The Case for a Virtualization-Based Trusted Execution Environment in Mobile Devices

Saeed Mirzamohammadi, Ardalan Amiri Sani
University of California, Irvine
saeed@uci.edu, ardalan@uci.edu

ABSTRACT

ARM processors used in modern mobile devices, such as smartphones and tablets, use TrustZone to implement a trusted execution environment (TEE). In this paper, we argue that virtualization hardware, already available on many ARM processors, should be used for this purpose instead. Virtualization hardware can be used to implement multiple isolated trusted environments, as opposed to a single such environment provided by TrustZone. This can prevent the bloat of the Trusted Computing Base (TCB) of the TEE and support new security services not currently possible, such as sandboxing of untrusted operating system kernel components.

We also address the concerns for the use of virtualization for the aforementioned purpose. Most notably, through extensive experiments, we show that, unlike widespread belief, virtualization overhead is small if the hypervisor is carefully designed to minimize its interpositions into the operating system activity. In addition, we discuss and address the concerns on supported features, backward-compatibility, and hypervisor's TCB size.

Going forward, given that virtualization provides a viable (if not superior) TEE solution, we suggest that ARM TrustZone hardware components should be mostly removed from future ARM SoCs. This can simplify the processor and SoC design and save some die space.

KEYWORDS

Mobile devices, Operating systems, Security, Virtualization

APSys '18, August 27–28, 2018, Jeju Island, Republic of Korea

ACM ISBN 978-1-4503-6006-7/18/08…$15.00
https://doi.org/10.1145/3265723.3265725

ACM Reference Format:
Saeed Mirzamohammadi, Ardalan Amiri Sani. 2018. The Case for a Virtualization-Based Trusted Execution Environment in Mobile Devices. In *APSys '18: 9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3265723.3265725

1 INTRODUCTION

ARM-based mobile devices as such as smartphones and tablets have two important hardware features that can be used to build trusted systems: TrustZone and virtualization hardware. In today's mobile devices, TrustZone is predominantly used to implement a trusted execution environment (TEE), which hosts important security services such as secure payment (e.g. Samsung Pay), Digital Rights Management (DRM), and a cryptographic key store. Virtualization hardware, on the other hand, is mainly intended for supporting virtual machines, and hence is typically left unused in mobile devices.

In this paper, we argue that virtualization hardware should be used to implement the TEE in mobile devices instead of TrustZone. Such a design supports multiple isolated TEEs, in contrast to TrustZone, which only supports one TEE, providing two important benefits. First, security services can be divided to run within separate TEEs, which prevents bloating of the Trusted Computing Base (TCB) of the TEE. Second, novel security services, such as sandboxing of untrusted kernel modules, become feasible.

However, there are some concerns about this proposal, which we address in this paper. The first concern is performance. It is widely believed that running the main operating system on top of a hypervisor incurs significant performance overhead [17]. At the same time, it is believed that TrustZone's overhead on the main operating system performance is negligible. Through extensive experimentation, hypervisor redesign, and a TrustZone design study, we provide evidence against both of these arguments. We show that a commodity hypervisor's overhead is mainly due to its frequent interposing on the operating system activities, a design needed only in a multi-tenant virtualization setup. When used to support TEEs, the hypervisor can be redesigned to minimize these interpositions and hence minimize its performance overhead on the main operating system. We present a "passive hypervisor" design, in which the hypervisor is invoked *only* for explicit communications between the operating system and

a TEE. With this design, we show that the performance of the operating system is very close (within about 0.45% on average and 1.89% maximum) of native execution[1].

Moreover, we show that TrustZone's overhead is not negligible either. Indeed, TrustZone incurs unavoidable cost to memory writes (by either CPU or I/O devices) in order to implement secure memory and I/O. Unfortunately, we cannot measure this overhead since this feature is not configurable on different devices that we have investigated. However, our study of its design shows that this overhead is about 1.5%.

The second concern is supported features. TrustZone supports several security features, such as secure I/O, secure memory, and secure boot, all of which are needed for the security services deployed in the TEE. We show that several such features can already be supported with virtualization hardware. Indeed, virtualization hardware can provide better flexibility for secure memory and secure I/O. Also, for those features not supported (i.e., secure boot and cryptographic keys), we discuss that they can be easily supported by virtualization as well.

The third concern is backward compatibility. There are already many services developed to run within a TrustZone TEE. We discuss how these services can be easily supported in a virtualization TEE by porting existing TEE operating systems to a virtual machine, similar to vTZ [26]. The last concern is the hypervisor's TCB size. While a hypervisor's TCB size is currently larger than that of TrustZone's monitor TCB, we discuss that this TCB can be significantly reduced.

Going forward, given that virtualization provides a viable (if not superior) TEE solution, we argue that TrustZone should be removed from future ARM SoC's. This will simplify the hardware and save some die space on the SoC chip.

2 TEE DESIGN

In this section, we describe TEE designs using TrustZone and virtualization hardware.

Modern mobile devices use ARM TrustZone to implement a TEE. TrustZone is a system-wide hardware isolation feature for ARM SoCs. It separates the system into a secure and a normal world and uses a secure monitor to handle switching between these two worlds. The secure world hosts the TEE, which is a secure operating system with its own user and kernel spaces. Some popular OSes used in the secure world are OP-TEE [7] and Trusty OS [3]. TrustZone partitions the memory and hardware I/O devices between the normal and secure worlds. All accesses from the normal world to the resources allocated for the secure world are denied. The secure world, however, has access to all resources. The TEE is used to host security services, such as payment services. These services can be invoked from the normal world through a regulated call gate, enabled by a world switch instruction called the "Secure Mode Call" (SMC). Figure 1a illustrates this design.

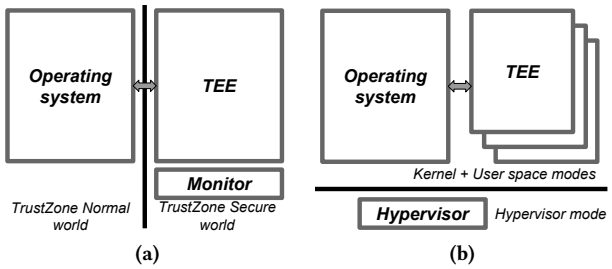

Figure 1: (a) TrustZone-based TEE architecture. (b) Virtualization-based TEE architecture.

In this paper, we make the case for using virtualization hardware to implement the TEE. Virtualization hardware was added to ARM processors in 2011 [35]. However, due to lack of critical use cases, this hardware is typically deactivated in commodity mobile devices. Figure 1b illustrates how this hardware can be used to implement the TEE. The main operating system as well as the TEEs run inside virtual machines on top of a hypervisor. The hypervisor then gives each virtual machine access to the resources it needs. For example, it gives a TEE access to its own secure memory and secure I/O devices, while it gives the main operating system access to unsecure memory and I/O. The operating system and TEEs can also communicate through the hypervisor. The mediation of the hypervisor increases the round trip time for this communication compared to the direct communication with TrustZone. However, we do not anticipate this to be an issue since security services hosted in TEEs are typically not performance-sensitive.

As the figure shows, this design enables having multiple TEEs. This provides two important advantages compared to a single TEE supported by ARM TrustZone. First, it allows for strong isolation between various security services. Recently, many novel security services have been proposed by the research community to be deployed within the TrustZone TEE. Examples are secure sensors [30], AdAttester [28], VButton [29], and TruZ-Droid [45]. However, these services require deploying extra code (e.g., device drivers) in the TEE, mainly in the kernel of the TEE operating system. Such a requirement is an important bottleneck for the adoption of these solutions in practice. This is because doing so will noticeably increase the TCB of the TEE, which hosts critical security services. With the virtualization-based TEE design, these services can be deployed in separate TEEs, without bloating the TCB.

Second, this design allows for sandboxing of the operating system kernel's components. For example, several recent bugs have been found in the Bluetooth [4] and WiFi [5, 44]

[1]The source code for the hypervisor and the benchmarks can be found at `https://trusslab.github.io/hyp_tee/`

subsystems of mobile devices. These bugs can be exploited by malicious parties over the network to mount remote attacks. An effective solution to prevent such exploits is to sandbox these network devices and their corresponding device drivers. Unfortunately, TrustZone TEE cannot be used for this purpose. This is because the TEE houses critical security services and adding these vulnerable devices to this TEE makes it vulnerable to attacks, essentially worsening the security of the system. On the other hand, with a virtualization-based design, isolated virtual machines can be used to sandbox these network devices, similar to Cinch [11]. These virtual machines will be given access only to their own I/O devices, hence properly isolating the vulnerabilities.

While a virtualization-based TEE provides those advantages, it raises some concerns that prohibits its widespread use: performance, supported features, backward-compatibility, and hypervisor's TCB size. We address the concerns on performance and supported features in §3 and §4, respectively. The concern on backward-compatibility is easily addressed by porting existing TrustZone TEE and applications to run in a virtual machine, similar to vTZ [26].

Here, we study the issue of TCB size. As discussed, a virtualization-based TEE allows for reduction in the TCB of the TEE software itself. However, there is concern with the large size of the hypervisor compared to the monitor code in the secure world (Figure 1). Our initial code size measurement shows that Xen hypervisor's TCB for ARM architecture is about 95 kLoC while TrustZone's monitor TCB (i.e., ARM Trusted Firmware) is about 28 kLoC. Our breakdown of Xen's TCB shows that this number can be significantly reduced. For example, about 36 kLoC is for drivers, 20 kLoC is for tools, and 2 kLoC is for cryptography, none of which is needed in our passive hypervisor. This leaves us about 36 kLoC for TCB. Indeed, others have also found that a hypervisor's TCB size can be reduced. For example, in [1], it is mentioned that the ARM code size in Xen can be reduced down to about 11 kLoC. Moreover, [43] and [42] also have shown that the hypervisor code size can be reduced to about 9 and 20 kLoC, respectively. Also, note that ARM Trusted Firmware (in the HiKey development board used in our prototype) can also be reduced to about 17 kLoC by removing its services and tools. This analysis shows that both Xen and ARM Trusted Firmware have comparable TCBs.

Note that while we focus on a commodity hypervisor (Xen) in this paper, we could also use microkernels or microhypervisors such as [2, 24, 27, 43].

3 PERFORMANCE

3.1 Virtualization's Performance

Using a virtualization-based TEE requires the hypervisor in the system all the time (Figure 1b). This, in turn, can affect the performance of the main operating system. In this section, we investigate this issue experimentally. We introduce several design decisions that altogether turn an existing hypervisor into a "passive" hypervisor, which interposes the operating system execution only to handle explicit security calls from it. We show that such a design can achieve performance close to that of native.

We start our investigation with an unmodified Xen hypervisor. We perform performance experiments in the operating system using the popular LMBench suite [34]. Our measurements show that Xen indeed incurs noticeable overhead to several microbenchmarks (up to 123.69%). We then perform detailed instrumentation to find out the sources of the performance overhead and remove them with several design decisions. In this section, we first introduce these design decisions. We then present the results of our measurements.

Design Decision I: No vCPU Scheduling. The hypervisor implements virtual CPUs (vCPU) on top of the physical CPUs (pCPU) in the system. It then assigns a configurable number of vCPUs to each virtual machine and schedules them. This scheduling incurs overhead. This is because the scheduler context-switches the vCPUs over pCPUs. On a context switch between two vCPUs, the scheduler stores the current vCPU's execution state (i.e., pCPU registers) in memory and restores the target vCPU's state. Moreover, to perform the context switch, the virtual machine execution traps into the hypervisor.

We observe that vCPU scheduling is only needed for multi-tenant virtualization environments, where the CPU resources need to be fairly shared between untrusting virtual machines. In a TEE design, transitions between the operating system and TEEs are mainly through explicit calls from the operating system. Therefore, no vCPU scheduling is needed. All the CPUs can be assigned to the main operating system. Each CPU can then switch to execute in a TEE if called by the operating system or to handle an interrupt from a secure I/O device (similar to transitions from the normal world to secure world with TrustZone).

We use two techniques to eliminate the vCPU scheduling. Our first technique is CPU pinning. That is, we pin the vCPUs to pCPUs. To do this, we allocate the same number of vCPUs for the operating system as the number of existing pCPUs in the system and we pin each vCPU to one pCPU. This prevents the hypervisor from performing any context switches.

Note that we will still be able to support explicit domain transitions. That is, if the operating system requires to invoke the TEE, it can issue a hypercall. The passive hypervisor can handle the hypercall by resuming the execution of the calling vCPU in the TEE. Once the request is handled, the vCPU will be programmed to continue its execution in the operating system. No vCPU scheduling will be performed.

Our second technique is to eliminate the use of the idle domain in Xen. When there is no running task, the operating system scheduler switches to the idle task. The idle task runs an idle loop, in which it calls the Wait-for-Interrupt (WFI) instruction. This instruction forces the processor to sleep and wait for an interrupt to wake up. With a commodity hypervisor, this instruction is configured to trap into the hypervisor, which then performs the idling by executing an "idle domain". This allows the hypervisor to make scheduling decisions, if needed. This trap, followed up by the execution of an idle domain, causes performance overhead. Therefore, we remove it in the passive hypervisor. That is, we deprivilege the WFI instruction, which prevents it from trapping. In this case, the operating system performs the idling itself, similar to an operating system running natively.

Design Decision II: Use Super Pages. One source of potential performance overhead is memory virtualization. In an ARM processor with virtualization hardware, there are two stages of address translations (guest virtual to guest physical and guest physical to system physical). A different set of page tables is used in each stage. The operating system controls the first set of page tables (i.e., stage-1 page tables) and the hypervisor controls the second one (i.e., stage-2 page tables). Therefore, the hypervisor can use different page sizes in stage-2 page tables. As we will show in §3.2, using the small page sizes of 4 KB causes noticeable performance overhead, for two reasons. First, for this page size, the page table walk consists of up to four lookups (hence four memory accesses). Second, such a page size results in more translation entries, which increases the Translation Lookaside Buffer (TLB) contention.

Therefore, our experiments show that it is critical to use super pages to implement the physical address space of the operating system. The question becomes: what super page size should the passive hypervisor use? ARMv8 processors support various super page sizes, e.g., 2MB, and 1GB. Requiring the use of 1 GB pages will minimize the performance overhead. However, it will prohibit us from launching more than a few domains, since the granularity of memory partitions is low. Therefore, we use 2 MB pages in the passive hypervisor (fortunately, this is Xen's choice as well). Our experiments show that this super page size results in close-to-native performance.

Design Decision III: No IPI Traps. A commodity hypervisor, such as Xen, interposes on Inter-Processor Interrupts (IPI), causing performance overhead. This interposition is needed in the hypervisor due to CPU virtualization. That is, when the operating system issues an IPI to another vCPU, it traps into the hypervisor, which then injects the IPI to the pCPU corresponding to the target vCPU. However, in a passive hypervisor, the vCPUs are statically mapped to pCPUs and hence this interposition can be eliminated.

We do not yet have this feature in our prototype, but here is how we plan to support it. ARM architecture has a hypervisor configuration register (HCR_EL2) that controls the traps to the hypervisor. HCR_EL2.IMO is the bit for routing physical interrupts to the guest or to the hypervisor. If this bit is unset, all the interrupts are routed to the guest removing the hypervisor from the interrupt path. While this is in principle possible and is our eventual prototype goal, it requires further rehauling of the hypervisor and possibly the guest, which is part of our ongoing efforts. However, in order to demonstrate the benefits of eliminating IPI traps in our benchmarks, we also measure the performance of our benchmarks on a single CPU (since such a configuration eliminates IPIs).

3.2 Virtualization Evaluation

We evaluate the effect of virtualization (both with commodity and passive hypervisors) on the main operating system. We use a HiKey development board, which has a Kirin 620 SoC with an octa-core ARM Cortex-A53 64-bit CPU operating at a maximum frequency of 1.2 GHz [6]. Moreover, the board comes with 2 GB of memory. We build the passive hypervisor on top of Xen 4.9 hypervisor. We use CentOS with Linux kernel version 4.1 for the operating system. We did our measurements using the LMbench micro-benchmarks [34]. We evaluate three configurations: (*i*) native, in which there is no hypervisor and the operating system runs natively on top of hardware, (*ii*) Xen, in which we run the operating system on top of unmodified Xen, and (*iii*) pXen, in which we run the operating system on top of our passive hypervisor.

Tables 1 and 2 show the results. Table 1 shows the results for benchmarks that use a single process. Our results show that for these benchmarks, Xen does not incur noticeable overhead (an average of 0.50%). The passive hypervisor further reduces this overhead to an average of 0.37%. Table 2 shows the results for the benchmarks that use multiple processes. It shows that Xen does incur significant overhead for these benchmarks (due to vCPU scheduling and IPI traps). Moreover, it shows that in a multi-core environment (Table 2, left), the passive hypervisor (pXen) achieves noticeably better performance compared to Xen due to its elimination of vCPU scheduling. For example, in the context switch benchmark, Xen has an overhead of 123% while the passive hypervisor has an overhead of 55%. The remaining overhead is because of IPIs, which happen in a multi-core environment. As mentioned in §3.1, we currently do not support the elimination of IPI traps. Therefore, to demonstrate the effect of this elimination, we also run our benchmarks in a single-core environment (Table 2, right) in order to avoid IPI traps. In these tests, we pinned the benchmarks' processes to a single CPU in the system. Our results show that without

Type	Name	Native	Stdev(%)	Xen	Stdev(%)	Xen Ovr (%)	pXen	Stdev (%)	pXen Ovr (%)
File Sys. BW. (ops/s)	mk	34657.58	0.91	34225.83	0.39	1.24	34298.08	0.07	**1.03**
	rm	51910.00	0.22	51755.25	0.18	0.29	51777.25	0.57	**0.25**
Syscall Lat. (us)	Simple	0.19	0	0.19	0	0	0.19	0	**0**
	rd.	0.60	0	0.60	0	0	0.60	0	**0**
	wr.	0.87	0	0.87	0	0	0.87	0	**0**
	stat	4.00	0	4.00	0	0	4.00	0	**0**
	fstat	0.85	0	0.85	0	0	0.85	0	**0**
	open/close	9.31	0	9.31	0	0	9.31	0	**0**
Select Lat. (us)	fd=250	14.68	0.13	14.76	0	0.54	14.71	0	**0.22**
Signal Lat. (us)	Installation	0.56	0	0.56	0	0	0.56	0	**0**
	Overhead	3.86	0	3.88	0	0.47	3.88	0	**0.47**
	Prot. fault	0.37	0	0.37	0	0	0.37	0	**0**
CPU Lat. (us)	Int64 bit	0.84	0	0.84	0	0	0.84	0	**0**
	Int64 add	0.08	0	0.08	0	0	0.08	0	**0**
	Int64 mul	3.34	0	3.35	0.17	0.29	3.35	0.17	**0.29**
	Int64 div	7.94	0	7.96	0.07	0.25	7.95	0	**0.12**
	Int64 mod	5.85	0	5.86	0	0	5.86	0	**0.17**
File read BW. (MB/s)	io_only	883.48	1.28	877.87	1.21	0.63	881.01	0.92	**0.27**
	Open2close	737.61	0.56	736.00	0.67	0.21	736.00	0.62	**0.21**
Mmap read BW. (MB/s)	io_only	2581.00	0.03	2539.66	0.58	1.60	2554.66	0.05	**1.02**
	Open2close	913.66	0.12	886.00	1.41	3.02	896.33	0.90	**1.89**
Memory Lat. (us)	load	107.73	0.03	109.83	0.63	1.95	109.70	0.72	**1.82**
Memory BW. (MB/s)	Read	2081.00	0.26	2056.00	0.12	1.20	2060.66	0.97	**1.00**
	Write	4665.66	0.01	4647.66	0.02	0.38	4654.33	0.02	**0.24**

Table 1: ***LMBench benchmarks that use one process. "Native", "Xen", and "pXen" columns show averages and the columns on their right show standard deviations. "Ovr" refers to overhead compared to native. Note that the results for memory benchmarks do include the effects of cache and TLB.***

		Multi-core Experiments								Single-core Experiments							
Type	Name	Native	Stdev (%)	Xen	Stdev (%)	Xen Ovr (%)	pXen	Stdev (%)	pXen Ovr (%)	Native	Stdev (%)	Xen	Stdev (%)	Xen Ovr (%)	pXen	Stdev (%)	pXen Ovr (%)
IPC	Pipe Lat. (us)	15.22	4.79	31.85	1.63	109.16	15.34	1.16	0.76	23.11	0.46	23.20	0.46	0.37	23.14	0.42	**0.12**
	Pipe BW. (MB/s)	807.52	0.04	673.72	0.59	16.56	748.12	0.39	7.35	804.18	1.04	801.49	0.42	0.33	803.66	0.63	**0.06**
	Unix stream Lat. (us)	22.20	4.62	41.52	0.98	87.04	41.07	0.14	84.97	31.67	0.56	31.76	0.19	0.28	31.55	0.85	**-0.34**
	Unix stream BW. (MB/s)	1988.14	1.74	1924.82	0.66	3.18	1962.99	0.32	1.26	915.03	0.39	909.32	1.53	0.62	918.00	1.19	**-0.32**
Fork	Exit Lat. (us)	552.96	3.24	599.80	1.09	8.47	571.56	1.27	3.36	494	0.20	501.33	2.37	1.48	501.15	2.34	**1.44**
	Execv Lat. (us)	1538.33	1.76	1639.00	1.62	6.54	1634.16	0.24	6.22	1464.00	0.24	1492.00	0.52	1.91	1490.00	1.27	**1.77**
	/bin/sh Lat. (us)	3303.33	2.50	3438.33	0.23	4.08	3341.16	0.35	1.14	3325.33	0.45	3434	0.48	3.26	3387.83	2.18	**1.87**
Context switch Lat., s=4k, p=2-96 (us)		7.47	0.13	16.71	0.03	123.69	11.58	0.04	55.02	11.43	0.10	11.64	1.72	1.80	11.56	1.40	**1.16**
Netw. Lat. (us)	TCP	76.13	0.10	115.00	0.18	51.05	84.34	0.08	10.78	87.61	0.24	87.92	0.13	0.35	87.88	0.16	**0.31**
	UDP	59.27	0.32	76.77	1.51	29.51	67.71	0.25	14.22	61.52	0.14	61.87	0.27	0.57	61.67	0.13	**0.25**

Table 2: ***LMBench benchmarks that use multiple processes. "Native", "Xen", and "pXen" columns show averages and the columns on their right show standard deviations. "Ovr" refers to overhead compared to native.***

the IPI traps, the passive hypervisor overhead can be reduced significantly (to an average of 0.63% across benchmarks).

Memory latency: We also measured the effect of the page size used in the stage-2 page tables on the memory latency. Our experiments show that the average memory load latency is 109.83 us and 158.13 us with the 2 MB and 4 kB page sizes, respectively. This shows that it is critical to use super pages in the stage-2 page tables, as mentioned in §3.1.

3.3 TrustZone's Performance

One of the components in SoCs with TrustZone support is TrustZone Address Space Controller (TZASC). TZASC performs security checks on every read or write accesses to memory or I/O devices. These checks are performed using filters. Each filter controls memory accesses by a single or multiple sources (i.e., CPU and DMA engines) and it can be set up to control up to 8 separate regions. Each region is a contiguous range of memory addresses. For every transaction, the controller looks at all the regions in the filter. If the transaction address matches the address range of the region, the controller checks whether the access is allowed based on the access type (secure/non-secure read/write access) and the access permissions set on the region.

We study TZC-400, a popular TZASC. Our study shows that the controller has a minimum 2 cycles overhead for each memory write access. This additional overhead impacts

both memory accesses and I/O devices (register accesses as well as Direct Memory Access (DMA)). For example, for memory writes, assuming a 107 ns memory access latency (Table 1) and a max frequency of 1.2 GHz (as in our prototype described in §3.2), the TrustZone overhead is about 1.5%. However, note that with memory virtualization, the overhead is due to address translation, which can be reduced with a good TLB algorithm. However, with TrustZone, the overhead is unavoidable since the security checks are performed on the physical addresses.

Note that memory reads are not mostly affected since the controller supports speculation access [12] (i.e., the transaction is dispatched to memory in parallel to the security checking). However, TZC-400 supports only 32 in-flight speculative transactions; thus, a memory-heavy benchmark might experience overhead for memory reads as well.

4 TEE FEATURES

In this section, we discuss TrustZone TEE features and mention that they are either already supported by virtualization hardware or can be easily supported.

4.1 Secure Memory

TrustZone supports secure memory for the TEE. It does so, as mentioned in §3.3, by performing checks on the physical address requests sent to TZASC. Virtualization hardware can also support isolated secure memory for its TEE domains. It does so using a two-stage address translation, which allows it to isolate the memory pages assigned to different virtual machines.

Indeed, we argue that virtualization's method of implementing secure memory is superior. First, it allows the hypervisor to assign memory to different domains at a 2 MB super page granularity (a smaller page size is possible but it degrades performance as shown in §3.2). On the other hand, TrustZone can only create 8 regions [12] and each region must be allocated contiguously on the physical memory.

Second, with an effective TLB algorithm, virtualization's overhead on memory access can be made to be very small. However, TrustZone always incurs a constant overhead of 2 cycles on memory writes as mentioned in 3.3. Since TrustZone performs the security checks on the physical address, TLB cannot reduce the overhead.

4.2 Secure I/O

TrustZone supports secure I/O for the TEE. That is, it can assign an I/O device to the TEE domain, in which case the I/O device will not be accessible to the normal world operating system. This is the technique behind several research efforts using ARM TrustZone including secure sensors [30], AdAttester [28], VButton [29], and TruZ-Droid [45].

Virtualization can also support secure I/O by assigning an I/O device to a virtual machine (a technique also known as direct device assignment [8, 14, 22, 31, 32]). In this technique, the registers of the device is mapped to the virtual machine's physical address space, the interrupts are redirected to the virtual machine, and the Direct Memory Access (DMA) operations of the I/O devices are limited to the virtual machine memory using I/O Memory Management Units (IOMMUs).

Indeed, we argue that virtualization's method of implementing secure I/O is superior. First, virtualization allows to secure I/O device interface partially. For example, Viola only monitors the operating system accesses to some registers of an I/O device [36, 37]. And SchrodinText only controls the GPU and display's access to the framebuffer using IOMMUs [9]. Implementing such systems with TrustZone's secure I/O requires giving full control of these I/O devices to the TEE, which unnecessarily bloats its TCB.

Second, the performance of I/O devices are also affected by the security checks performed by the TZASC. More specifically, CPU write access to I/O device registers as well as DMA writes are subject to the 2 cycle overhead of TZASC (and reads can suffer from overhead too if the limited number of speculative accesses are not adequate as mentioned in §3.3). Similar to memory access, virtualization also adds overhead to these operations due to address translations. However, virtualization's overhead can be mitigated by a good translation caching algorithm.

4.3 Secure Boot & Cryptographic Keys

TrustZone's secure boot allows it to check the integrity of the operating system image before loading it. Moreover, the cryptographic keys accessible only in the secure world allows for implementation of various cryptographic protocols (including the integrity check used in secure boot) as well as a key store used by applications. These features are not currently supported in virtualization hardware. However, we argue that they can simply be added. First, the cryptographic keys are burned on some form of a read-only memory (e.g., "One Time Programmable (OTP) or eFuse memory" [13]) only available to the secure world (the EL3 privilege level). To make them available to the hypervisor, they should be simply be made accessible to the EL2 privilege level. We believe that hardware modification needed to achieve this is trivial.

Second, secure boot is implemented in multiple stages in software (e.g., in the BIOS, bootloader, and the secure world code) [40]. Therefore, adding this feature to the hypervisor only requires modifications to these software layers.

5 GOING FORWARD

5.1 Our Future Work

Motivated by our preliminary results presented in this paper, we plan to build a complete prototype of a virtualization-based TEE. More specifically, we plan to do the following.

First, we plan to add IPI trap routing to the guest in our current prototype, as discussed in §3.1. Second, we plan to support several TEEs, one running existing TrustZone TEE software (to support existing security services), a few to sandbox network devices (e.g., bluetooth and USB), and a few to isolate some of the TEE's security services for TCB reduction. Third, while we have currently managed to reduce the overhead of the passive hypervisor noticeably for our benchmarks, we plan to further investigate the reason behind the remaining (even though small) overheads and eliminate them, if possible.

5.2 Hardware Proposal

Based on the analysis provided in this paper, we posit that future ARM SoCs should remove most of ARM TrustZone components (except for secure boot and cryptographic keys, which we propose to be managed by the hypervisor). This has the benefit of hardware simplification. TrustZone is an SoC-wide solution. It relies on several hardware components including previously discussed TZASC for protecting DRAM, TrustZone Memory Adapter (TZMA) for protecting SRAM, TrustZone Protection Controller (TZPC) for configuring peripherals as secure or non-secure, read-only memory used for cryptographic keys, and (secure) Random Number Generator (RNG) for cryptographic use cases. It also relies on CPU extensions for supporting another privilege level, Generic Interrupt Controller (GIC) for protecting the interrupts of secure I/O devices, and an extra signal on Advanced Microcontroller Bus Architecture (AMBA) for indicating whether a memory access is from the secure or normal world. Removing TrustZone allows for the removal of all but two (the read-only memory used for keys and the secure random number generator) of these components and also for the simplification of the CPU, GIC, and the AMBA bus, which simplifies the SoC design, providing die space for other features to be implemented.

6 RELATED WORK

There have been several research attempts on using a hypervisor for security purposes. vTZ [26] uses the hypervisor to virtualize the functionality of TrustZone for multiple guest virtual machines. In contrast, we focus on having a low-overhead passive hypervisor to replace TrustZone altogether. Cho et al. [17] implement a TEE using a hypervisor. Yet, to mitigate the performance overhead, they deactive it when the TEE is not needed. They also use TrustZone to protect the memory of this TEE and the hypervisor when they are deactivated. In contrast, we only use the hypervisor to implement a TEE and do not depend on TrustZone and we mitigate the performance concerns through hypervisor optimizations. BitVisor [42] uses a small hypervisor (in terms of code size) to provide I/O device security for a single virtual machine. Viola [36, 37] provides sensor notifications by monitoring the I/O devices using the hypervisor. Ditio [38] provides auditing of sensor activities. SchrodinText [9] securely displays sensitive textual contexts. SecVisor [41] and XNPro [39] protect the kernel memory from code injection attacks using the hypervisor for x86 and ARM systems. Overshadow [15] and InkTag [25] protect applications from an untrusted operating system. In contrast, we use the hypervisor to provide a TEE for mobile devices. Cox et al. in [18] and Heiser in [23] argue for the use of hypervisors for security services in mobile and embedded devices. This is aligned with our vision as we believe that virtualization hardware and hypervisors are well-suited for this task.

Some existing work evaluates the performance of virtualization and provides novel designs. Dall et al. [19] evaluates the performance of ARM virtualization with a focus on ARM servers. Cherkasova et al. [16] measure the performance of I/O activities in Xen. Gehrmann et al. [20, 21] similarly compare the use of virtualization and ARM TrustZone on mobile phones, but do not provide performance measurement on a real hardware. OKL4 microvisor [24] is a hypervisor that has the flexibility of a microkernel. It is built based on multiple isolated components and supports multiple virtual machines. NOVA [43] also has a similar idea but for x86 systems and minimizes the code base by moving the virtualization support code to the user level. Cells [10] virtualizes Android at the operating system level to provide multiple Android phones with seperate phone numbers without using ARM virtualization hardware. LightVM [33] modifies Xen and its tools to have performance comparable to containers for certain operations such as virtual machine boot time and migration. However, we focus on improving the performance of the system when virtualization is used to provide a TEE.

7 CONCLUSIONS

We presented our investigation on the benefits of using virtualization hardware to implement a TEE in ARM-based devices compared to using TrustZone. We also addressed the concerns for such a design. Given the viability (if not superiority) of the virtualization-based TEE, we suggested TrustZone to be removed from future ARM SoCs.

ACKNOWLEDGMENTS

This work was supported in part by NSF Award #1617513. The authors thank the anonymous APSys reviewers.

REFERENCES

[1] 2013. Xen on ARM. https://www.slideshare.net/xen_com_mgr/alsf13-stabellini.

[2] 2016. Fiasco.OC microkernel. https://os.inf.tu-dresden.de/fiasco/.

[3] 2017. Trusty TEE. https://source.android.com/security/trusty/.

[4] 2018. Bluetooth Vulnerability (BlueBorne). https://www.armis.com/blueborne/.

[5] 2018. Broadcom Wi-Fi bug. https://nvd.nist.gov/vuln/detail/CVE-2017-9417.

[6] 2018. HiKey (LeMaker version) Specification. http://www.lemaker.org/product-hikey-specification.html.

[7] 2018. Open Portable Trusted Execution Environment (OP-TEE). https://www.op-tee.org/.

[8] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. 2006. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal* (2006).

[9] A. Amiri Sani. 2017. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proc. ACM MobiSys*.

[10] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. 2011. Cells: A Virtual Mobile Smartphone Architecture. In *Proc. ACM SOSP*.

[11] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. 2016. Defending against Malicious Peripherals with Cinch. In *Proc. USENIX Security Symposium*.

[12] ARM. 2013, 2014. ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual, Revision: r0p1. *ARM DDI* 0504C (ID063014) (2013, 2014).

[13] ARM. 2014. Juno ARM Development Platform SoC, Revision r0p0, Technical Overview. https://static.docs.arm.com/dto0038/a/DTO0038A.pdf. *ARM DTO* 0038A (ID040516) (2014).

[14] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. A. Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. USENIX OSDI*.

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

[16] L. Cherkasova and R. Gardner. 2005. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proc. USENIX ATC*.

[17] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. 2016. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *Proc. USENIX ATC*.

[18] L. P. Cox and P. M. Chen. 2007. Pocket Hypervisors: Opportunities and Challenges. In *Proc. IEEE/ACM HotMobile*.

[19] C. Dall, S. Li, J. T. Lim, J. Nieh, and G. Koloventzos. 2016. ARM virtualization: performance and architectural implications. In *Proc. ACM ISCA*.

[20] H. Douglas. 2010. *Thin Hypervisor-Based Security Architectures for Embedded Platforms*. Ph.D. Dissertation. Royal Institute of Technology.

[21] C. Gehrmann, H. Douglas, and D. K. Nilsson. 2011. Are there good Reasons for Protecting Mobile Phones with Hypervisors?. In *Proc. IEEE Consumer Communications and Networking Conference (CCNC)*.

[22] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, and A. Schuster. 2012. ELI: Bare-Metal Performance for I/O Virtualization. In *Proc. ACM ASPLOS*.

[23] G. Heiser. 2009. Hypervisors for Consumer Electronics. In *Proc. IEEE Consumer Communications and Networking Conference (CCNC)*.

[24] G. Heiser and B. Leslie. 2010. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*.

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

[26] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. 2017. vTZ: Virtualizing ARM TrustZone. In *Proc. USENIX Security Symposium*.

[27] 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. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. ACM SOSP*.

[28] W. Li, H. Li, H. Chen, and Y. Xia. 2015. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proc. ACM MobiSys*.

[29] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. 2018. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proc. ACM MobiSys*.

[30] H. Liu, S. Saroiu, A. Wolman, and H. Raj. 2012. Software Abstractions for Trusted Sensors. In *Proc. ACM MobiSys*.

[31] J. Liu, W. Huang, B. Abali, and D. K. Panda. 2006. High Performance VMM-Bypass I/O in Virtual Machines. In *Proc. USENIX ATC*.

[32] M. Liu, T. Li, N. Jia, A. Currid, and V. Troy. 2015. Understanding the Virtualization "Tax" of Scale-out Pass-Through GPUs in GaaS Clouds: An Empirical Study. In *Proc. IEEE HPCA*.

[33] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proc. ACM SOSP*.

[34] L. W. McVoy and C. Staelin. 1996. lmbench: Portable tools for performance analysis. In *Proc. USENIX ATC*.

[35] Roberto Mijat and Andy Nightingale. 2011. Virtualization is Coming to a Platform Near You. ARM White Paper.

[36] S. Mirzamohammadi and A. Amiri Sani. 2016. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. In *Proc. ACM MobiSys*.

[37] S. Mirzamohammadi and A. Amiri Sani. 2018. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. *IEEE Transactions on Mobile Computing (TMC)* (2018).

[38] S. Mirzamohammadi, J. A. Chen, A. Amiri Sani, S. Mehrotra, and G. Tsudik. 2017. Ditio: Trustworthy Auditing of Sensor Activities in Mobile & IoT Devices. In *Proc. ACM SenSys*.

[39] J. Nordholz, J. Vetter, M. Peter, M. Junker-Petschick, and J. Danisevskis. 2015. XNPro: Low-Impact Hypervisor-Based Execution Prevention on ARM. In *Proc. ACM International Workshop on Trustworthy Embedded Devices (TrustED)*.

[40] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. 2017. BootStomp: On the Security of Bootloaders in Mobile Devices. In *Proc. USENIX Security Symposium*.

[41] A. Seshadri, M. Luk, N. Qu, and A. Perrig. 2007. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. *ACM SIGOPS Operating Systems Review* (2007).

[42] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. 2009. Bitvisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proc. ACM VEE*.

[43] U. Steinberg and B. Kauer. 2010. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proc. ACM EuroSys*.

[44] M. Vanhoef and F. Piessens. 2017. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proc. ACM CCS*.

[45] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du. 2018. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *Proc. ACM MobiSys*.