# Minimizing a Smartphone's TCB for Security-Critical Programs with Exclusively-Used, Physically-Isolated, Statically-Partitioned Hardware

Zhihao Yao[†*], Seyed Mohammadjavad Seyed Talebi[†*], Mingyi Chen[†], Ardalan Amiri Sani[†], Thomas Anderson[‡]
[†]UC Irvine, [‡]University of Washington
{z.yao, mjavad, mingyi.chen, ardalan}@uci.edu, tom@cs.washington.edu

## ABSTRACT

Smartphone owners often need to run security-critical programs on the same device as other untrusted and potentially malicious programs. This requires users to trust hardware and system software to correctly sandbox malicious programs, trust that is often misplaced. Our goal is to minimize the number and complexity of hardware and software components that a smartphone owner needs to trust. We present a split-trust hardware design composed of statically-partitioned, physically-isolated trust domains. We introduce a few simple, formally-verified hardware components to enable a program to gain provably exclusive and simultaneous access to both computation and I/O on a temporary basis. To manage this hardware, we present OctopOS, an OS composed of mutually distrustful subsystems. We present a prototype of this machine (hardware and OS) on a CPU-FPGA board and show that it incurs a small hardware cost compared to modern smartphone SoCs. For security-critical programs, we show that this machine significantly reduces the required trust compared to mainstream TEEs while achieving usable performance. For normal programs, performance is similar to a legacy machine.

## CCS CONCEPTS

• **Security and privacy → Systems security**; **Mobile platform security**; **Trusted computing**.

## KEYWORDS

Physical isolation, static partitioning, exclusive use

**ACM Reference Format:**
Zhihao Yao, Seyed Mohammadjavad Seyed Talebi, Mingyi Chen, Ardalan Amiri Sani, Thomas Anderson. 2023. Minimizing a Smartphone's TCB for Security-Critical Programs with Exclusively-Used, Physically-Isolated, Statically-Partitioned Hardware. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys '23), June 18–22, 2023, Helsinki, Finland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3581791.3596864

## 1 INTRODUCTION

Because of their ubiquity and portability, modern smartphones are often used to run security-critical programs along with diverse, untrusted, and potentially malicious programs. For example, most of us perform financial tasks, such as banking and payments [1] on our smartphones. Many of us also run health-related programs, e.g., to receive test results and diagnoses from our health providers. There is also interest in using these devices to perform life-critical tasks such as controlling an insulin pump [2] or monitoring breathing [3], although security concerns currently pose a roadblock [2].

Realizing this computing paradigm should be straightforward. The job of an operating system (OS) is to isolate security-critical programs from other programs running on the same hardware. Yet, this has proven to be challenging in practice due to vulnerabilities in system software (e.g., OS, hypervisor, and device drivers) [4–12] and hardware (e.g., processor, memory, interconnects, and I/O devices including their firmware) [13–19]. Malicious programs can exploit these vulnerabilities to take control of the machine and any program running on it. We must trust that hardware and system software can effectively sandbox and neutralize malicious programs, but this trust often proves to be misplaced.

To address this challenge, a new approach has emerged. It uses *Trusted Execution Environments (TEEs)* to host security-critical programs without requiring trust in the OS. Unfortunately, today's TEEs still require us to trust the hardware and the security monitor implementing the TEE guarantees. This trust has also proven unjustified. Existing TEEs have fallen victim to various attacks, e.g., hardware-based side-channel attacks [16, 20–28], attacks exploiting software vulnerabilities [29–32], and attacks based on design flaws [33–36].

In this paper, we present a solution to enable smartphones to be used for both security-critical and non-critical programs. Our goal is to minimize the Trusted Computing Base (TCB). More specifically, our goal is to minimize the number and complexity of hardware and software components that need to be trusted by the smartphone owner, when executing a security-critical program, to fend off adversarial inputs.

Our key principle is *provably exclusive access* to hardware and software components. That is, we design a solution to enable a security-critical program to *exclusively use complex hardware and software components and be able to verify the exclusive use.* The exclusive use of a component makes it unreachable to attackers.

More concretely, we present a hardware design for a smartphone. Called a *split-trust hardware*, it comprises multiple trust domains, one or multiple for TEEs, one for each I/O device, one for a resource manager, and one for hosting a commodity OS, e.g., Android, and its programs. The trust domains are *statically-partitioned* and *physically-isolated*: they each have their own processor and memory (and one I/O device in the case of an I/O domain) and do not share any underlying hardware components; they can only communicate by message passing over a hardware mailbox. Moreover, we introduce a few simple, *formally-verified* hardware components that enable a program to gain provably exclusive access to one or multiple domains.

[*] Equal contribution



*MobiSys '23, June 18–22, 2023, Helsinki, Finland*

ACM ISBN 979-8-4007-0110-8/23/06.
https://doi.org/10.1145/3581791.3596864

We then present OctopOS, an OS to manage this hardware. Unlike existing OSes, which have a single, trusted-by-all nucleus, i.e., the kernel, OctopOS comprises mutually distrustful subsystems: a TEE runtime for security-critical programs, I/O services, a resource manager, and a compatibility layer for a commodity, untrusted OS. The fundamental aspect of OctopOS is that components *do not trust, but verify* messages received from other components.

We rigorously evaluate the TCB of our machine. We show that it significantly reduces the TCB compared to mainstream TEEs and achieves one close to the lower bound.

We present a complete prototype of our machine (hardware and OS) on top of a CPU-FPGA board (Xilinx Zynq UltraScale+ MPSoC ZCU102). We use the powerful ARM Cortex A53 CPU to host the commodity, untrusted OS (PetaLinux) and its programs with high performance. We use the FPGA to build the other trust domains: two TEEs, a resource manager, and four I/O domains (an input domain, an output domain, a storage domain, and a network domain). We use (weak) microcontrollers for these other domains.

Using our prototype, we build two important security-critical programs for our machine:[1] (*i*) a banking program that can securely interact with the user, and (*ii*) an insulin pump program that can securely execute its algorithm and communicate with an (emulated) glucose monitor and pump.

Using our prototype, we show that the added hardware cost is small (i.e., 1-2%) compared to modern SoCs used in smartphones. Moreover, we show that *security-critical program can achieve usable performance despite the use of weak microcontrollers for all TEE and I/O domains*. We also show that *normal programs can achieve the same compute and I/O performance as on a legacy machine*, which is defined as a machine using the same powerful CPU as our untrusted domain but with that CPU being in full control of all I/O devices and main memory.

**Secure hardware trend.** Our vision of using physical isolation and exclusive use for security is in line with recent hardware trends from the smartphone industry. Apple has integrated the Secure Enclave Processor (SEP) into its products [37] and used it to secure user's secret data and to control biometric sensors (i.e., Touch ID and Face ID) [38]. Similarly, Pixel 6 uses the tensor security core to host security-critical tasks such as key management and secure boot [39]. Our work takes this vision further by allowing user-provided, third-party security-critical programs (including those that rely on I/O devices) to use dedicated hardware by developing a model for how that can be safely done.

## 2 TRUST IN EXISTING SYSTEMS

The TCB in a system comprises the hardware and software components that need to be trusted. Historically, the OS has been a trusted part of the system and hence part of the TCB (Figure 1 (a)). As commodity OSes have become more complex over the years, more and more vulnerabilities have been found in them, allowing malware to exploit them and compromise the OS [4–7, 9–12, 40]. As an example, there have been about 1700 security vulnerabilities reported in the Linux kernel just since 2016 [5]. Therefore, trust in commodity OSes is not warranted.

There have been several attempts to build trustworthy OSes. These include microkernels [41–45], exokernels and library OSes [46–49], formally verified OSes (and hypervisors) [44, 50–57], and OSes written in safe languages [58–61]. While effective, these

[1]We open source our hardware design and formal verification proofs at https://github.com/trusslab/octopos_hardware, and OctopOS and security-critical programs at https://github.com/trusslab/octopos.

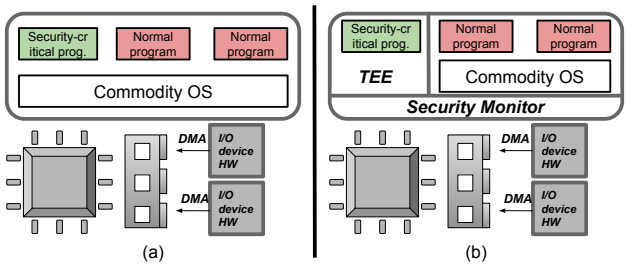


**Figure 1:** ***(a) Traditional design where the OS isolates security-critical programs from normal programs. (b) Use of a TEE to isolate a security-critical program.***

solutions require replacing commodity OSes with a new OS. This is a challenging task due to the abundance of existing programs, device drivers, and developers for commodity OSes. More importantly, using these OSes still requires trust in hardware, which is not warranted either, as we will discuss.

About two decades ago, a new approach started to gain popularity. The idea is to create an isolated environment, called a TEE, to host a security-critical program. This allows the use of a commodity OS, but relegates it to be only in charge of untrusted, normal programs such as games, utility apps, and entertainment platforms. The TEE enables a security-critical program to ensure its own integrity and confidentiality, but leaves the OS in charge of resource management (and hence the availability guarantee). Therefore, one does not need to trust the OS when running a security-critical program, reducing the TCB. Figure 1 (b) illustrates this design. It shows a *security monitor* is used to isolate a TEE from the OS. The security monitor can be implemented purely in software (i.e., a hypervisor) [62, 63] or using a combination of hardware and software. ARM TrustZone and Intel SGX are examples of the latter. Others include AMD Secure Encrypted Virtualization (SEV), Intel Trusted Domain Extensions (TDX), ARMv9's Realms [64], and Keystone for RISC-V [65].

Despite their success, existing TEE solutions still have a large TCB including the security monitor and several hardware components such as the very complex processor, memory, I/O devices in some cases, and dynamically-programmable protection hardware such as address space controllers and MMUs. Unfortunately, all of these components can be compromised by an adversary. For examples, hypervisors contain many vulnerabilities [8, 66]. TEE OSes in TrustZone have also contained vulnerabilities and have been exploited in the past [29–32]. AMD SEV has also been shown to contain several vulnerabilities due to design flaws [33–35]. AMD's response to these vulnerabilities have been enhanced versions of SEV, called SEV-ES and SEV-SNP. Unfortunately, these versions have also fallen to attacks exploiting side channels [28] or additional design flaws [36].

Hardware components have been exploited as well. Hardware-based side-channel attacks have recently emerged as a serious threat to computing systems. For example, SGX enclaves and TrustZone have been compromised using several such attacks [16, 20–27]. The core reason behind this is that existing machines run the untrusted OS and TEEs on the same hardware, sharing underlying microarchitectural features such as cache [20, 23–27] and speculative execution engine [14–16, 21], as well as architectural ones such as virtual memory [22]. The memory subsystem has also proved vulnerable to Rowhammer attacks [13, 67–71]. The complexity of these hardware components ensures that more vulnerabilities are

likely to be discovered and exploited. For example, researchers have recently demonstrated a suite of new side channels using the CPU interconnect [17], the x87 floating-point unit, and Advanced Vector extensions (AVX) instructions (among others) [18].

## 3 KEY GOAL AND PRINCIPLE

**Trust definitions.** We define two types of trust in the TCB: *strong trust* and *weak trust*. We say a component is strongly trusted if it needs to guard against *adversarial inputs*. An example is an OS that is trusted to isolate a program from other malicious programs, which can issue adversarial syscalls to the OS concurrently to the protected program. This component must be trusted to prevent these other programs from exploiting any vulnerabilities in it. This is challenging as demonstrated by the plethora of reported exploits.

We say that a component is weakly trusted if it just needs to operate correctly in the absence of adversarial inputs. An example is an OS that only serves a single program (and assuming application-level networking). This component must only be trusted to not exert buggy behavior under normal usage. This can be (more) easily achieved in practice.

Due to their obvious criticality, in this work, we focus on the strongly-trusted components in the TCB. For brevity, when talking about TCB, we mainly refer to these components.

Finally, we note that all components of the TCB need to be trusted not to have any backdoors implanted by an adversary.

**Key goal.** Our goal in this work is to minimize *both the number and complexity* of (strongly-trusted) components in the TCB. Our rationale for the former is obvious: the fewer trusted components, the better. Our rationale for the latter is that it is difficult for complex hardware or software components to adequately protect themselves against attacks; by contrast, simpler components can fend off attacks through comprehensive testing, analysis, and formal verification.

**Key principle.** Our key principle to achieve this goal is *provably exclusive access* to hardware and software components. That is, we design our machine to enable a security-critical program to *exclusively use complex hardware and software components and be able to verify the exclusive use*. More specifically, our goal is to have most components, especially complex ones such as the processor and system software, (1) be reset to a clean state before use, (2) then used exclusively by a security-critical program in a verifiable fashion through remote and/or local attestation, and (3) then again reset to a clean state right after use. In this case, such a component does not need to be (strongly) trusted anymore as it cannot be reached by an attacker while serving the security-critical program, nor does it need to worry about residual state from the security-critical program while serving other, potentially malicious, programs.

To realize this principle, we introduce a novel *split-trust hardware design* (§4). We then introduce an OS for this hardware, called OctopOS (§5).

## 4 SPLIT-TRUST HARDWARE

Modern machines leverage hardware with a *hierarchical privilege model*. That is, hardware provides multiple privilege levels, each with more privilege than previous ones, with one all-powerful level to "rule them all."[2] This model results inevitably in several complex components in the TCB such as the processor, protection hardware, and system software.

[2]A reference to Tolkien's The Lord's of the Rings.

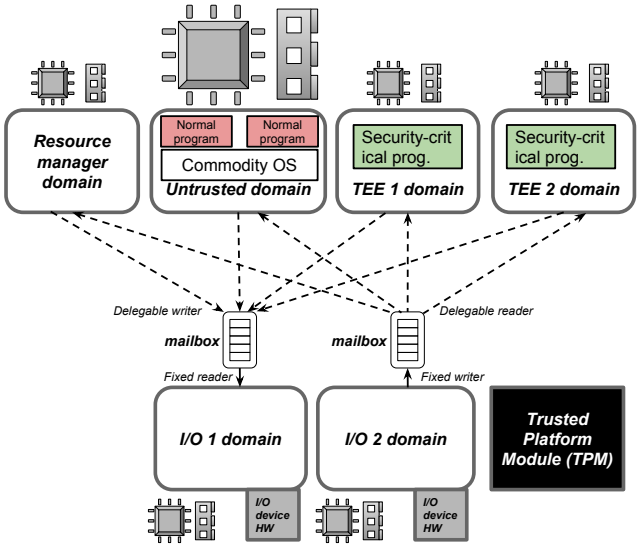


**Figure 2: *Simplified overview of the split-trust hardware. The figure does not show all mailboxes for clarity.***

In this paper, we demonstrate a novel hardware design, the *split-trust* hardware, in which the hardware is split into multiple isolated trust domains. Each domain is intended for one aspect of the machine: one or multiple for TEEs, one for each I/O device (i.e., an I/O domain), one for a commodity OS and its untrusted programs (i.e., the untrusted domain), and one for a resource manager, which is in charge of *constrained* resource scheduling and access control. The benefit of the split-trust hardware is that a security-critical program can *exclusively* take control of and use its own domain and *exclusively* communicate with other domains (§4.2), e.g., for I/O and IPC, hence significantly reducing the TCB. Figure 2 shows a simplified view of this hardware design. Next, we discuss its key aspects.

### 4.1 Physical Isolation & Static Partitioning

We follow two important principles in our hardware design. (1) Domains must be *physically isolated* (i.e., share no hardware components). (2) The isolation boundary between them cannot be programmatically and dynamically modified as *there is no trusted-by-all hardware or software component* to be tasked with that. This implies that we cannot rely on programmable protection hardware, such as an MMU, IOMMU, or address space controller, to enforce isolation. As a result, our design *statically partitions* the hardware resources between domains.

More specifically, each trust domain has its own processor. We use a powerful CPU for the untrusted domain, which accommodates a commodity OS and its (untrusted) programs, to achieve high performance. This CPU is similar to the powerful CPU used in modern smartphone SoCs. We use weaker microcontrollers for other domains in order to keep the hardware cost small. Each domain has its own memory as well and domains do not (and cannot) share memory.

Each I/O domain also has exclusive control of an I/O device, which is wired to and only programmable by the processor of that domain and which directly interrupts that processor. (We will discuss how DMA is handled in §4.5.)

### 4.2 Exclusive Inter-Domain Communication

To be able to act as one machine, the domains need to be able to communicate. We introduce a simple, yet powerful, hardware

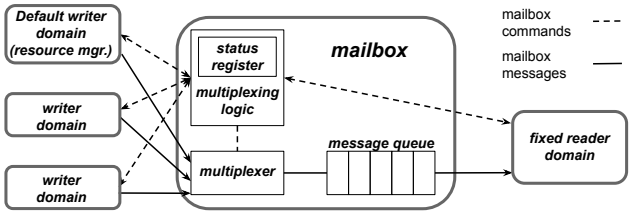


**Figure 3: *Mailbox design.***

primitive for this purpose: *verifiably delegable hardware mailbox*. At its core, a mailbox is a hardware queue, allowing two domains (i.e., the writer and reader) to communicate through message passing.

The key novelty of our mailbox is how it enables exclusive communication using its *delegation model*. A mailbox has a fixed end (reader or writer) and a delegable one. The fixed end is hardwired to a specific domain. The delegable one is wired to multiple domains, but only one can use it at a time, enforced by a hardware multiplexer within the mailbox. This end is by default (i.e., after a mailbox reset) under the control of the resource manager domain. But the resource manager can delegate it to another domain, which is then able to *exclusively* communicate with the domain on the fixed end of the mailbox.

Figure 3 shows the design of the mailbox with a fixed reader. For example, consider the serial output domain in our prototype. It is the fixed reader of a mailbox. Any domain with write access to the mailbox can (exclusively) send content to the output domain to be displayed in the terminal.

The delegation model of our mailbox has another important property: *limited yet irrevocable* delegation. When the resource manager delegates the mailbox to a domain, it sets a *quota* for the delegation in terms of both the maximum number of allowed messages and maximum delegation time. As long as the quota has not expired (i.e., *a session*), the domain can use the mailbox and the resource manager cannot revoke its access to the mailbox. The session expires when either the message limit or the time limit expires. (The message limit can be set to infinite, but not the time limit.)

This delegation model enables a limited form of availability, which we refer to as *session availability*. That is, a domain with exclusive communication access to another domain can be sure to retain its access for a known period of time or number of messages. This is critical for some security guarantees on smartphones. For example, a security-critical program can ensure that the User Interface (UI) will not be hijacked or covered with overlays when the program is interacting with the user [72, 73]. Or a security-critical program that has authenticated to and hence unlocked a sensitive actuator domain (e.g., insulin pump) can ensure that no other program can hijack the session and manipulate the actuator. We leverage session availability in our own apps (§7).

As the resource manager is not trusted by other domains, the delegation must be *verifiable*. The mailbox hardware provides a facility for this verification. As Figure 3 shows, all domains connected to the mailbox can read a status register from the mailbox hardware. The status register specifies the domain that can read/write to the mailbox and the remaining quota. The domain with delegated access can therefore verify its access and quota. (Other domains will receive a dummy value when reading the status register for confidentiality.)

Domains transmit both commands and data to each other through mailboxes. Because commands are typically short but data messages are typically long, we use two types of mailboxes to optimize the hardware design, namely *control-plane mailboxes* and *data-plane mailboxes*. These two types of mailboxes share the same hardware properties, but have different sizes (i.e., message size and queue size).

## 4.3 Power Management

Our mailbox primitive cannot, on its own, guarantee session availability. This is because we need to ensure that during a session, the domains used by a security-critical program remain powered up (given adequate energy in the battery).

The Power Management Unit (PMU) normally takes commands from the resource manager. The resource manager uses this capability to reset other domains when needed, e.g., reset a TEE domain before running a new program, or apply Dynamic Voltage Frequency Scaling (DVFS) to manage the system's power consumption. (We do not support DVFS for the domains in our prototype. Hence, in the rest of the paper, we mainly focus on the reset interface, although similar principles can be applied to DVFS.)

However, the resource manager is not a trusted component; hence it may try to reset a domain during a session. Therefore, we add a simple hardware component, called the *reset guard*, for controlling all the reset signals that are local to a domain, which ensures that as long as the quota on a mailbox has not expired, the domains on both sides of the mailbox (including the domain's mailboxes) cannot be reset, hence ensuring session availability. The resource manager simply fails to reset a domain if the domain has an ongoing delegation. Once the quota expires (or if the access to the delegable end of the mailbox is yielded), the mailbox is returned back to the resource manager, and the resource manager is allowed to reset and reuse the domains (assuming no other ongoing delegations).

## 4.4 Hardware Root of Trust

A hardware root of trust is needed during remote attestation to convince the party in charge of a security-critical program of the authenticity of the hardware and the correctness of the loaded program. We use a Trusted Platform Module (TPM) to realize the root of trust for the split-trust hardware.

**Why TPM?** TPM, as specified by the Trusted Computing Group (TCG), is a tamper-resistant security co-processor connected to the main processor over a bus [74]. Traditionally, it provides security features for the machine as a whole, such as the measurements of the loaded software. This makes TPM unsuitable for more fine-grained security features, such as remote attestation of a specific program. As a result, in-processor TEE solutions, such as SGX, integrate the root of trust in the processor itself, tightly coupling it with various features of the processor (such as virtual memory and cache), further bloating the trusted processor.

Our key insight is that TPM can provide fine-grained security features for a split-trust machine since different components of this OS run on separate processors. This allows the machine to enjoy the security benefits of TPM without suffering from its main limitation.

To integrate TPM into a split-trust machine, we need a different set of parameters (i.e., the number of Platform Configuration Registers (PCRs) and their access permissions, i.e., localities) from the ones found in existing TPM chips, in order to provide one PCR per domain and securely extend it with the measurement of software loaded in the domain. The bootloader of a domain measures the

boot image and extends the corresponding PCR with the measurement, and the PCR values are then used to provide a cryptographic proof of the software loaded into the domain (§5.1).

### 4.5 High Performance I/O

By default, the data plane of I/O domains are implemented over mailboxes. However, this raises a performance concern due to additional data copies (to and from mailbox). While the performance overhead is acceptable for TEE domains, it is not so for the untrusted domain. An important hardware primitive that enables a legacy machine to achieve high I/O performance is DMA. To safely use DMA in our machine, we introduce *domain-bound DMA*, defined with the following two restrictions. (1) The DMA engine is hard-wired to only read/write to the memory of the untrusted domain. (2) The DMA engine can stream data in/out of the I/O device only when the I/O domain is used by the untrusted domain.

We achieve this with a simple hardware component called the *arbiter*, which is a switch that decides if the data streams of the I/O device is connected to a DMA engine or to a simple FIFO queue accessible to the I/O domain.

### 4.6 Domain and Mailbox Reset

Domains and their mailboxes need to be reset before and after use (§3). We reset the mailboxes directly in hardware upon delegation, yield, and session expiration. We leave the resetting of the domains to the resource manager, albeit under the limitations enforced by the reset guard (§4.3). Even though the resource manager is untrusted, this does not pose a problem since a program can verify, using local and remote attestation through TPM as well as some measures provided by the domain runtime that (1) a domain has been reset, (2) it has not been used since last reset, (3) it will be reset after use and before use by other domains. We provide more details on the verification process with an example in §5.1.

## 5 OCTOPOS

We introduce OctopOS, an OS to manage the split-trust hardware. Unlike existing OSes, which have an all-powerful trusted-by-all kernel, OctopOS is composed of *mutually-distrustful components*. These components include I/O services for I/O domains, a runtime for TEE domains, a resource manager, and a compatibility-layer for the untrusted domain.

### 5.1 Fundamental Aspect

The fundamental aspect of OctopOS is that components *do not trust, but verify* any messages received from other untrusted components. We illustrate this aspect with one example.

Imagine a security-critical program that needs access to the input and output domains in order to interact with the user (e.g., to ask for username and password). The program, running in a TEE domain, sends a message to the resource manager and asks for the two domains to be delegated to it for a certain amount of time, e.g., one minute. More specifically, the program asks the resource manager to delegate the mailboxes of the input and output domain to the TEE domain. The resource manager waits for these domains to become available (if not at first), resets them, and then performs the delegation if it deems the request reasonable (e.g., if it is not for a very long period of time). It then responds to the TEE domain, confirming the successful delegation.

At this point, the security-critical program performs a series of verifications before it uses these domains. First, it uses the status register of the delegated mailboxes to verify that (1) its own domain is given exclusive access to the mailbox and (2) the delegation quota is correct (since otherwise the session might end abruptly, allowing the resource manager to hijack the program's interaction with the user). Second, the program needs to ensure that the right software has been loaded into the input and output domains and that the domains have been reset (otherwise the resource manager could install a keylogger/eavesdropper in these domains or simply inject code into them by exploiting their vulnerabilities). It performs the verification by checking the PCR value of each of the domains from the TPM. The PCR value provides a cryptographic proof of all the software loaded into a domain. Moreover, our I/O services further extend the PCR of their domain upon handling their first request. This way, the PCR value proves freshness (or lack thereof), i.e., that the domain has been reset prior to delegation.

Performing all these verifications on every interaction with other domains would be a daunting task, if it were to be done by the developer of a security-critical program. Therefore, OctopOS provides all of these for the developers in its components in the form of high-level API. We next discuss each of these components in more detail.

### 5.2 Components

*5.2.1 I/O Services.* Each I/O domain runs a service to manage it. The I/O service incorporates the software stack needed to program and use an I/O device, e.g., device driver. In addition, it provides an API that can be called (through messages) by any domain that has exclusive access to the mailboxes of the corresponding I/O domain.

There are two types of I/O devices. The first is *non-restricted I/O devices*. These are devices that can be used by a security-critical program without any restrictions during a session, such as the serial output and network devices in our prototype. For these devices, we ensure that the I/O domain is reset before and after use by another domain.

The second type is *restricted I/O devices*. These are devices that cannot be used freely by a security-critical program during a session and require the resource manager to enforce restrictions (i.e., fine-grained access control). In our prototype, storage is of this type since it contains data of other programs. Even if the data are encrypted, they need to be protected if a general availability guarantee is needed (§8.4). For these devices, we still ensure exclusive access to the domain during the session. We also ensure reset after use. However, we cannot ensure the domain is reset to a clean state before use. This is because after reset, the resource manager needs to communicate with the I/O service to restrict its usage, e.g., limit the storage domain to using only one partition allocated for a security-critical program, before delegating the domain's mailboxes to a TEE domain.

We have carefully designed an API for such I/O services. The core of the API revolves around the notion of an *I/O resource*. For example, in the case of the storage service, each partition is a resource. The API allows the manager to allocate resources and bind them to specific security-critical programs. It also allows the program to authenticate itself in order to use the resource and to verify the status of the service. We omit the details of the API due to space limitation.

Finally, we note that this design adds the storage service to the TCB when it is used by a security-critical program (§8.4). In contrast, other I/O services are not directly reachable by the adversary when used exclusively by a TEE domain and hence are not part of the TCB.

*5.2.2 TEE Runtime.* In order to facilitate the development of security-critical programs, we have developed a runtime for TEEs,

which provides a high-level API. A program may choose to utilize this runtime (which is part of the TCB), or its own.

We provide several categories of functions in this API: (1) Requesting and verifying access to other domains; this category also helps the program manage the remaining quota of mailboxes by calling a callback function upon quota updates, so that the program can decide whether to continue using the mailbox or not. It depends on the program's security goals to notify the user that the quota is about to expire. (2) High-level abstractions for using I/O services such as socket-based networking and terminal prints. (3) Assistance with the TPM, e.g., to request a remote attestation report. (4) Support for secure IPC between TEE domains. (5) Security-critical routines such as cryptographic primitives.

*5.2.3 Resource Manager.* At a high level, the resource manager is in charge of resource scheduling, access control, and system-wide, untrusted I/O functionalities. More specifically, it performs the following three tasks. First, it makes constrained scheduling decisions. When a new security-critical program needs to execute, or when an existing one requests exclusive communication with another domain (for I/O or IPC), the manager checks the availability of resources, grants the request, or blocks it until the resource is available. Compared to schedulers in commodity OSes, scheduling in OctopOS is more restricted. This is because the resource manager cannot preempt a domain as long as mailbox quotas have not expired (§4.2). Second, the resource manager restricts the usage of some I/O domains to enforce fine-grained access control, as discussed in §5.2.1. Finally, the manager implements system-wide, untrusted I/O functionalities. For example, as the manager is the initial client of the input and output domains, it implements the shell (i.e., the UI). The UI, however, can be delegated to security-critical programs upon request.

*5.2.4 Untrusted Domain's Compatibility Layer.* In OctopOS, a commodity OS runs in the untrusted domain, and hence by definition manages its own processor and memory. (In contrast, OctopOS is in charge of managing all the domains and their interactions with each other.) Yet, the commodity OS is not given direct control of I/O devices as they are managed by separate I/O domains.

We address this issue by developing a compatibility layer for the untrusted OS. In our prototype, which uses PetaLinux, the compatibility layer consists of several kernel modules, each pretending to be a device driver. Transparent to Linux and its program, they communicate to the resource manager to get access to the I/O services' mailboxes (or to set up DMA) and then communicate to them. These Linux drivers can be used to run Android in the untrusted domain as well.

# 6 PROTOTYPE

We have built a prototype of the split-trust hardware and OctopOS on the Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA board. We use the Cortex A53 ARM processor on the SoC for the untrusted domain in order to achieve high performance for the commodity OS (PetaLinux) and its programs. We use the FPGA to synthesize 7 simple Microblaze microcontrollers (i.e., no MMU and no cache): two TEE domains, the resource manager domain, and four I/O domains (serial input, serial output, storage, and Gigabit Ethernet). (Note that we are limited to I/O devices in the development board and hence could not use more smartphone-specific I/O devices such as WiFi. However, our principles and approaches apply equally to these other I/O devices as well.) We leverage the (single-threaded) Standalone library [75] to program the microcontrollers. We use the entirety of the main memory for the untrusted domain. For other domains, we use a total of 3.2 MB of on-chip memory including some ROM for bootloaders and some RAM. We run the TPM (emulator) [76] on a separate Raspberry Pi 4 board connected to the main board through serial ports. We use another Microblaze microcontroller to mediate the communications of the domains with the TPM.

In addition, we use the FPGA to synthesize the mailboxes (12 in total), the arbiter for DMA for the network domain (other domains do not support DMA), the reset guard, as well as 11 hardware queues for permanent domain connections (such as for all domains to communicate with TPM or for TEE domains to communicate with the resource manager). The control-plane mailboxes have the capacity of 4 messages of 64 B each, and the data-plane mailboxes have the capacity of 4 messages of 512 B each. As a concrete example, our storage domain has 4 mailboxes: two for its control plane (send/receive) and two for its data plane (send/receive).

As mentioned in §4.1, an I/O device is only programmable by its domain. This includes access to registers and receiving interrupts from the I/O device. In our prototype, we use I/O interrupts only for the network device and use polling for the rest. The interrupts to the network domain's microcontroller is from the FIFO queue that holds the packets and are only used when the domain serves a TEE domain (§4.5). When serving the untrusted domain, the domain-bound DMA engine directly interrupts the A53 processor on DMA completion.

We faced two noteworthy limitations in our prototype. First, while we have strived for our domains to share no hardware, currently, all our domains share the same clock source and our FPGA-based domains share the same power domain. Second, the on-board SD card reader and flash memory are directly programmable by the A53 processor and hence could not be used for the storage domain. Our solution was to connect a MicroSD card reader directly to FPGA through Pmod [77]. This provides physical isolation for the storage domain, but significantly degrades its performance due to Pmod's limited throughput. Therefore, for performance evaluation, we instead use DRAM as our storage (we partition out a chunk of DRAM and use it exclusively for the storage domain). This allows us to stress the performance of the mailboxes of the storage domain and get an upper bound for our storage performance, which we cannot do with the Pmod prototype.

We note that requiring an FPGA board to experiment with our machine may pose a road block for many researchers. Therefore, we also develop an emulator for our hardware design. The emulator runs on a Linux-based host OS such as Ubuntu and is able to fully boot and run OctopOS.

Overall, we have implemented OctopOS and our hardware emulator in about 39k lines of C code (including 5k of modified drivers from Xilinx and crypto libraries). We report the LoC for our hardware below.

## 6.1 Verified Hardware Design

The split-trust hardware has only four simple hardware components that are part of the TCB (§8.4): mailbox, DMA arbiter, reset guard, and ROM (for bootloaders). We have implemented these components in 1630 lines of Verilog code as well as 800 lines of Python code.

The simplicity of our trusted hardware components enables us to formally verify them. We use SymbiYosys to perform formal verification [78]. SymbiYosys is a front-end for Yosys-based formal hardware verification flows. We took a pragmatic approach to infer 20 theorems (some comprising multiple lemmas) from our

| Property | Proved theorems |
|---|---|
| Mailbox exclusive access | Domains w/o exclusive access to mailbox cannot change which domain has exclusive access, nor the remaining quota. |
| | If a domain does not yield its exclusive access, its exclusive access is guaranteed as long as the quota has not expired. |
| | The domain with exclusive access to the mailbox can correctly read or write from/to the queue. |
| | The domains w/o exclusive access to the mailbox cannot read/write to the queue. |
| Mailbox limited delegation | When given exclusive access, a domain cannot use the mailbox more than its delegated quota. |
| | When the quota delegated to a domain expires, the domain loses exclusive access. |
| Mailbox verifiable excl. access | The domain with exclusive access can correctly verify its exclusive access and remaining quota. |
| | The domain on fixed end of mailbox can correctly verify domain with excl. access on the other end and remaining quota. |
| Mailbox default excl. access | After reset, the resource manager domain has exclusive access by default. |
| | The resource manager domain does not lose its exclusive access unless it delegates it. |
| | When a domain loses excl. access (yield/expiration), the excl. access will be given to the resource manager domain. |
| Mailbox confidentiality | Domains w/o excl. access cannot use mailbox's verif. interface to learn which domain has excl. access and remain. quota. |
| | Upon delegation/yield/expiration, the data in the queue is wiped. |
| Reset Guard | The reset signal does not get forwarded if any other domain is using one of the domain's mailboxes. |
| | The reset signal does not get forwarded if the domain is using any of the other domain's mailboxes. |
| Arbiter control | The control interface can change its state between trusted and untrusted. |
| | Nothing other than the control interface can change the arbiter's state. |
| Arbiter excl. access | When an arbiter is connected to a trusted domain, a mailbox can correctly read or write data. |
| | When an arbiter is connected to an untrusted domain, a DMA engine can correctly read or write data. |
| ROM | A memory can be transformed into read-only access, a change that is irreversible. |

**Table 1:** ***Theorems we prove for our hardware components. Proving some of these require proving lemmas not listed here.***

guarantees. Formal verification ensures that our hardware design satisfies these theorems and hence our guarantees. Indeed, we have discovered and fixed a delegation logic error during verification.

We use the SMTBMC engine, which uses *k*-induction to formally verify our hardware design against these theorems. Table 1 shows the list of theorems we prove for our hardware components. Overall, we developed 3000 lines of SystemVerilog code for our hardware verification. We describe all the theorems in a separate document, which can be found in our hardware repository[3]. Below, we present one example.

**Theorem example.** We demonstrate the Verilog code (adjusted for readability) that we develop for verifying the theorem that "when the quota delegated to a domain expires, the domain loses exclusive access" (Table 1 Row 6). As specified by the pseudo-code below, the SMTBMC engine proves that on the rising edge of a clock cycle, when either the time limit or quota limit becomes zero, the new owner is determined to be the resource manager.

In lines 5-6, the "q_expired" register compares the remaining quota limit with zero, and in lines 7-8, the "t_expired" register compares the remaining time limit with zero. In both cases, the expired registers are not triggered if the current owner is the resource manager. Line 9 checks if the time limit or quota limit has expired, and if so, the new owner must be the resource manager.

```
1  reg init = 1;
2  always @(posedge clk) begin
3    if (init) assume (!aresetn);
4    if (aresetn) begin
5      q_expired <=
6        (remain_quota == 0) && (owner != `ID_RM);
7      t_expired <=
8        (remain_time == 0) && (owner != `ID_RM);
9      if (t_expired || q_expired)
10       assert (owner == `ID_RM);
11   end
12   init <= 0;
13 end
```

[3]https://github.com/trusslab/octopos_hardware/raw/main/docs/OctopOS-TRM.pdf

## 7 SECURITY-CRITICAL PROGRAMS

We discuss two security-critical programs that we have built for our machine. These programs are simplified yet representative of real-world applications.

**I. Secure banking.** Our secure banking program allows a user to securely log in to their account and view their account balance. The program leverages several features of our machine. First, it uses exclusive access to the UI (i.e., shell) as well as our session availability guarantee to make sure all inputs come from the user (and not malware) and that outputs are only displayed to the user. On legacy machines that do not support session availability, it has been shown that user's interaction with a banking app can be hijacked or covered with overlays [72, 73, 79]. Upon getting exclusive access to the UI, the program needs to convince the user that they are interacting securely with the program. It does so by displaying a secret established *a priori* between the user and the bank. Moreover, the program utilizes the runtime APIs to monitor the quota left for the UI session, and prompts the user to stop interacting with the program if the quota is low.

Second, the program uses exclusive access to the network domain to transfer confidential information. One might wonder why it is not adequate to use a secure networking protocol, such as TLS, for this purpose. Such protocols leave open some side-channel attack vectors [80], which our exclusive network access closes against on-device attackers; external network side-channel attacks are still possible. Note that a secure networking protocol is still needed for protecting the data against adversaries outside our machine (although we have not incorporated such a protocol in our prototype yet).

Finally, the program uses remote attestation to enable the bank server to verify the integrity of the program running on the user's device before any sensitive account information is released or any commands are accepted. Specifically, (1) the server provides the program with a challenge (i.e., a nonce), and the program passes the challenge to the TPM, which generates an attestation report.

(2) The program sends the report to the server, which verifies it and then sends the expected PCR values of the I/O services to the program, (3) which then uses them for local attestation of I/O domains (including that of the network service).

**II. Secure insulin pump.** Diabetic patients need to administer insulin to control the glucose level in their blood. New glucose monitor and insulin pumps have recently emerged that can be programmed through a smartphone, although security concerns currently requires using a dedicated smartphone [2]. (We note that some patients use an open source, unofficial Android app [81] to control the pump, albeit at their own risk.) Our machine can enable the use of user's own smartphone to securely execute these life-critical tasks.

We build two versions of this security-critical program in our OS. The first version allows the user to directly program the insulin pump (in which case a glucose monitor is not used). The second version automatically reads the user's glucose level and uses that (and previous historical readings) to decide how much insulin to pump.

These programs leverage our session availability and exclusive access to the insulin pump (and the glucose monitor in the second version of the app), e.g., via Bluetooth or through the headphone jack. This way, the program can securely authenticate itself to these devices and not worry that the session may be hijacked. The program also uses exclusive access to the network domain to securely communicate with the health provider's server, which uses remote attestation to enable the provider's server to trust the program, similar to our secure banking program. Finally, the second version of this program needs to be executed in fixed intervals and store its sensor readings across sessions. This requires a stronger availability guarantee, called *general availability* (as opposed to the more limited session availability). For this, it trusts the resource manager and the storage domain, as discussed in §8.4. The first version does not need the additional trust since it only requires session availability.

# 8 TCB & SECURITY ANALYSIS

## 8.1 TCB Notation

We introduce and use a simple, compact notation for TCB, discussed here with an abstract example:

$$\texttt{Owner} \quad \overset{\texttt{G1,G2}}{\top}\ \texttt{CompA(1), CompB(2)} \ \cup\ \overset{\texttt{G3}}{\top}\ \texttt{1,2,CompC(3)}$$

The key operator is the $\top$ sign, which resembles a T (as in Trust). It helps denote *the set of (strongly-trusted) components in the TCB*. The elements on top of the $\top$ sign, e.g., `G1`, are the security guarantees, e.g., confidentiality and integrity. This allows for differentiating trust assumptions for different guarantees and combining them using the $\cup$ sign. The elements in front of the $\top$ sign are the trusted components. For succinctness, we tag a repeating component with a number in parenthesis on its first appearance and use the number in other locations.

## 8.2 Lower Bound of TCB

Assuming that the program communicates with the outside world, the lower bound can be achieved if the machine is dedicated to executing a security-critical program:

$$\texttt{Owner} \quad \overset{\texttt{C,I,A}}{\top}\ \texttt{Prog.,RoT}$$

where `C`, `I`, `A` stand for Confidentiality, Integrity, and Availability. This shows that the owner at the very least needs to trust the (security-critical) program and the Root of Trust (`RoT`). The trust in the program is fundamental: the program needs to protect itself against adversarial inputs, e.g., malicious network packets. (This could imply trust in the network interface card. However, we assume that the network interface card is isolated by the program, e.g., using an IOMMU). The program in the TCB includes the runtime used by the program to interact with the hardware.

The trust in the RoT is also fundamental and stems from the fact that an adversary controlling the machine may try to fool the verifier of remote attestation by attempting to attack and compromise the RoT. The trust in the RoT includes trust in the bootloader, the ROM used to store the bootloader, the hardware/firmware used for remote attestation, e.g., TPM, as well as the hardware vendor that certifies attestation reports.

Finally, note that we do not consider the processor to be part of the TCB because the program can sanitize the adversarial inputs and prevent them from reaching the processor in a meaningful way.

## 8.3 TCB of Existing Systems

First, we consider a traditional system that uses an OS to provide isolation:

$$\texttt{Owner} \quad \overset{\texttt{C,I,A}}{\top}\ \texttt{Prog.,OS,Proc.,Mem.,I/O,interconn.,P.HW,RoT}$$

This shows that the owner needs to trust the hardware including the processor, memory, I/O devices, protection hardware (`P.HW`) such as MMU and IOMMU, and interconnects. Moreover, the OS is also trusted, including device drivers. In this case, the program includes the libraries used by the program to interact with the OS and hardware.

Next, we write the TCB for a popular TEE solution for smartphones, TrustZone, in Formula 1. `SM` is the security monitor (i.e., the secure world OS and monitor code). We note that TrustZone allows the secure world to take full control of an I/O device, i.e., secure I/O (`Sec-I/O`). Yet, this device and its driver are exposed to multiple programs in the secure world and hence are trusted. Another noteworthy issue is that, in general, the OS is trusted when availability is needed as it is in charge of resource scheduling. However, in TrustZone, the secure world OS (part of the `SM` in the formula) can be configured to handle some of the interrupts and hence can control the availability of the corresponding resources [82].

## 8.4 Our TCB

Formula 2 shows the TCB of our machine. `As`, `Ag`, `SD`, and `RM` stand for session availability, general availability, storage domain, and resource manager, respectively. Our system requires trust in a few cases that were not part of the lower bound. First, for confidentiality, integrity, and session availability, the owner needs to trust the mailboxes used by the program, the arbiter (if domain-bound DMA is used), and the domain reset guard as these components interact with untrusted components. As discussed in §6.1, the simple design of these components allowed us to formally verify them, making this trust acceptable. Second, if a program needs general availability guarantees (e.g., it needs to be executed in fixed intervals) and needs to store data across sessions, it needs to trust the resource manager domain and the storage domain. The only way to eliminate the trust in the storage domain for general availability is to have separate storage devices for each security-critical program. Unfortunately, this is prohibitively expensive. Note that we assume that the program protects the confidentiality and integrity of its stored data using proper cryptographic primitives, although we have not implemented that in our prototype.

$$\text{Owner} \overset{\text{C,I}}{\top}_{\text{Prog.(1),SM(2),Processor(3),Mem.(4),Sec-I/O(5),interconn.(6),P.HW(7),RoT(8)}} \cup \overset{\text{A}}{\top}_{\text{1,2,3,4,5,6,7,8,OS}} \quad (1)$$

$$\text{Owner} \overset{\text{C,I,As}}{\top}_{\text{Prog.(1),mailbox(2),reset-guard(3),arbiter(4),RoT(5)}} \cup \overset{\text{Ag}}{\top}_{\text{1,2,3,4,5,RM,SD}} \quad (2)$$

It is noteworthy that our machine eliminates the need to trust several complex hardware and software components such as the processor, memory, I/O devices, the interconnects (since our machine does not share any buses between trust domains) and system software (security monitor, OS, and device drivers), compared to existing TEEs. Overall, the TCB of our machine is significantly smaller than modern, popular TEEs. Moreover, our TCB is rather close to the lower bound. Achieving a smaller TCB for a machine that can host security-critical and untrusted programs concurrently would be challenging.

## 8.5 Security Analysis

**Threat model.** We assume an attacker can run malicious programs in the machine and tries to exploit any software or hardware vulnerabilities. We also assume that adversary can send malicious packets over the network to the machine. Below, we discuss various such attacks and their implications. Physical attacks are out of scope.

**Software vulnerability-based exploits.** Vulnerabilities in trusted software components would lead to attacks. An attacker that compromises the program can obviously change its behavior. An attacker that compromises the bootloader (including the code that cleans up the state in a domain upon reset) can falsify the remote attestation report or access/impact data from other sessions. An attacker that can compromise the storage service can delete the program's data. An attacker that can compromise the resource manager can starve the program of resources (but cannot impact the availability of a session once it is granted). An attacker that manages to compromise other software components, e.g., I/O services, other security-critical programs, and the untrusted OS, cannot mount an attack on the program.

**Hardware vulnerability-based exploits.** In a split-trust machine, unlike existing TEEs, vulnerabilities in many complex hardware components such as the processor cannot be exploited since the adversary never shares the underlying hardware with the security-critical program. Therefore, the attacker cannot leverage various hardware-based attacks such as cache side-channel attacks, interconnect side-channel attacks, speculative execution attacks, and Rowhammer attacks. Only vulnerabilities in the trusted hardware components (i.e., mailbox, arbiter, reset guard, ROM, and TPM) would lead to attacks. The first four are formally verified (§6.1) and TPM is a mature and secure technology.

**Timing side-channel attacks.** All trusted software and hardware components are vulnerable to timing side-channel attacks. In our machine, the only components that may expose useful timing channels are the TPM and the program runtime. Such attacks (and others) have been demonstrated on TPMs before [83–87]. As TPM is a mature technology, vulnerabilities get fixed. Indeed, there have been several works that formally verify various aspects of the TPM standard [88–90]. We have not analyzed the timing channel of the runtime we have developed for security-critical programs.

**Power management attacks.** These types of attacks can induce faults in the victim program's execution by manipulating the frequency or voltage of the processor and have been demonstrated against TEEs [91–93]. As mentioned in §4.3, our machine does not allow power management of a domain in a session, and hence mitigates such attacks.

Power management data can also be used as a side channel. More specifically, an attacker may try to monitor the voltage and frequency of a domain (which changes according to DVFS) and use that as a side channel to extract secrets from a domain. We note that our current prototype is not vulnerable to this side channel since our TEE domains do not support DVFS. However, our hardware can support the use of DVFS-capable processors for TEE domains. In such a case, we will need to close this channel. To do so, we will need to ensure that the PMU does not leak any information about a domain to another domain. This can be done rather trivially within the PMU firmware, which should be formally verified and hardened.

Hertzbleed [94] turns a power side channel into a timing attack. We leave it to the program and its runtime to migitate such an attack.

**Remote network attacks.** Similar to a legacy machine, a security-critical program must protect itself against malicious network messages in our machine. However, our machine provides some protection against network attacks that target the network stack. This is because it sandboxes the network device and its device driver in its own domain. As a result, programs that do not use the network at the time of a exploit are protected from these attacks. This is in contrast to a legacy machine in which a single successful exploit of the kernel-based network stack may result in a full takeover.

**Out of scope: physical attacks.** We assume that the adversary does not have physical access to the device. Therefore, we do not protect against physical attacks. However, if the program does not use any I/O devices, it can use on-chip computation and memory encryption to protect its secrets against physical attacks [95–97]. These are orthogonal to our design and hence can simply be added to our machine. However, we note that if a program uses I/O devices, no general solution can be used to prevent physical attacks. While storage and network devices can use encryption (i.e., full-disk encryption), other devices such as output devices, cameras, sensors, and actuators cannot be universally protected.

# 9 EVALUATION

Our FPGA-based hardware implementation serves two purposes. First, we use it to estimate the hardware cost of our solution in terms of chip area. Second, it provides a bound on the performance impact of the solution. A deployed solution would likely replace the FPGA components with higher-performance non-reprogrammable ASIC elements, such as an integrated SoC or specialized chiplets [98].

However, despite the use of FPGA and weak microcontrollers for TEE and I/O domains, we show that security-critical programs can achieve decent performance, while normal programs can achieve the same compute and I/O performance as on a legacy machine.

| FPGA resource | Count | Equivalent transistor count |
|---|---|---|
| Look-up table | 69,999 | 2,519,964 |
| Flip flop | 63,188 | 1,516,512 |
| Block RAM | 27,061,649 (bits) | 162,369,894 |

**Table 2: *Extra hardware cost in our machine.***

| Configuration | Throughput (MB/s) | Latency ($\mu$s) |
|---|---|---|
| A53-Microblaze | 7.07±0 | 18.2±0 |
| Microblaze-Microblaze | 9.64±0.01 | 15.26±0.05 |

**Table 3: *Mailbox performance.***

## 9.1 Hardware Cost

We calculate an estimate for the number of transistors needed for our additional hardware components (all the components synthesized on the FPGA in our prototype). We calculate this estimate by measuring the number of look-up tables, flip flops, and block RAMs used by our hardware and converting them to transistor count using the following estimates: 6 NAND gates per look-up table [99], 6 transistors per NAND gate [100], 24 transistors for each flip flop [101], and 6 transistor for each bit of on-chip memory (assuming a conventional 6-transistor SRAM cell [102]). Our calculation shows that our machine requires about 166.4 M additional transistors (162 M of which are used for on-chip memory). Table 2 shows the breakdown. This compares favorably with the number of transistors used in modern SoCs in smartphones. For example, Apple A15 Bionic and HiSilicon Kirin 9000 use 15 B transistors [103, 104]. This means that, if our solution is added to an SoC or implemented as a chiplet [98], the additional hardware cost would likely be 1-2%.

## 9.2 Performance

We measure various performance aspects of our machine. Note that all domains except the untrusted one use an FPGA with a 100 MHz clock. The Ethernet controller IP uses an external 50 MHz clock. Therefore, our results represent a lower bound on our machine's performance; we expect superior performance on ASIC. We repeat each experiment 5 times and report the average and standard deviation.

**Mailbox performance.** We measure the throughput and latency of communication over our mailbox. For throughput, we measure the time to send 10,000 messages of 512 B over a data-plane mailbox. For latency, we measure the round trip time to send a 64 B message and receive an acknowledgment over a control-plane mailbox. We perform these experiments in two configurations: one for communication between the hard-wired ARM Cortex A53 (the untrusted domain) and an FPGA-based Microblaze microcontroller, and one for communication between two FGPA-based Microblaze microcontrollers. Table 3 shows the results. One might wonder why the A53-Microblaze configuration achieves lower performance. We believe this is because this configuration requires the data to pass the FPGA boundary, hence passing through voltage level shifters and isolation blocks [105]. Moreover, the FPGA is in a different clock domain than A53.

**Storage performance.** We measure the performance of our storage domain, which uses the mailbox for its data plane (i.e., no DMA). To do so, we perform 2000 reads/writes of 512 B each. We evaluate three configurations: a best-case configuration where the storage domain directly performs reads/writes (hence giving us an upper bound on the DRAM-based storage performance), and two configurations where the untrusted domain or a TEE domain uses the storage service over the domain's mailboxes. Table 4 shows the results. They show that our mailbox-based storage domain can achieve decent performance (as can also be seen from our boot-time measurements reported below). It also shows that the additional copies caused by the mailbox add noticeable overhead compared to the best-case scenario. To further improve this performance for the untrusted domain, one can use domain-bound DMA for the storage domain.

| Configuration | Read throughput (MB/s) | Write throughput (MB/s) |
|---|---|---|
| Best-case | 8.13±0.00 | 6.10±0.00 |
| Untrusted dom. | 4.17±0.09 | 4.06±0.00 |
| TEE domain | 4.39±0.00 | 3.93±0.00 |

**Table 4: *Storage performance.***

| Configuration | Throughput (Mbit/s) | RTT (ms) |
|---|---|---|
| Baseline | 943±0 | 0.17±0.01 |
| Untrusted domain | 943±0 | 0.17±0.02 |
| TEE domain | 0.567±0.001 | 23.92±0.02 |

**Table 5: *Network performance.***

**Network performance.** We measure the performance of our network domain, which uses domain-bound DMA for high performance for the untrusted domain (§4.5). We evaluate three configurations, similar to those used for storage experiments. For measuring the throughput for the baseline and the untrusted configurations, we use iPerf; for round-trip time (RTT) measurements, we use Ping. For the TEE configuration, we develop custom programs for measurements. For all experiments, we connect the board to a PC, which acts as a server. Table 5 shows the results. They show that our domain-bound DMA is capable of matching the performance of a legacy machine. Moreover, the network performance for a TEE is usable.

We believe, based on some tests that we have conducted, that it is possible to further improve the TEE network performance by about 10 X. This is because, currently in the network domain, we add an artificial delay between accessing the mailbox and the network IP, which limits performance. We do so to prevent data corruption, which according to our extensive investigation, is caused by a bug in the Ethernet AXI IP from Xilinx (potentially the bug discussed in [106]). Since the IP is closed source, we are not able to fix the bug.

**Boot time and breakdown.** We measure the boot time of our machine. All the boot images are transferred from the storage domain to their corresponding domains over mailboxes. Due to presence of multiple domains, booting OctopOS from a partition in the storage service is a carefully choreographed dance, requiring steps taken by bootloaders in each domain and the resource manager. Due to space limitations, we do not provide the details of the boot process, but measure and report it. Our measurements show that it takes 4.03±0.00 s to boot all domains excluding the untrusted domain, which takes an additional 8.65±0.01 s to boot.

**Untrusted program performance.** We use the network file system to evaluate the performance of an untrusted program. Our benchmark reads 100 files each containing 10,000 random numbers from a network file system, sorts them, and writes them back to the same file system. We choose this benchmark since it stresses CPU, memory, and network (for which we have domain-bound DMA). Our evaluation shows the benchmark takes the same amount of time (3.86±0.03 s) on our machine as on a legacy machine with the same A53 processor, RAM, and Gigabit Ethernet (3.84±0.04 s).

**Security-critical program performance.** We measure the execution time of two security-critical programs. In our experiments, we assume that no other domain needs and hence competes for the I/O domains. This allows for simple optimizations, e.g., proactively resetting the network domain.

The first program is secure banking (§7). We measure the time it takes to launch, including time needed to acquire keyboard, serial, and network, to perform attestation, and finally, to display prompts for user credentials. Our measurements show the overall execution time is 2.38±0.42 s.

We also develop and evaluate a more performance-intensive security-critical program. It reads a 1 MB file from the storage domain, computes its hash, and sends the hash over the network to a server. The overall execution time is 1.75±0.00 s. Looking at the breakdown, it takes 0.30±0.00 s to launch (including time needed to acquire exclusive access to storage and network, excluding local attestation through TPM), 0.22±0.00 s to read the file from storage, 1.21±0.00 s to compute the hash, and 0.01±0.00 s to send the hash over the network. To better assess this execution time, we write a normal program to perform similar tasks on a legacy machine with the A53 processor, RAM-FS, and Gigabit Ethernet. This program takes 0.23±0.00 s to execute.

If an I/O domain is in use when we run the security-critical program, there will be two types of additional delay. First, our security-critical program needs to wait for the I/O domain to become available. Second, in the case of the network, the program needs to wait for the network domain to perform ICMP route discovery and other network protocols, which can take around 4.12±0.96 s in our current prototype (without any optimizations). But note the app can mitigate part of these delays by overlapping them with other parts of its execution.

**Programming effort.** We evaluate the programming effort for both types of developers. We report the programming effort required to develop a security-critical program on top of OctopOS. Currently, the runtime provides 49 APIs for the application developers to use. The secure banking program presented in §7 has 482 lines of code, which includes 58 lines for the main logic, 107 lines for the user interface, 207 lines for network communication (including attestation), and 93 lines for managing delegated resources. The secure insulin pump program (second version) has 563 lines of code, which includes 217 lines for the main logic, 200 lines for network communication (including attestation), and 128 lines for managing delegated resources.

The network domain has 7217 lines of code (including modified drivers from Xilinx). The storage, keyboard, and serial domain have 1091, 154, and 165 lines of code, respectively. These numbers exclude the domains' bootloaders and lower-level OctopOS code for hardware support, such as our mailbox driver, which an I/O service developer can reuse.

**Impact of exclusive I/O use.** We evaluate the impact of executing a security-critical program that uses storage on the storage performance of the untrusted domain. More specifically, we launch a security-critical program in a TEE that exclusively reads/writes 1 MB from/to storage, while the untrusted domain is reading a 100 MB file data (which normally takes 24.26±0.31 s to finish). Our measurements show that the security-critical program causes a 2.58±0.03 s gap where the untrusted domain cannot access storage.

## 9.3 Energy Consumption

We estimate the energy consumption of running security-critical programs on our hardware. We measure the actual execution time of each domain, and multiply the time by the per-domain power estimation. The estimation is obtained by running the power report program on our hardware design using the Xilinx Vivado software.

Our measurements show the energy consumption of all the domains involved in launching the banking program (including booting, initialization, requesting resources, and performing attestation) is 3.21±0.64 Joules.

We also measure the energy consumption of the other security-critical program that reads a 1 MB file, hashes it, and sends the hash over network. The energy consumption of all the domains that are involved is 2.03±0.42 Joules. In comparison, we measure the estimated energy consumption of the 1 MB file hashing experiment on the legacy machine. Xilinx Vivado software estimate the runtime energy of the A53 processor on the SoC of our FPGA board to be to be 2.74 watts (with no DVFS). We calculate the energy consumption of the program running on the legacy machine to be 0.63±0.00 Joules.

To provide a frame of reference, note that the overall amount of energy in a fully charged battery in a modern smartphone (i.e., Google Pixel 7) is 60517 Joules.

# 10 RELATED WORK

**Physical isolation & static partitioning**. Notary [107] safeguards approval transactions by running its agent on a separate SoC from the ones running the kernel and the communication stack. Our work shares the idea of using physically-isolated trust domains and also resets the domains before and after use by other programs. In contrast, we show how to safely mediate access to shared I/O devices for a workload of concurrent security-critical and untrusted programs.

Likewise, I/O-Devices-as-a-Service (IDaaS) suggests that I/O devices should have their own separate microcontrollers (and observes that they often do) and advocates for hardening their interfaces against potentially malicious kernel behavior [108]. Our approach also uses separate I/O microcontrollers but does not require trust in the microcontroller software, by resetting the I/O domain between uses.

**Exclusive use.** Flicker [109] uses the late launch feature of Intel Trusted Execution Technology (TXT) [110], to exclusively run a program on the processor. The exclusive use of the hardware results in minimizing the trusted components. However, Flicker's design requires stopping all other programs (including untrusted ones) when running a security-critical program. Our approach can run untrusted programs and security-critical programs concurrently (albeit with the limitation that I/O domains cannot be shared). Consider our secure insulin pump program (§7), which might need to be run frequently while the user is actively doing other, less security-critical, tasks on the main processor. Realizing this in Flicker can result in significant disruptions to other programs and to the user as a result.

**Secure I/O for TEEs.** SGXIO uses a hypervisor and a TPM to create a trusted path for an SGX enclave to access an I/O device [111]. The solution requires the enclave program not only to trust SGX's firmware and hardware, but also the hypervisor. Cure [112] adds a few hardware primitives in order to allow the security monitor to assign a peripheral (i.e., access to MMIO registers and DMA target addresses) to an enclave. These primitives are designed to be programmed by a trusted-by-all security monitor (unlike our work).

**Time protection.** Ge et al. add time protection to seL4, which closes many of the available side channels in commodity processors [113]. As the paper mentions, some processors do not provide mechanisms needed to close channels. Moreover, channels using busses could not be closed, and they have recently been shown to be effectively exploitable [17]. Our approach of using completely separate hardware for security-critical programs addresses these

concerns for these programs. We do, however, note that our approach (as it stands) does not scale to support all (normal) programs, which may have their own security needs. Therefore, we believe that time protection remains an important abstraction to be explored for when the same processor is asked to host multiple programs.

**Physical isolation & dynamic partitioning.** IRONHIDE introduces dynamic spatial partitioning of processor cores and their communication channels to form isolated enclaves [114]. Non-virtualized composable microprocessor [115] proposes a new server architecture that dynamically partitions CPU cores, memory, and accelerators. In contrast, we statically partition the hardware resources, resulting in a simpler design and a smaller TCB (i.e. no security monitor).

**Other TEE solutions.** Komodo is a verified security monitor that can create enclaves for security-critical programs [116]. Li et al. formally verify the firmware in Realms, part of ARM confidential computing [117]. Use of formal verification warrants the strong trust in the security monitor/firmware, but not the ARM processor that hosts both security-critical and untrusted programs. For example, Li et al. mention that "[p]rotection against known software error injection attacks and side-channel attacks require appropriate usage of architectural mitigations and are beyond the scope of this paper."

Sanctum uses hardware modifications to RISC-V alongside a software security monitor to create isolated enclaves [118]. Compared to SGX, Sanctum enclaves are protected against both cache and page fault side-channel attacks. MI6 time-partitions hardware resources and implements a rigorous "purge" operation that erases microarchitectural and memory states associated with a security-sensitive program [119]. None, however, addresses other potential hardware vulnerabilities such as interconnect side channels.

Sanctuary leverages the Address Space Controller hardware to enable strong isolation in TrustZone's normal world [120]. Sanctuary still requires a security monitor to program the controller.

## 11 DISCUSSIONS

**Scalability and Performance.** The exclusive use of TEE domains limits the number of concurrent security-critical programs. Moreover, our choice of using weak microcontrollers, small amounts of memory, and I/O without DMA for TEEs limit the performance of security-critical programs. We believe that the former is not a serious issue since we do not expect a large number of security-critical programs executing simultaneously in a smartphone.

The latter is mostly a non-issue either since security-critical programs are more concerned with security guarantees than performance. However, there are exceptions, for example, authentication of the user by applying machine learning algorithms to photos taken by the camera or privacy-preserving federated learning [121]. We believe that these programs can leverage accelerators (which will be available in the machine in the form of additional I/O domains). Indeed, Nider et al. propose a machine with no CPU and several self-managed devices [122], showing the diminished role of CPU for performance. We also note that our design allows for using more powerful processors for the TEE domains, albeit at the cost of additional hardware budget.

One might wonder whether we can use a single DMA engine to improve performance of an I/O device for all TEE domains. This is not feasible since domains' memories are physically separated. Instead, we can potentially use multiple domain-bound DMA engines, one for each TEE domain.

**Usability.** We argue that the exclusive use of hardware resources by security-critical programs in our machine does not cause usability problems for normal programs, for three reasons. First, security-critical programs in smartphones already use some I/O devices exclusively. For example, the UI (display and touchscreen) is used exclusively (e.g., when using TrustZone-based Protected Confirmation [123]) due to its small form factor.

Second, the performance impact on other I/O types, such as networking and storage, can be minimal when security-critical programs use short sessions, e.g., a few seconds. In §9.2, we experimentally demonstrate this impact for storage. Moreover, TCP network connection keepalives persist for tens of seconds. Further, since smartphone network connections are frequently dropped during handoffs, most widely used applications transparently re-establish lost connections without user visible changes. Security-critical programs can be designed to initiate, use, and close their connections in a single session (a practice that we use in our own security-critical programs).

It is also possible to mitigate these issues using multiple I/O domains of the same type. For example, all smartphones have both WiFi and cellular network interfaces. One can imagine allowing normal programs to share and use one of these while security-critical programs use the other (through two separate I/O domains in our hardware).

Third, most security-critical programs rely on only a subset of the I/O domains. For example, our insulin pump program (second version) mainly requires access to its sensor and pump as well as a brief access to storage. While this program is running, all other I/O domains, e.g., network, UI, and even storage, can be used by normal programs.

We finally note that any attempt to allow simultaneous sharing of hardware resources will undoubtedly increase the TCB. For example, enabling multiple domains to render to the display simultaneously will require trusting the display domain in our machine.

## 12 CONCLUSIONS

Smartphone owners expect to use their devices for a mixture of security-critical and ordinary tasks, yet this requires trust that the hardware and system software is able to isolate those tasks from each other, trust that is often misplaced. Our goal in this work is to minimize the TCB when executing security-critical programs. We present a hardware design with multiple statically-partitioned, physically-isolated trust domains, coordinated using a few simple, formally-verified hardware components, along with OctopOS, an OS to manage this hardware. We describe a complete prototype implemented on a CPU-FPGA board and show that it incurs a small hardware cost. For security-critical programs, our machine significantly reduces the TCB compared to existing solutions, and achieves usable performance. For normal programs, it achieves similar performance to a legacy machine.

## ACKNOWLEDGMENTS

The work was supported by NSF Awards #1617513, #1718923, #1846230, and #1953932. The authors thank Felix Xiaozhu Lin, anonymous reviewers, and the shepherd for their insightful feedback. They also thank Xilinx for donating an FPGA board to this project.

## APPENDIX

The research artifact accompanying this paper is available via https://doi.org/10.5281/zenodo.7898001

# REFERENCES


[1] A. Phaneuf. State of mobile banking in 2020: top apps, features, statistics and market trends. https://www.businessinsider.com/mobile-banking-market-trends, 2019.

[2] DiabetesMine Team. NEWS: OmniPod Tubeless Insulin Pump to Offer Smartphone Control Soon. https://www.healthline.com/diabetesmine/omnipod-smartphone-control-diabetes, 2019.

[3] R. Nandakumar, S. Gollakota, and N. Watson. Contactless Sleep Apnea Detection on Smartphones. In *Proc. ACM MobiSys*, 2015.

[4] J. Vander Stoep. Android: Protecting the Kernel. In *Linux Security Summit (LSS)*, 2016.

[5] CVE Details. Linux Kernel: Vulnerability Statistics. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html, 2022.

[6] Bugs and Vulnerabilities Founds by Syzkaller in Linux Kernel. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[7] CVE Details. Windows 10: Vulnerability Statistics. https://www.cvedetails.com/product/32238/Microsoft-Windows-10.html, 2021.

[8] CVE Details. XEN: Vulnerability Statistics. https://www.cvedetails.com/product/23463/XEN-XEN.html, 2021.

[9] H. Zhang, D. She, and Z. Qian. Android Root and its Providers: A double-Edged Sword. In *Proc. ACM CCS*, 2015.

[10] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten Years Later. In *Proc. ACM ASPLOS*, 2011.

[11] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *Proc. ACM EuroSys*, 2006.

[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proc. ACM SOSP*, 2001.

[13] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proc. ACM ISCA*, 2014.

[14] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proc. USENIX Security Symposium*, 2018.

[15] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2019.

[16] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, Baris Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proc. USENIX Security Symposium*, 2018.

[17] R. Paccagnella, L. Luo, and C. W. Fletcher. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *Proc. USENIX Security Symposium*, 2021.

[18] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow. Osiris: Automated Discovery of Microarchitectural Side Channels. In *Proc. USENIX Security Symposium*, 2021.

[19] National Vulnerability Database. CVE-2021-0200: Out-of-bounds write in the firmware for Intel(R) Ethernet 700 Series Controllers before version 8.2 may allow a privileged user to potentially enable an escalation of privilege via local access. https://nvd.nist.gov/vuln/detail/CVE-2021-0200.

[20] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proc. USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[21] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[22] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *Proc. USENIX Security Symposium*, 2020.

[23] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX Amplifies the Power of Cache Attacks. In *Proc. Springer International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[24] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *Proc. ACM European Workshop on Systems Security (EuroSec)*, 2017.

[25] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proc. Springer International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[26] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Proc. USENIX Security Symposium*, 2016.

[27] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[28] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng. CipherLeaks: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *Proc. USENIX Security Symposium*, 2021.

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

[30] CVE Details. Op-tee: Vulnerability Statistics. https://www.cvedetails.com/product/56969/Linaro-Op-tee.html, https://www.cvedetails.com/product/42749/Linaro-Op-tee.html, https://www.cvedetails.com/product/36161/Op-tee-Op-tee-Os.html, 2021.

[31] Quarklab. BREAKING SAMSUNG'S ARM TRUSTZONE. https://i.blackhat.com/USA-19/Thursday/us-19-Peterlin-Breaking-Samsungs-ARM-TrustZone.pdf, 2019.

[32] National Vulnerability Database. Vulnerability summary for cve-2015-6639.

[33] F. Hetzelt and R. Buhren. Security Analysis of Encrypted Virtual Machines. In *Proc. ACM VEE*, 2017.

[34] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *Proc. USENIX Security Symposium*, 2019.

[35] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[36] M. Li, Y. Zhang, and Z. Lin. CrossLine: Breaking "security-by-crash" based Memory Isolation in AMD SEV. In *Proc. ACM CCS*, 2021.

[37] Apple Platform Security - Secure Enclave. https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web, 2021.

[38] Apple Platform Security - Touch ID and Face ID security. https://support.apple.com/guide/security/touch-id-and-face-id-security-sec067eb0c9e/web, 2021.

[39] D. Kleidermacher, J. Seed, B. Barbello, S. Somogyi, and Pixel & Tensor security teams Android. Pixel 6: Setting a new standard for mobile security. https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html, 2021.

[40] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*, 2011.

[41] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proc. Summer 1986 USENIX Conference*, 1986.

[42] J. Liedtke. Improving IPC by Kernel Design. *ACM SIGOPS Operating Systems Review*, 1993.

[43] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, 2000.

[44] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. ACM SOSP*, 2009.

[45] K. Elphinstone and G. Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proc. ACM SOSP*, 2013.

[46] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. In *Proc. ACM SOSP*, 1995.

[47] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proc. ACM SOSP*, 1997.

[48] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proc. ACM ASPLOS*, 2011.

[49] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *Proc. USENIX OSDI*, 2014.

[50] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S. Weng, H. Zhang, and Y. Guo. Deep Specifications and Certified Abstraction Layers. In *Proc. ACM POPL*, 2015.

[51] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. USENIX OSDI*, 2016.

[52] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. In *Proc. USENIX Security Symposium*, 2016.

[53] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proc. ACM SOSP*, 2017.

[54] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang. Nickel: A framework for Design and Verification of Information Flow Control Systems. In *Proc. USENIX OSDI*, 2018.

[55] S. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. A Secure and Formally Verified Linux KVM Hypervisor. 2021.

[56] S. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proc. USENIX Security Symposium*, 2021.

[57] R. Tao, J. Yao, X. Li, S. Li, J. Nieh, and R. Gu. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proc. ACM SOSP*, 2021.

[58] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proc. ACM EuroSys*, 2006.

[59] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 2007.

[60] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64 kB Computer Safely and Efficiently. In *Proc. ACM SOSP*,

2017.

[61] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proc. USENIX OSDI*, 2020.

[62] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: a Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. ACM ASPLOS*, 2008.

[63] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proc. ACM ASPLOS*, 2013.

[64] R. Grisenthwaite. Arm CCA will put confidential compute in the hands of every developer. https://www.arm.com/company/news/2021/06/arm-cca-will-put-confidential-compute-in-the-hands-of-every-developer, 2021.

[65] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proc. ACM EuroSys*, 2020.

[66] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *Proc. ACM MobiSys*, 2016.

[67] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *Proc. USENIX Security Symposium*, 2016.

[68] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proc. ACM CCS*, 2016.

[69] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *Proc. USENIX Security Symposium*, 2016.

[70] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2018.

[71] K. Loughlin, S. Saroiu, A. Wolman, and B. Kasikci. Stop! Hammer Time: Rethinking Our Approach to Rowhammer Mitigations. In *Proc. ACM HotOS*, 2021.

[72] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. USENIX Security*, 2014.

[73] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *Proc. ACM MobiSys*, 2019.

[74] Trusted Computing Group (TCG). TPM 2.0 Library. https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2019.

[75] Xilinx. Xilinx Standalone Library Documentation. OS and Libraries Document Collection. UG643 (v2021.1) June 16, 2021.

[76] IBM. Software TPM Introduction. http://ibmswtpm.sourceforge.net/ibmswtpm2.html, 2021.

[77] K. Franz. Add a microSD Slot with the Pmod MicroSD. https://digilent.com/blog/add-a-microsd-slot-with-the-pmod-microsd/, 2021.

[78] YosysHQ GmbH. SymbiYosys (sby) Documentation. https://symbiyosys.readthedocs.io/en/latest/index.html, 2021.

[79] GoatRAT Attacks Automated Payment Systems. https://labs.k7computing.com/index.php/goatrat-attacks-automated-payment-systems, 2023.

[80] Y. Xiao, M. Li, S. Chen, and Y. Zhang. Stacco: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Proc. ACM CCS*, 2017.

[81] AndroidAPS app documentation. http://wiki.aaps.app/en/latest/, 2022.

[82] Arm CoreLink GIC-600 Generic Interrupt Controller Technical Reference Manual. https://developer.arm.com/documentation/100336/0106/operation/security, 2019.

[83] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *Proc. USENIX Security Symposium*, 2007.

[84] E. R. Sparks. A Security Assessment of Trusted Platform Modules. *Dartmouth College Undergraduate Theses. 53*, 2007.

[85] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog. BIOS Chronomancy: Fixing the Core Root of Trust for Measurement. In *Proc. ACM CCS*, 2013.

[86] S. Han, W. Shin, J. Park, and H. Kim. A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping. In *Proc. USENIX Security Symposium*, 2018.

[87] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger. TPM-Fail: TPM meets Timing and Lattice Attacks. In *Proc. USENIX Security Symposium*, 2020.

[88] L. Chen and J. Li. Flexible and Scalable Digital Signatures in TPM 2.0. In *Proc. ACM CCS*, 2013.

[89] J. Shao, Y. Qin, D. Feng, and W. Wang. Formal Analysis of Enhanced Authorization in the TPM 2.0. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*, 2015.

[90] S. Wesemeyer, C. J.P. Newton, H. Treharne, L. Chen, R. Sasse, and J. Whitefield. Formal Analysis and Implementation of a TPM 2.0-based Direct Anonymous Attestation Scheme. In *Proc. ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2020.

[91] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKscrew: Exposing the Perils of Security-Oblivious Energy Management. In *Proc. USENIX Security Symposium*, 2017.

[92] P. Qiu, D. Wang, Y. Lyu, and G. Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *Proc. ACM CCS*, 2019.

[93] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[94] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *Proc. USENIX Security Symposium*, 2022.

[95] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. De Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proc. ACM ASPLOS*, 2015.

[96] S. Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptol. ePrint Arch.*, 2016.

[97] M. H. Yun and L. Zhong. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. In *Proc. Internet Society NDSS*, 2019.

[98] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. Loh, M. Subramony, and S. White. Pioneering Chiplet Technology and Design for the AMD EPYC and Ryzen Processor Families: Industrial Product. In *Proc. ACM/IEEE ISCA*, 2021.

[99] M. Posner. How many ASIC Gates does it take to fill an FPGA? https://blogs.synopsys.com/breakingthethreelaws/2015/02/how-many-asic-gates-does-it-take-to-fill-an-fpga/, 2015.

[100] V. Strumpen. Introduction to Digital Circuits: Basic Digital Circuits. http://bibl.ica.jku.at/dc/build/html/basiccircuits/basiccircuits.html, 2015.

[101] Y. Shizuku, T. Hirose, N. Kuroki, M. Numa, and M. Okada. A 24-transistor static flip-flop consisting of nors and inverters for low-power digital vlsis. In *Proc. IEEE International New Circuits and Systems Conference (NEWCAS)*, 2014.

[102] P. Athe and S Dasgupta. A Comparative Study of 6T, 8T and 9T Decanano SRAM cell. In *Proc. IEEE Symposium on Industrial Electronics & Applications*, 2009.

[103] S. Shankland. https://www.cnet.com/tech/mobile/apples-a15-bionic-chip-powers-iphone-13-with-15-billion-transistors-new-graphics-and-ai/, 2021.

[104] A. Frumusanu. Huawei Announces Mate 40 Series: Powered by 15.3bn Transistors 5nm Kirin 9000. https://www.anandtech.com/show/16156/huawei-announces-mate-40-series, 2020.

[105] Xilinx. Zynq UltraScale + Device. Technical Reference Manual. UG1085 (v2.2) December 4, 2020.

[106] Common AXI Themes on Xilinx's Forum (see Section "Out-of-protocol designs" for the discussion on a bug in Xilinx's Ethernet-lite controller). https://zipcpu.com/blog/2021/03/20/xilinx-forums.html, 2021.

[107] A. Athalye, A. Belay, M.F. Kaashoek, R. Morris, and N. Zeldovich. Notary: A device for secure transaction approval. In *Proc. ACM SOSP*, 2019.

[108] A. Amiri Sani and T. Anderson. The Case for I/O-Device-as-a-Service. In *Proc. ACM HotOS*, 2019.

[109] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. ACM EuroSys*, 2008.

[110] W. Futral and J. Greene. *Intel Trusted Execution Technology for Server Platforms: A Guide to More Secure Datacenters.* Apress Media LLC, Springer Nature, 2013.

[111] S. Weiser and M. Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proc. ACM CODASPY*, 2017.

[112] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A. Sadeghi, and E. Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *Proc. USENIX Security Symposium*, 2021.

[113] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser. Time Protection: The Missing OS Abstraction. In *Proc. ACM EuroSys*, 2019.

[114] H. Omar and O. Khan. IRONHIDE: A Secure Multicore that Efficiently Mitigates Microarchitecture State Attacks for Interactive Applications. In *Proc. IEEE HPCA*, 2020.

[115] Calling for the Return of Non-Virtualized Microprocessor Systems. https://www.sigarch.org/calling-for-the-return-of-non-virtualized-microprocessor-systems/, 2022.

[116] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proc. ACM SOSP*, 2017.

[117] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proc. USENIX OSDI*, 2022.

[118] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proc. USENIX Security Symposium*, 2016.

[119] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proc. ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2019.

[120] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *Proc. Internet Society NDSS*, 2019.

[121] F. Mo, H. Haddadi, K. Katevas, E. Marin, D. Perino, and N. Kourtellis. PPFL: Privacy-Preserving Federated Learning with Trusted Execution Environments. In *Proc. ACM MobiSys*, 2021.

[122] J. Nider and A. Fedorova. The Last CPU. In *Proc. ACM HotOS*, 2021.

[123] Android Protected Confirmation. https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html, 2018.