the SA unique key K_{SA}. The key is then used to decrypt S_{SA_S} which results in the state object S_{SA}. S_{SA}, which contains the secret key K_{TOTP}, is then returned to the *GenOTP* SA. Finally, the SA runs the TOTP algorithm to compute a fresh OTP_t from the key K_{TOTP} and the current timestamp t. The generated OTP is returned to the LA which displays it to the user. The customer can now use the OTP to perform a two-factor authentication.

4) GenOTP Performance: Besides performing microbenchmarks We also measured performance of the implemented *GenOTP* app. Averaging over 100 runs, we measured the time it takes to perform the *Provison key* and the *Generate OTP* processes shown in Figure 6. The results are listed in Table V. Provisioning a key onto the device takes around 1s, whereas the provisioning time increases by factor 1.3 without L2 cache. Measurements include all steps from SANCTUARY Instance setup, SA signature computation, encrypting and storing the secret key, up to the point where the SANCTUARY Instance is completely teared down. Only the communication and processing delays introduced by the back end are not included. We again split the measurement into multiple phases: (1) SANCTUARY Instance is started and the call to the back end is issued, (2) the secret key is received and processed by the SA, and (3) the secret key is stored, the SANCTUARY Instance teared down and all resources reclaimed by the normal world. The measurement of *generate OTP* is divided into two phases: (1) setup and retrieval of the secret key from the *Sealing TA*, and (2) generation of a fresh OTP and SANCTUARY Instance teardown. Generating a fresh OTP using SANCTUARY takes around half a second. When the L2 cache is deactivated, the process time increases by a factor of 1.6. The relative standard deviation values for the *GenOTP* measurements range from 11% to 21% for the configuration with L2 activated and from 11% to 22% for the configuration with L2 deactivated.
The results show that the SANCTUARY design is indeed practical in real-world scenarios, even without the L2 cache. The setup of the SANCTUARY Instance and the communication with other normal-world and secure-world components is fast enough such that the user experience is not influenced. Moreover, since the SANCTUARY Instance runs on an isolated core, the LOS does not have to be suspended and can run in parallel with the SANCTUARY Instance. This means the delays introduced by the SANCTUARY setup and teardown never result in a frozen UI since the LOS is always fully responsive. Therefore, SANCTUARY fulfills functional requirement 7: *Positive user experience* from Section III-B.

Measurement	**with L2 (ms)**	**without L2 (ms)**
Provision key:	884	1174
Setup & Server call	780	1067
Process server result	10	10
Save state & Teardown	94	97
Generate OTP:	365	630
Setup & Retrieve state	266	514
Generate OTP & Teardown	99	116

Table V: GenOTP App Performance.

VIII. RELATED WORK

In this section we compare SANCTUARY against existing TEE implementations in hardware and software.

A. Secure Hardware Architectures

Hardware-based security architectures have been developed by both, academia and industry. Industry solutions like Intel Software Guard Extensions (SGX) [31] and ARM TrustZone [5] are available in commercial off-the-shelf products. Intel SGX [31] provides hardware-enforced code and data isolation, while the TCB consists of the CPU and its microcode only. So-called enclaves run security-sensitive code that can be can be verified via local and remote attestation. However, SGX is tailored to Intel x86 desktop/server chips, and thus not found in embedded (or mobile) devices. For mobile devices, ARM offers a TEE implementation with TrustZone [5]. TrustZone isolates critical code by dividing physical hardware in virtual normal-world and secure-world realms. The secure world runs its own TOS and TAs, but vendors are very strict about which applications may run in the secure world. SANCTUARY overcomes this restriction by only having a minimal and fixed set of functionality in the secure world, while the remaining sensitive code runs in isolated normal-world enclaves.

Sanctum [14] provides protected enclave execution similar to Intel's SGX. Unlike SGX, it extends the open-source RISC-V platform, and provides additional protection mechanisms against side-channel attacks by applying cache partitioning to the last level cache (LLC), while flushing the per-core L1 cache upon enclave exit. SPM [50] and follow-up works like Sancus [41], [42] propose an isolation architecture for low-end embedded systems with a hardware-only TCB. They extend the `openMSP430` CPU architecture with additional CPU instructions for secure provisioning and protected storage, as well as an extended memory access logic with isolation enforcement. TrustLite [34] uses the generalized concept of an Execution-Aware Memory Protection Unit (EAMPU) to enforce program counter based memory access policies stored in tables directly in the SoC and a trusted loader to enable isolated trusted applications on a low-end embedded processor architectures. All these approaches are based on CPU architectures not commonly available in end-user devices, while SANCTUARY is based on the widely used ARM architecture.

B. Secure Software Architectures

Komodo aims to strengthen software isolation between the TrustZone applications in the secure world by using a hardened, formally verified microkernel as the secure-world OS [18]. Komodo replaces deployed microkernels by solutions like MobiCore [7] and hence does not support legacy systems.

Hypervisor-based approaches like vTZ [28], AppSec [46], Terra [21], InkTag [26], TrustVisor [37] or MiniBox [35] provide isolation using virtualization. This has four main disadvantages: (i) their TCB contains a relatively large hypervisor, (ii) they block usage of virtualization for non-security purposes, (iii) they require additional hardware to protect against Direct Memory Access (DMA) attacks, and (iv) they negatively influence the performance of the OS. SANCTUARY does not rely on virtualization and can even be used in combination with a hypervisor. Cho et al. [13] try to mitigate the influence on the OS by activating the hypervisor on-demand. Therefore, the OS is only influenced when sensitive code is executed. In SANCTUARY, the performance of the OS is not influenced when sensitive code is executed in parallel since no hypervisor is running underneath the normal-world OS.

Other approaches try to minimize the normal-world TCB by protecting the non-secure kernel. TZ-RKP [10] and SPROBES [22] both protect the LOS kernel by instrumenting critical functionality to trap into the secure world, where the call is filtered. As demonstrated by the Towelroot exploit [27], such mechanisms can be circumvented. KENALI [48] instead uses data-flow integrity to enforce policies of the LOS kernel's access control system, while SKEE [9] aims to detect attacks against the kernel by providing an isolated execution environment at the kernel's privilege level running a kernel monitor. SANCTUARY does not require the kernel to be trusted to guarantee isolated execution, moreover, SANCTUARY also protects the LOS kernel from potentially malicious SAs.

Flicker [38] and TrustICE [51] provide *temporal isolation* only, i.e., they cannot provide isolation for systems where TEEs execute in parallel with untrusted software. Hence, on todays commonly used multi-core systems the applicability of these approaches is very limited. With temporal isolation, the entire system has to be suspended, i.e., hibernation of the LOS and all applications. Afterwards, the TEE can execute exclusively on the system and only after the TEE has terminated, the normal system can be restored and continue execution. Flicker uses Intel's Trusted Execution Technology (TXT) to reset the system at runtime to a trusted execution state. TrustICE is conceptually similar to Flicker: it uses the secure world, rather than TXT, to reset the normal world to a trusted state. In TrustICE, TA binaries are stored in TrustZone memory. When a TEE is started, the LOS is suspended and the binaries are copied to normal-world memory for execution. After the TEE finished execution, the LOS has to be restored by the secure world. During execution, TrustICE provides only one-way isolation and executes in kernel-mode, this means that malicious TAs can manipulate normal-world software, e.g., compromise the LOS. SANCTUARY, in contrast, does provide *spacial isolation*, which enables the *parallel* execution of untrusted code with one or multiple TEE instances. Furthermore, SANCTUARY offers hardware-enforced two-way isolation and restricts SAs to user-mode execution. Hence, SANCTUARY protects systems from malicious SAs, which is highly relevant for practical deployment.

IX. Conclusion

We presented SANCTUARY, our novel security architecture for extending the TrustZone software ecosystem with user-space enclaves. SANCTUARY provides hardware-enforced two-way isolation obviating the need to trust or vet the code of SAs, as malicious SAs cannot have more power than normal user-space applications.

SANCTUARY is based on the bus master identity filtering introduced with ARM's latest memory controller design and allows the parallel isolation of individual CPU cores for executing security-sensitive code, i.e., SANCTUARY does not affect the user experience negatively. Furthermore, our performance evaluations for our proof-of-concept implementation shows low latencies for typical use cases, all of which makes SANCTUARY highly practical.

Acknowledgments

This work was co-funded by the DFG (projects P3 and S2 within CRC 1119 CROSSING, and HWSec), by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

References

[1] "OP-TEE," https://www.op-tee.org/.

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[3] ARM Limited, "ARM Cortex-A Series Programmer's Guide for ARMv8-A," http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf.

[4] ——, "GlobalPlatform TEE & ARM TrustZone technology: Building security into your platform," https://pdfs.semanticscholar.org/presentation/7b94/63d58a2d4ec9724c5933419be6f08754ce86.pdf.

[5] ——, "Security technology: building a secure system using TrustZone technology," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2008.

[6] ——, "CoreLink TrustZone Address Space Controller TZC-380," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0431c/DDI0431C_tzasc_tzc380_r0p1_trm.pdf, 2010.

[7] ——, "Giesecke & Devrient and ARM Protect Mobile Applications From Data Theft," https://www.arm.com/about/newsroom/26718.php, 2010.

[8] ——, "ARM CoreLink TZC-400 TrustZone Address Space Controller," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0504c/DDI0504C_tzc400_r0p1_trm.pdf, 2013.

[9] A. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm," in *23rd Annual Network and Distributed System Security Symposium*, ser. NDSS, 2016.

[10] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2014.

[11] S. S. Bhat, "Interaction of suspend code (s3) with the cpu hotplug infrastructure," https://www.kernel.org/doc/Documentation/power/suspend-and-cpuhotplug.txt, 2014.

[12] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *Annual Network and Distributed System Security Symposium*, ser. NDSS, 2016.

[13] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices," in *USENIX Annual Technical Conference (USENIX ATC)*, 2016.

[14] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation." in *USENIX Security Symposium*, 2016.

[15] Dan Rosenberg, "Reflections on trusting trustzone," https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf, 2014.

[16] L. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi, "Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM," in *8th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS, 2013.

[17] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security & Privacy*, 2014.

[18] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP, 2017.

[19] Gal Beniamini, "Qsee privilege escalation vulnerabilitiy," http://bits-please.blogspot.de/2015/08/full-trustzone-exploit-for-msm8974.html, 2015.

[20] ——, "Qsee privilege escalation vulnerabilitiy," http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, 2016.

[21] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP, 2003.

[22] X. Ge, H. Vijayakumar, and T. Jaeger, "SPROBES: Enforcing kernel code integrity on the trustzone architecture," in *Mobile Security Technologies*, ser. MoST, 2014.

[23] Global Platform, "Tee management framework (version 1.0)," https://www.globalplatform.org/specificationform.asp?fid=7866, 2016.

[24] Google, "Zircon micro kernel," https://fuchsia.googlesource.com/zircon.

[25] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[26] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2013.

[27] G. Hotz, "Towelroot android root exploit," https://towelroot.com/, 2014.

[28] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[29] W. Huang, V. Rudchenko, H. Shuang, Z. Huang, and D. Lie, "Pearl-TEE: Supporting Untrusted Applications in TrustZone," 2018.

[30] IETF, "Totp: Time-based one-time password algorithm - rfc6238," https://tools.ietf.org/html/rfc6238, 2011.

[31] Intel, "Intel Software Guard Extensions Programming Reference," https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[32] C. Josh Thomas, Nathan Keltner, "Reflections on trusting trustzone," https://pacsec.jp/psj14/PSJ2014_Josh_PacSec2014-v1.pdf, 2014.

[33] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*.

[34] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 10.

[35] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[36] J. M., "Intel Software Guard Extensions Remote Attestation End-to-End Example," https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example, 2018.

[37] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[38] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *ACM SIGOPS Operating Systems Review*, 2008.

[39] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[40] M. Meeker, "Internet trends 2015," *Glokalde*, vol. 1, no. 3, 2015.

[41] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *22nd USENIX Security symposium*, 2013.

[42] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for IoT Devices," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, p. 7, 2017.

[43] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *RSA Conference*, 2006.

[44] Project Zero, "Lifting the hyper visor," https://googleprojectzero.blogspot.de/2017/02/lifting-hyper-visor-bypassing-samsungs.html, 2017.

[45] ——, "Trust issues: Exploiting trustzone tees," https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017.

[46] J. Ren, Y. Qi, Y. Dai, X. Wang, and Y. Shi, "Appsec: A safe execution environment for security sensitive applications," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE, 2015.

[47] D. Shen, "Exploiting trustzone on android," https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf, 2015.

[48] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *23rd Annual Network and Distributed System Security Symposium*, ser. NDSS, 2016.

[49] N. Stephens, "Behind the pwn of a trustzone," https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose, 2016.

[50] R. Strackx, F. Piessens, and B. Preneel, "Efficient isolation of trusted subsystems in embedded systems," in *Security and Privacy in Communication Networks*, 2010.

[51] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[52] Tencent, "Defeating samsung knox with zero privilege," https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege-wp.pdf, 2017.

[53] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security Symposium*, 2014.