A Personal Computer for a Distrustful World

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

Personal computer owners often want to be able to run security-critical programs on the same machine as other untrusted and potentially malicious programs. While ostensibly trivial, 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 computer owner needs to trust to withstand adversarial inputs. We present a hardware design, called the split-trust machine model, which is 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 SoCs. For security-critical programs, we show that this machine significantly reduces the required trust compared to mainstream TEEs, and for normal programs, we show that it achieves similar performance as a legacy machine.

1 Introduction

Because of their ubiquity and portability, modern personal computers such as smartphones and laptops are often used to run security-critical programs along with diverse, untrusted, and potentially malicious programs. For example, most of us perform routine financial tasks, such as banking and payments [1, 2], on our personal computers. And many of us run health-related programs, e.g., to receive test results and diagnoses from our health providers, and in some cases, to perform life-critical tasks, such as to control an insulin pump [3] or monitor breathing [4], on these same devices.

Realizing this computing paradigm should be straightforward. We can use an OS (or some other system software such as a hypervisor) to isolate these 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) [5–13] and hardware (e.g., processor, memory, interconnects, and I/O devices including their firmware) [14–20]. Malicious programs that interact with the OS and use the same hardware can exploit these vulnerabilities to take control of the machine and any programs running on it. Therefore, we must trust that the hardware and system software can effectively sandbox and neutralize malicious programs. This trust often proves to be unwarranted.

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 [17, 21–28], attacks exploiting software vulnerabilities [29–32], and attacks based on design flaws [33–35].

In this paper, we present a solution to enable personal computers to be used for both security-critical and non-critical programs. Our goal is to minimize both the number and the complexity of hardware and software components that need to be strongly trusted by the owner of the computer in order to execute a security-critical program. As we will define in §2.1, we say that a component is strongly trusted if it needs to be able to withstand and neutralize 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. Due to exclusive use, a component only needs to be weakly trusted. That is, it only needs to operate correctly in the absence of adversarial inputs.

More concretely, we present a hardware design for a personal computer. Called a split-trust machine model, 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 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.

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 OS.

We rigorously evaluate the required trust, i.e., the Trusted Computing Base (TCB), of this machine. We show that our machine 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 a CPU-FPGA board (Xilinx Zynq UltraScale+ MPSoC ZCU102). We use the powerful ARM Cortex A53 CPU to host the commodity 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, including the TEEs. This choice as well as the small number of TEE domains is based on our observation that security-critical programs, unlike regular programs, are often not as computationally intensive, and the number of such programs that run simultaneously is typically small. In other respects, however, they are like normal programs: they start and stop, run in the background, do I/O, and so forth.

Using our prototype, we build two important security-critical programs for our machine: (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 (emulated) glucose monitor and pump.

Finally, using our prototype, we show that the added hardware cost is small (i.e., 1-2%) compared to modern SoCs. Moreover, we show that normal programs can achieve the same performance as on a legacy machine.

# 2 Background

## 2.1 Trust Definitions

The hardware and software components that need to be trusted for a program to execute securely form its TCB. In our work, we find that it is not adequate to determine whether a component is trusted. We need to determine the type of trust.

More specifically, we define two types of trust: strong trust and weak trust. We say a component is strongly trusted if it needs to guard itself against adversarial inputs. For example, imagine an OS that is trusted to isolate a program from other malicious programs. The malicious programs can issue adversarial syscalls to the OS concurrently to the protected program. In such a case, the program owner needs to trust that the component (e.g., the OS) prevent these other programs from exploiting any vulnerabilities (logical or implementation-related). Ensuring that a software or hardware component is not exploitable is very challenging, as demonstrated by the plethora of reported exploits. Therefore, we believe that strong trust should be minimized for security-critical programs. We note, however, that there are methods for hardening hardware and software components, such as formal verification. Strong trust is acceptable if a component is known to be adequately hardened against vulnerabilities.

We say that a component is weakly trusted if it just needs to operate correctly in the absence of adversarial inputs. For example, consider the same OS mentioned above, but assume that the security-critical program is the only one running on top of the OS (and assume no networking with the outside world). In such a case, the program owner only needs to trust that: (1) the component (e.g., the OS) does not exert buggy behavior under normal usage, i.e., when processing well-formed inputs, and (2) it is not compromised by an adversary before use and upon distribution (e.g., through implanted backdoors). These trust assumptions can be (more) easily met in practice by ensuring that: (1) component designers test it adequately under various expected usage models, (2) the source code of the component is available for inspection by security experts and users, and (3) users can verify the component before use through remote attestation. Therefore, we believe that weak trust is acceptable for security-critical programs.

## 2.2 Trust in Existing Systems

Historically, the OS has been a strongly-trusted part of the system. That is, the OS is trusted to isolate a program from other programs, benign or malicious, and provide three important security guarantees: integrity, confidentiality, and availability. Figure 1 (a) illustrates this traditional design.

As commodity OSes have become more complex over the years, more and more bugs and vulnerabilities have been found in them, allowing malware to exploit them and com-

---

1We will open source our hardware design, OctopOS, our security-critical programs, and our formal verification proofs.
promise the OS [5–8, 10–13, 36]. As an example, there have been about 1400 vulnerabilities reported in the Linux kernel just since 2016. This trend means that strong trust in the OS is no longer warranted.

There have been several attempts to build trustworthy OSes. These include microkernels [37–41], exokernels and library OSes [42–45], formally verified OSes (and hypervisors) [40, 46–53], and OSes written in safe languages [54–57]. While effective, these 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 strong 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 even if the OS is untrusted, but leaves the OS in charge of resource management (and hence the availability guarantee). 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) [58, 59] or using a combination of hardware and software. ARM TrustZone and Intel SGX are famous examples of the latter. Other examples include AMD Secure Encrypted Virtualization (SEV), Intel Trusted Domain Extensions (TDX), Apple’s Secure Enclave Processor (SEP), ARMv9’s Realms [60], and Keystone for RISC-V [61].

Despite their success, existing TEE solutions still require many components to be strongly trusted 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 [9, 62]. The TEE OS in TrustZone also contains vulnerabilities and has been exploited in the past [29–32]. AMD SEV has also been shown to contain several vulnerabilities due to design flaws, all of which have been exploited [33–35].

Hardware components have been exploited as well. Processor-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 [17, 21–28]. The core reason behind this is that existing solutions execute the untrusted OS and TEEs on the same hardware, forcing them to share the underlying microarchitectural features such as caches [21, 24–28] and speculative execution engine [15–17, 22], as well as architectural ones such as virtual memory [23]. The memory subsystem has also proved vulnerable and fallen to Rowhammer attacks [14, 63–67]. The complexity of these hardware components ensures that many more such vulnerabilities will be discovered and exploited. For example, researchers have recently demonstrated a suite of new side channels using the interconnects [18], the x87 floating-point unit, and Advanced Vector extensions (AVX) instructions (among others) [19].

3 Key Goal and Principle

Key goal. Our goal in this work is to minimize the (1) number and (2) complexity of strongly-trusted components. The rationale for (1) is that it is difficult for hardware or software components to adequately protect themselves against adversarial inputs. The rationale for (2) is that it is easier to ensure that a component can fend off adversarial inputs if it is simple, which allows for 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 only needs to be weakly trusted as it does not need to worry about adversarial inputs 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 hardware design, the split-trust machine model (§4). We then introduce an OS for this machine, called OctopOS (§5).

4 Split-Trust Machine Model

Modern machines leverage hardware with a hierarchical privilege model. That is, the hardware provides multiple privilege levels, each with more privilege than previous ones, with one all-powerful privilege level to “rule them all.” These privilege levels are implemented inside the CPU and use other programmable protection hardware components, such as MMUs. This model results inevitably in several complex, strongly-trusted components such as the processor, protection hardware, and system software, which if compromised, affect all programs.

In this paper, we demonstrate a novel hardware design, the split-trust machine model, in which the hardware is split into

2A reference to Tolkien’s The Lord’s of the Rings.
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 model is that a security-critical program can exclusively take control of and use its own domain and exclusively communicate with other domains, e.g., for I/O and IPC, hence significantly reducing the strongly-trusted components. (Exclusive inter-domain communication is enabled with a novel hardware mailbox abstraction that we will introduce in §4.2.) Figure 2 shows a simplified view of this hardware design. Next, we discuss its key aspects.

4.1 Static Partitioning and Physical Isolation

We follow two important principles in our hardware design: (1) domains must be physically isolated (i.e., share no hardware components), and (2) the isolation boundary between them cannot be programmatically and dynamically modified as there is no trusted-by-all hardware or software component. The latter implies that we cannot rely on programmable protection hardware, such as MMU, IOMMU, and address space controller, to enforce isolation. As a result, our design statically partitions the hardware resources between domains. Each domain owns its own hardware components (physical isolation) and that ownership is decided at hardware fabrication time and cannot be changed later (static partitioning).

More specifically, each trust domain has its own processor and memory. We use a powerful CPU for the untrusted domain, which accommodates a commodity OS and its (untrusted) programs, to achieve high performance. 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.

An 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 its interrupts are routed to 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 primitive for this purpose: verifiably delegatable hardware mailbox (mailbox for short).

At its core, a mailbox is a hardware queue, allowing two domains (i.e., the writer and reader) to communicate through message passing. A mailbox provides one-way communication. For two-way communication, two mailboxes are needed.

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 delegatable one. The fixed end is hard-wired to a specific domain. The delegatable 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. (The design of the fixed writer mailbox is similar.) For example, consider the serial output domain in our prototype. This domain 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 messages communicated 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...
A hardware root of trust is needed during remote attestation. We add a simple hardware, called the
reset guard, for the reset signals, which ensures that as long as the quota on a mailbox
has not expired, the domains on both sides of the mailbox cannot be reset, hence ensuring session availability.

When either the message limit or the time limit expires. (The message limit can be set to infinite, but not the time domain, enforcing a temporal limit on the session length.)

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 personal computers. 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. 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.

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.)

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 (assuming there is adequate energy if battery-powered).

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, called the reset guard, for the reset signals, which ensures that as long as the quota on a mailbox has not expired, the domains on both sides of the mailbox cannot be reset, hence ensuring session availability.

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. In a split-trust machine, we use a Trusted Platform Module (TPM) to realize the root of trust.

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. Hence, 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, further bloating the strongly-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 in separate domains. 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 from the ones found in existing TPM chips in order to provide one Platform Configuration Register (PCR) per domain and securely extend it with the measurement of software loaded in the domain. We do not provide more details here due to space limitations. We do, however, note that modifying the number of PCRs and their access permissions in TPM does not change its fundamental design principles. Indeed, the TPM specification does not specify these parameters, leaving them to implementers.

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:

- The DMA engine is hard-wired to only read/write to the memory of the untrusted domain.
- 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.

When an I/O domain is used by a TEE domain, DMA is not used and data is transferred using mailboxes. But when the untrusted domain uses the I/O domain, data is transferred using domain-bound DMA for performance reasons. 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. As in a legacy machine, the untrusted domain may also use an IOMMU to further restrict DMA targets in order to isolate its own address spaces.
4.6 Domain and Mailbox Reset

As mentioned in §3, a key requirement for exclusive use of a domain is that the domain (and all its mailboxes) are reset to a clean state prior to and after use, in a way verifiable by the security-critical program. 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 the 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.

This verification is rather straightforward for a TEE domain. The bootloader, which is stored in a ROM and is part of the remotely attested root of trust (§7.2), is tasked with fully cleaning all the state information in the domain upon reset. Once the program is loaded in the domain, it takes exclusive control of resources. The only way for the resource manager to take the domain back is to reset it (if allowed by the reset guard), which then triggers the bootloader to clean the state. The verification process is less straightforward for I/O domains. We will discuss that 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 nucleus, i.e., the 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 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 client domain, i.e., the domain that has exclusive access to the mailboxes of the corresponding I/O domain.

As mentioned in §3, our goal is to enable a security-critical program to use hardware and software components, such as an I/O device and its service, exclusively and to ensure that the components are reset to a clean state before and after use. I/O services play a role in achieving this goal for I/O domains, as discussed next.

Non-restricted I/O devices. Let us first consider I/O 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, when a security-critical program asks for access to an I/O device, the manager resets the corresponding I/O domain (which triggers the bootloader on the I/O domain to clean all the state information in it) and then delegates its mailboxes to the TEE domain running the program. The program first uses the mailboxes to verify its exclusive access and the session quota. It then uses the attestation report from TPM to verify the software loaded in the I/O domain. It also verifies that the I/O domain has not been used since it has been reset. The latter requires assistance from the I/O service. That is, upon receiving the very first message and before processing the message (in order to prevent exposure to adversarial inputs), the service further extends the domain PCR register in TPM with a constant. This way, the report from TPM reveals whether the domain has been used or not.

Before the program’s session expires or is yielded, the program also needs to ensure that the I/O domain will be reset again before use by any other domain. We also achieve this with assistance from the I/O service. That is, before yield/expiration, the program calls an API in the I/O service to disable message processing, after which the service becomes unusable until it is reset again.

Restricted devices. Next, let us consider I/O devices that cannot be used freely by a security-critical program during a session and require the resource manager to enforce some restrictions (i.e., fine-grained access control). In our prototype, storage falls in this category because the storage drive contains data of other programs as well. Even if the data are encrypted, they need to be protected if a general availability guarantee is needed (§7.4). 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 before delegating the domain 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 on the disk 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 makes the storage service strongly-trusted (§7.4).

5.2 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 can choose to use this runtime, or can use its own.

We provide several categories of functions in this API. The
first category assists with requesting and verifying access to other domains (§5.1). This category also helps the program manage the remaining quota of mailboxes by calling a callback function upon quota updates. The second category provides high-level abstractions for using I/O services such as socket-based networking and terminal prints. The third category assists with using TPM, e.g., to request a remote attestation report. The fourth category provides support for secure IPC. It enables two TEE domains to exclusively communicate. The last category provides support for developing security-critical programs such as cryptographic primitives.

5.3 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 §5.1.

Finally, the manager implements system-wide, untrusted I/O functionalities. One example is the shell. 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. Another example is a file system for the boot partition in the storage domain. The resource manager domain uses this file system to help the bootloaders on other domains read and load boot images. Note that this file system is untrusted. This is because (1) each domain uses secure boot to check the authenticity of the boot image, and (2) the domains verify the images loaded into other domains using local attestation from TPM.

5.4 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. 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 for Linux, 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 and then communicate to them. For example, we have developed a block device driver for Linux to use the storage domain. Similarly, we have developed drivers for the network, input, and output domains.

6 Prototype

We have built a prototype of the split-trust hardware and its OS, OctopOS, on the Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA board. The SoC on the board has an FPGA as well as an ARM Cortex A53 processor. We use the ARM processor 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). For our TEE runtime, I/O services, and the resource manager, we leverage the Standalone library [72], which is single-threaded, 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) [73] 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). We note that we synthesize two types of mailboxes: control-plane mailboxes and data-plane mailboxes. The former has 4 messages of 64 B each and the latter has 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). Our mailbox interrupts a domain on every send and receive, which the domains can use to process incoming messages in a timely fashion and to ensure that outgoing messages have been successfully queued.

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 domain 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.

When building the split-trust hardware, we faced numerous difficulties resulting from limitations imposed by the FPGA board. One limitation is noteworthy: 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 [74]. This provides physical isolation for the storage domain, but significantly degrades its performance due to Pmod’s limited throughput (§9.2).

We note that requiring an FPGA board to experiment with the 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. The emulator runs each of the domains in a separate process on the host OS. In addition to the four aforementioned I/O domains, the emulator emulates a glucose monitor and an insulin pump using a separate I/O domain, which we use to test one of our security-critical programs (§8).

Overall, we have implemented OctoPOS and our hardware emulator in about 37k lines of C code (including 6k of modified drivers from Xilinx and crypto libraries). We report the LoC for our hardware below.

### 6.1 Verified Hardware Design

The split-trust machine model has only four simple hardware components that are strongly trusted (see §7.4 for the TCB analysis): 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 to generate various mailboxes (i.e., mailboxes with different number of readers/writers) from a template design.

The simplicity of our strongly-trusted hardware components enables us to formally verify them. We use SymbiYosys to perform formal verification [75]. SymbiYosys is a frontend for Yosys-based formal hardware verification flows. We use the SMTBMC engine, which uses $k$-induction to formally verify safety features in hardware. Table 1 shows the list of theorems we prove for the mailbox (we omit the rest due to space limitation). Overall, we developed 3000 lines of SystemVerilog code for our hardware verification.

### 7 TCB & Security Analysis

#### 7.1 TCB Notation

The owner of a machine, when running a security-critical program in it, trusts various hardware and software components: the TCB. We introduce and use a simple, compact notation for TCB, discussed here with an abstract example:

![TCB Notation Example](image)

The key operator is the $\top$ sign, which resembles a T (as in Trust). It helps denote a set of trust assumptions. 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 $s$: and $w$: name the components that are strongly and weakly trusted. A component might appear in multiple locations in the TCB. For succinctness, in some formulas, we tag a repeating component with a number in parenthesis on its first appearance and use the number in other locations.

#### 7.2 Lower Bound of TCB

This can be achieved if the machine is dedicated to executing a security-critical program (and nothing else):

![Lower Bound of TCB](image)

where $C, I, A$ stand for Confidentiality, Integrity, and Availability; $P.HW$ is the protection hardware (e.g., MMU and IOMMU); and RoT stands for Root of Trust.
This shows that the owner at the very least needs to strongly trust the (security-critical) program and the RoT. The strong trust in the program itself is fundamental: the program needs to protect itself against adversarial inputs, e.g., malicious network packets. Note that the program in the TCB includes the runtime used by the program to interact with the hardware.

The strong 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 strong trust in the RoT includes strong 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.

### 7.3 TCB of Existing Systems

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

\[
\begin{array}{c|c|c}
\text{C.I.A} & \text{s: Prog., OS, Proc., Mem., I/O, interconn., P.HW, RoT} \\
\hline
\text{Owner} & \text{w:} \\
\end{array}
\]

This shows that the owner strongly trusts the hardware including the processor, memory, I/O devices, protection hardware, and interconnects. Moreover, the OS is also strongly 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, TrustZone, in Formula 1.

\[\text{SM} \text{ is the security monitor (i.e., the secure world OS and monitor code). We note that TrustZone allows the secure world to take full control of an I/O device, i.e., secure I/O (Sec-I/O). Yet, this device and its driver are exposed to multiple programs in the secure world and hence are strongly trusted. Another noteworthy issue is that the OS is strongly trusted when availability is needed as it is in charge of resource scheduling. As can be seen, quite a few hardware and software components are strongly trusted.} \]

### 7.4 Our TCB

Formula 2 shows the TCB of our machine. \text{As, Ag, SD,} and \text{RM} stand for session availability, general availability, storage domain, and resource manager, respectively. Our system requires strong trust in a few cases that were not part of the lower bound. First, for confidentiality, integrity, and session availability, the owner needs to strongly 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 allows us to formally verify them, making this strong trust acceptable (§2.1). 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 strongly trust the resource manager domain and the storage domain. Note that the storage domain does not need to be strongly trusted for other security guarantees since (1) a program can use cryptographic primitives to protect the confidentiality and integrity of data and (2) it can rely on session guarantees provided by our hardware for session availability. The only way to eliminate the strong trust in the storage domain for general availability is to have separate storage devices for each security-critical program. Unfortunately, this is prohibitively expensive. Formal verification of the storage service and the resource manager (which we have not attempted) would make strong trust in them acceptable.

It is noteworthy that our machine eliminates the need to strongly trust several complex hardware and software components such as the processor, memory, I/O devices, the interconnects (since our machine does not share any busses between trust domains) and system software (security monitor, OS, and device drivers), compared to existing TEEs. Moreover, the hardware component listed as weakly-trusted (processor, memory, I/O, and interconnects) are those of the domains used by the security-critical program. This has important implications: for example, it means that the complex powerful processor of the untrusted domain is not trusted at all (not even weakly). 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.

Finally, note that both our TCB and the lower bound have the bootloader and the program runtime as strongly trusted. We have not formally verified these components.

### 7.5 Security Analysis

**Threat model.** We assume an adversary can run untrusted and security-critical programs in the machine and tries to exploit any software or hardware vulnerabilities. Below, we discuss various such attacks and their implications. Physical attacks and remote network attacks are out of scope, but we discuss their implications.

**Software vulnerability-based exploits.** As shown in our threat model, the only software components that need to be strongly trusted are the security-critical program itself (which includes its runtime), the bootloader (part of the RoT), the storage service and the resource manager for general availability. 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 a session availability 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 strongly-trusted hardware components (i.e., mailbox, arbiter, reset guard, ROM, and TPM) would lead to attacks. The first four of these are formally verified (§6.1) and TPM is a mature and secure technology.

**Timing side-channel attacks.** All strongly-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 [76–80]. As TPM is a mature technology, vulnerabilities get fixed. Indeed, there have been several works that formally verify various aspects of the TPM standard [81–83]. We have not analyzed the timing channel of the runtime we have developed for security-critical programs. We however note that our machine allows such programs to provide their own runtime.

**Power management attacks.** These types of attacks induce faults in the victim program’s execution by manipulating the frequency or voltage of the processor and have been demonstrated against TEEs [84–86]. As TPM is a mature technology, vulnerabilities get fixed. Indeed, there have been several works that formally verify various aspects of the TPM standard [81–83]. We have not analyzed the timing channel of the runtime we have developed for security-critical programs. We however note that our machine allows such programs to provide their own runtime.

**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 [87–89]. 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.

**Out of scope: remote network attacks.** In this work, we are concerned with malicious programs residing on the same device as the security-critical program. Therefore, we do not explicitly protect against remote adversaries trying to exploit vulnerabilities over the network. However, we note that our split-trust machine naturally provides some protection against these attackers. This is because it sandboxes the network device and its device driver in its own domain. Therefore, a successful attack over the network mainly impacts this domain. As a result, programs that do not use the network are protected from these attacks. This is in contrast to a legacy machine in which a single successful exploit over the network may result in a full takeover.

### 8 Sample Security-Critical Programs

We briefly discuss two security-critical programs that we have built for our machine.

**I. Secure banking.** Our secure banking program ensures that only the user can access confidential account information. The program leverages several features of our machine. First, it uses exclusive access to the UI (i.e., shell) to make sure all inputs come from the user (and not malware) and that outputs are only displayed to the user. 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.

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 [90], which our exclusive network access closes against on-device attackers; external network side-channel attacks are still possible. Finally, the program uses remote attestation to enable the bank cloud server to trust the program running on the user’s device.

**II. Secure insulin pump.** Diabetic patients need to monitor the glucose level in their blood and administer insulin accordingly. New glucose monitor and insulin pumps have recently emerged that can perform this task automatically [3].

We build this security-critical program in our OS. This program leverages exclusive access to the glucose monitor and insulin pump, e.g., through the headphone jack or via Bluetooth. This way, the program can securely authenticate...
We measure various performance aspects of our machine. We calculate an estimate for the number of transistors needed with a 100 MHz clock (the Ethernet controller IP uses an external 156.25 MHz clock). We repeat each experiment 5 times and report the average and standard deviation.

### 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 [91].

#### 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 [92], 6 transistors per NAND gate [93], 24 transistors for each flip flop [94], and 6 transistor for each bit of on-chip memory (assuming a conventional 6-transistor SRAM cell [95]). Our calculation shows that our machine requires about 164.7 M additional transistors (161 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. For example, Apple A15 Bionic and HiSilicon Kirin 9000 use 15 B transistors [96, 97]. This means that, if our solution is added to an SoC or implemented as a chiplet, 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 156.25 MHz clock). We repeat each experiment 5 times and report the average and standard deviation.

| Configuration          | Throughput (MB/s) | Latency (µs) |
|------------------------|-------------------|--------------|
| A53-Microblaze         | 7.07±0.00         | 18.2±0.00    |
| Microblaze-Microblaze  | 9.64±0.01         | 15.26±0.05   |

#### Table 2: Cost of additional hardware in our machine.

| FPGA resource | Count | Equivalent transistor count |
|---------------|-------|-----------------------------|
| Look-up table | 63,289 | 2,278,404                   |
| Flip flop     | 57,033 | 1,368,792                   |
| Block RAM     | 26,840,190 (bits) | 161,041,140               |

#### Table 3: Mailbox performance.

#### Table 4: Storage performance.

**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 our data-plane mailbox. For latency, we measure the round trip time to send a 64 B message and receive an acknowledgment over our control-plane mailboxes. 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 FPGA-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 [98]. 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). As mentioned in §6, our prototype uses a Pmod-based microSD card for storage. However, the Pmod connection limits the throughput of our storage service.

To measure the storage performance, we perform 2000 reads/writes of 512 B each. We evaluate three configurations: (1) a best-case performance where the storage domain directly performs the reads/writes (hence mainly stressing the Pmod-based microSD card), and (2) the untrusted domain as well as (3) a TEE domain storage performance (where the untrusted domain or a TEE domain uses the storage service). Table 4 shows the results. They show that our storage performance is mainly limited by the Pmod connection.

**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: (1) a baseline where the A53 processor uses the Ethernet device (using an IP on the FPGA provided by Xilinx) and (2) the untrusted and (3) TEE domains using our network service. 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 the measurements. For all experiments, we connect the board to a PC and measure the numbers reported by the measurement programs on the board. Table 5 shows the results. They
show that our domain-bound DMA is capable of matching the performance of a legacy machine.

**Boot time and breakdown.** We measure the time it takes our system to boot. Due to presence of multiple domains, booting OctopOS from a partition in the storage service is a carefully choreographed dance, requiring steps taken by the 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 48.1±0.08 s to boot all domains excluding the untrusted domain, which takes an additional 87.52±2.1 s to boot. Our boot time is mainly spent on (1) reading image files from the boot partition in our (slow) storage domain, and (2) computing the hash of the executables on weak Microblaze microcontrollers, needed for extending the TPM PCRs.

**Untrusted program performance.** The use of a powerful CPU, a large amount of RAM, and domain-bound DMA means that normal programs in the untrusted domain can achieve the same performance they get on a legacy machine. To demonstrate this, we use the network file system. 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 it takes 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 time it takes for a simple security-critical program to boot (upon a shell command) and acquire exclusive access to both input and output domains (excluding local attestation through TPM, including I/O domain reset). Our measurements show that this process takes 3.74±0.03 s.

### 10 Related Work

**Security by physical isolation.** Using a separate, dedicated processor with its own memory and I/O devices for security-critical tasks is a recent hardware trend in personal computers. Apple has integrated the Secure Enclave Processor into its products [99] since about 2014 and used it to secure the user’s secret data and to control biometric sensors (i.e., Touch ID and Face ID) [100]. Similarly, Google recently announced that Pixel 6 uses the tensor security core to host security-critical tasks such as key management and secure boot [101].

Our work takes the concept of security by physical isolation further by allowing programs (including those that rely on I/O devices) to use dedicated processors by developing a model for how that can be done safely.

Notary [102] 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 (although we have not formally verified our bootloader code that cleans up the state upon reset, but plan to investigate adopting Notary’s deterministic start). 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 [103]. Our approach also uses separate I/O microcontrollers but does not require strong trust in the microcontroller software, by resetting the I/O domain between uses.

**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 [104]. The solution requires the enclave program not only to trust SGX’s firmware and hardware, but also the hypervisor. CURE [105] 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 [106]. 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 [18]. 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.

**Other TEE solutions.** Flicker [107] uses the late launch feature of Intel Trusted Execution Technology (TXT) [108], to exclusively run a program on the processor. The exclusive use of the hardware results in minimizing the strongly-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. Consider our secure insulin pump program (§8), 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.

| Configuration   | Throughput (Mbit/s) | RTT (ms) |
|-----------------|---------------------|----------|
| Baseline        | ±0.17               | ±0.17    |
| Untrusted domain| ±0.17               | ±0.17    |
| TEE domain      | ±0.022±0.001        | ±23.92   |

Table 5: Network performance.
Komodo is a verified security monitor that can create en-
claves for security-critical programs [109]. Use of formal
verification warrants the strong trust in the security monitor,
but not the ARM processor that hosts both security-critical
and untrusted programs. Sanctum uses hardware modific-
tions to RISC-V alongside a software security monitor to cre-
ate isolated enclaves. Compared to SGX, Sanctum enclaves
are protected against both cache and page fault side-channel
attacks. While this is important, Sanctum does not address
other potential hardware vulnerabilities such as side channels
through interconnects.

11 Conclusions

Personal computer owners expect to use their devices for a
mixture of security-critical and ordinary tasks, yet this re-
quires strong 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 an FPGA and show that it incurs a small
hardware cost. For security-critical programs, we show that
our approach significantly reduces the TCB compared to ex-
isting solutions, and achieves one close to the lower bound.
For normal programs, we show that our approach 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 for his invaluable feedback on an earlier draft
of this paper. They also thank Xilinx for donating an FPGA
board to this project.

References

[1] A. Meola. The digital trends disrupting the banking
industry in 2021. https://www.businessinsider.com/banking-industry-trends, 2021.

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

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

[4] R. Nandakumar, S. Gollakota, and N. Watson. Contact-
less Sleep Apnea Detection on Smartphones. In Proc.
ACM MobiSys, 2015.

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

[6] CVE Details. Linux Kernel: Vulnerability Statist-
ics. https://www.cvedetails.com/product/47/
Linux-Linux-Kernel.html, 2021.

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

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

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

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

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

[12] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Licht-
enberg, C. McGarvey, B. Ondruszek, S. K. Rajamani,
and A. Ustuner. Thorough Static Analysis of Device
Drivers. In Proc. ACM EuroSys, 2006.

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

[14] 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.

[15] 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 Ker-
nel Memory from User Space. In Proc. USENIX Secu-
rity Symposium, 2018.

[16] 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 At-
tacks: Exploiting Speculative Execution. In Proc. IEEE
Symposium on Security and Privacy (S&P), 2019.
[17] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, Baris Kasikci, F. Piessens, M. Silverstein, 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

[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] 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.

[37] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. In Proc. Summer 1986 USENIX Conference, 1986.

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

[39] 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.
[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

[54] 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.

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

[56] 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.

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

[58] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmaniam, 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.

[59] 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.

[60] 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.
[63] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giafrida, and H. Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In Proc. USENIX Security Symposium, 2016.

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

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

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

[71] Trusted Computing Group (TCG). TCG PC Client Specific TPM Interface Specification (TIS), Specification Version 1.3. https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientTPMInterfaceSpecification_TIS_1-3_27_03212013.pdf, 2013.

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

[73] IBM. Software TPM Introduction. http://ibmswtspm.sourceforge.net/ibmswtspm2.html, 2021.

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

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

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

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

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

[79] 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.

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

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

[82] 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.

[83] 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.

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

[85] 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.

[86] 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.

[87] 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.
[88] S. Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptol. ePrint Arch.*, 2016.

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

[90] 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.

[91] 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.

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

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

[94] 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 vlsi. In *Proc. IEEE International New Circuits and Systems Conference (NEWCAS)*, 2014.

[95] 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.

[96] S. Shankland. Apple’s A15 Bionic chip powers iPhone 13 with 15 billion transistors, new graphics and AI. https://www.cnet.com/tech/mobile/apples-a15-bionic-chip-powers-iphone-13-with-15-billion-transistors-new-graphics-and-ai/, 2021.

[97] 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.

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

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

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

[101] D. Kleidermacher, J. Seed, B. Barbelo, 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.

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

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

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

[105] 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.

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

[107] 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.

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

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