fTPM: A Software-only Implementation of a TPM Chip

Himanshu Raj,[*] *Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox,*
Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon,
Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten
Microsoft

Abstract: *Commodity CPU architectures, such as ARM and Intel CPUs, have started to offer trusted computing features in their CPUs aimed at displacing dedicated trusted hardware. Unfortunately, these CPU architectures raise serious challenges to building trusted systems because they omit providing secure resources outside the CPU perimeter.*

This paper shows how to overcome these challenges to build software systems with security guarantees similar to those of dedicated trusted hardware. We present the design and implementation of a firmware-based TPM 2.0 (fTPM) leveraging ARM TrustZone. Our fTPM is the reference implementation of a TPM 2.0 used in millions of mobile devices. We also describe a set of mechanisms needed for the fTPM that can be useful for building more sophisticated trusted applications beyond just a TPM.

1 Introduction

In recent years, commodity CPU architectures have started to offer built-in features for trusted computing. TrustZone on ARM [1] and Software Guard Extensions (SGX) [25] on Intel CPUs offer runtime environments strongly isolated from the rest of the platform's software, including the OS, applications, and firmware. With these features, CPU manufacturers can offer platforms with a set of security guarantees similar to those provided via dedicated security hardware, such as secure coprocessors, smartcards, or hardware security tokens.

Unfortunately, the nature of these features raises serious challenges for building secure software with guarantees that match those of dedicated trusted hardware. While runtime isolation is important, these features omit many other secure resources present in dedicated trusted hardware, such as storage, secure counters, clocks, and entropy. These omissions raise an important question: *Can we overcome the limitations of commodity CPU security features to build software systems with security guarantees similar to those of trusted hardware?*

In this work, we answer this question by implementing a software-only Trusted Platform Module (TPM) using ARM TrustZone. We demonstrate that the low-level primitives offered by ARM TrustZone and Intel SGX can be used to build systems with high-level trusted computing semantics. Second, we show that these CPU security features can displace the need for dedicated trusted hardware. Third, we demonstrate that these CPU features can offer backward compatibility, a property often very useful in practice. Google and Microsoft already offer operating systems that leverage commodity TPMs. Building a backwards compatible TPM in software means that no changes are needed to Google and Microsoft operating systems. Finally, we describe a set of mechanisms needed for our software-only TPM that can also be useful for building more sophisticated trusted applications beyond just a TPM.

This paper presents firmware-TPM (fTPM), an end-to-end implementation of a TPM using ARM TrustZone. fTPM provides security guarantees similar, although not identical, to a discrete TPM chip. Our implementation is the reference implementation used in all ARM-based mobile devices running Windows including Microsoft Surface and Windows Phone, comprising millions of mobile devices. fTPM was the first hardware or software implementation to support the newly released TPM 2.0 specification. The fTPM has much better performance than TPM chips and is fully backwards compatible: *no modifications are required to the OS services or applications between a mobile device equipped with a TPM chip and one equipped with an fTPM; all modifications are limited only to firmware and drivers.*

To address the above question, this paper starts with an analysis of ARM TrustZone's security guarantees. We thoroughly examine the shortcomings of the ARM TrustZone technology needed for building secure services, whether for fTPM or others. We also examine Intel's

[*]Currently with ContainerX.

SGX and show that many of TrustZone's shortcomings remain present.

We present three approaches to overcome the limitations of ARM TrustZone: (1) provisioning additional trusted hardware, (2) making design compromises that do not affect TPM's security and (3) slightly changing the semantics of a small number of TPM 2.0 commands to adapt them to TrustZone's limitations. Based on these approaches, our implementation uses a variety of mechanisms, such as *cooperative checkpointing*, *fate sharing*, and *atomic updates*, that help the fTPM overcome the limitations of commodity CPU security features. This paper demonstrates that these limitations can be overcome or compensated for when building a software-only implementation of a dedicated trusted hardware component, such as a TPM chip. The fTPM has been deployed in millions of mobile devices running legacy operating systems and applications originally designed for discrete TPM chips.

Finally, this paper omits some low-level details of our implementation and a more extensive set of performance results. These can be found in the fTPM technical report [44].

2 Trusted Platform Module: An Overview

Although TPMs are more than a decade old, we are seeing a resurgence of interest in TPMs from both industry and the research community. TPMs have had a mixed history, in part due to the initial perception that the primary use for TPMs would be to enable digital rights management (DRM). TPMs were seen as a mechanism to force users to give up control of their own machines to corporations. Another factor was the spotty security record of some the early TPM specifications: TPM version 1.1 [52] was shown to be vulnerable to an unsophisticated attack, known as the *PIN reset* attack [49]. Over time, however, TPMs have been able to overcome their mixed reputation, and are now a mainstream component available in many commodity desktops and laptops.

TPMs provide a small set of primitives that can offer a high degree of security assurance. First, TPMs offer strong machine identities. A TPM can be equipped with a unique RSA key pair whose private key never leaves the physical perimeter of a TPM chip. Such a key can effectively act as a globally unique, unforgeable machine identity. Additionally, TPMs can prevent undesired (i.e., malicious) software rollbacks, can offer isolated and secure storage of credentials on behalf of applications or users, and can attest the identity of the software running on the machine. Both industry and the research community have used these primitives as building blocks in a variety of secure systems. This section presents several such systems.

2.1 TPM-based Secure Systems in Industry

Microsoft. Modern versions of the Windows OS use TPMs to offer features, such as BitLocker disk encryption, virtual smart cards, early launch anti-malware (ELAM), and key and device health attestations.

BitLocker [37] is a full-disk encryption system that uses the TPM to protect the encryption keys. Because the decryption keys are locked by the TPM, an attacker cannot read the data just by removing a hard disk and installing it in another computer. During the startup process, the TPM releases the decryption keys only after comparing a hash of OS configuration values with a snapshot taken earlier. This verifies the integrity of the Windows OS startup process. BitLocker has been offered since 2007 when it was made available in Windows Vista.

Virtual smart cards [38] use the TPM to emulate the functionality of physical smart cards, rather than requiring the use of a separate physical smart card and reader. Virtual smart cards are created in the TPM and offer similar properties to physical smart cards – their keys are not exportable from the TPM, and the cryptography is isolated from the rest of the system.

ELAM [35] enables Windows to launch anti-malware before any third-party drivers or applications. The anti-malware software can be first- or third-party (e.g., Microsoft Windows Defender or Symantec Endpoint Protection). Finally, Windows also uses the TPM to construct attestations of cryptographic keys and device boot parameters [36]. Enterprise IT managers use these attestations to assess the health of devices they manage. A common use is to gate access to high-value network resources based on its attestations.

Google. Modern versions of Chrome OS [22] use TPMs for a variety of tasks, including software and firmware rollback prevention, protecting user data encryption keys, and attesting the mode of a device.

Automatic updates enable a remote party (e.g., Google) to update the firmware or the OS of devices that run Chrome OS. Such devices are vulnerable to "remote rollback attacks", where a remote attacker replaces newer software, through a difficult-to-exploit vulnerability, with older software, with a well-known and easy-to-exploit vulnerability. Chrome devices use the TPM to prevent software updates to versions older than the current one.

eCryptfs [14] is a disk encryption system used by Chrome OS to protect user data. Chrome OS uses the TPM to rate limit password guessing on the file system encryption key. Any attempt to guess the AES keys requires the use of a TPM, a single-threaded device that is relatively slow. This prevents parallelized attacks and limits the effectiveness of password brute-force attacks.

Chrome devices can be booted into one of four different modes, corresponding to the state of the developer switch and the recovery switch at power on. These switches may be physically present on the device, or they may be virtual, in which case they are triggered by certain key presses at power on. Chrome OS uses the TPM to attest the device's current mode to any software running on the machine, a feature used for reporting policy compliance.

More details on the additional ways in which Chrome devices make use of TPMs are described in [22].

2.2 TPM-Based Secure Systems in Research

The research community has proposed many uses for TPMs in recent years.

• **Secure VMs for the cloud:** Software stacks in typical multi-tenant clouds are large and complex, and thus prone to compromise or abuse from adversaries including the cloud operators, which may lead to leakage of security-sensitive data. CloudVisor [58] and Credo [43] are virtualization-approaches that protect the privacy and integrity of customer's VMs on commodity cloud infrastructure, even when the virtual machine monitor (VMM) or the management VM becomes compromised. These systems require TPMs to attest to cloud customers the secure configuration of the hosts running their VMs.

• **Secure applications, OSs and hypervisors:** Flicker [33], TrustVisor [32], Memoir [41] leverage the TPM to provide various (but limited) forms of runtimes with strong code and data integrity and confidentiality. Code running in these runtimes is protected from the rest of the OS. These systems have small TCBs because they exclude the bulk of the OS.

• **Novel secure functionality:** Pasture [30] is a secure messaging and logging library that provides secure offline data access. Pasture leverages the TPM to provide two safety properties: access-undeniability (a user cannot deny any offline data access obtained by his device without failing an audit) and verifiable-revocation (a user who generates a verifiable proof of revocation of unaccessed data can never access that data in the future). These two properties are essential to an offline video rental service or to an offline logging and revocation service.

Policy-sealed data [47] relies on TPMs to provide a new abstraction for cloud services that lets data be sealed (i.e., encrypted to a customer-defined policy) and then unsealed (i.e., decrypted) only by hosts whose configurations match the policy.

cTPM [9] extends the TPM functionality across several devices as long as they are owned by the same user. cTPM thus offers strong user identities (across all of her devices), and cross-device isolated secure storage.

Finally, mobile devices can leverage a TPM to offer new trusted services [19, 31, 28]. One example is *trusted sensors* whose readings have a high degree of authenticity and integrity. Trusted sensors enable new mobile apps relevant to scenarios in which sensor readings are very valuable, such as finance (e.g., cash transfers and deposits) and health (e.g., gather health data) [48, 56]. Another example is enforcing driver distraction regulations for in-car music or navigation systems [28].

2.3 TPM 2.0: A New TPM Specification

The Trusted Computing Group (TCG) recently defined the specification for TPM version 2.0 [54]. This newer TPM is needed for two key reasons. First, the crypto algorithms in TPM 1.2 [55] have become inadequate. For example, TPM 1.2 only offers SHA-1 and not SHA-2; SHA-1 is now considered weak and cryptographers are reluctant to use it. Another example is the introduction of ECC with TPM 2.0.

The second reason is the lack of an universally-accepted reference implementation of the TPM 1.2 specification. As a result, different TPM 1.2 implementations exhibit slightly different behaviors. The lack of a reference implementation also keeps the TPM 1.2 specification ambiguous. It can be difficult to specify the exact behavior of cryptographic protocols in English. Instead, with TPM 2.0 the specification is the same as the reference implementation. The specification consists of several documents describing the behavior of the codebase, and these documents are derived directly from the TPM 2.0 codebase, thereby ensuring uniform behavior.

Recently, TPM manufacturers have started to release discrete chips implementing TPM 2.0. Also, at least one manufacturer has released a firmware upgrade that can update a TPM 1.2 chip into one that implements both TPM 2.0 and TPM 1.2. Note that although TPM 2.0 subsumes the functionality of TPM 1.2, it is not backwards compatible. A BIOS built to use a TPM 1.2 would not work with a TPM 2.0-only chip. A list of differences between the two versions is provided by the TCG [53].

3 Modern Trusted Computing Hardware

Recognizing the increasing demand for security, modern CPUs have started to incorporate trusted computing features, such as ARM TrustZone [1] and Intel Software Guard Extensions (SGX) [25]. This section presents the background on ARM TrustZone (including its shortcomings); this background is important to the design of fTPM. Later, Section 12 will describe Intel's SGX and its shortcomings.

3.1 ARM TrustZone

ARM TrustZone is ARM's hardware support for trusted computing. It is a set of security extensions found in many recent ARM processors (including Cortex A8, Cortex A9, and Cortex A15). ARM TrustZone provides two virtual processors backed by hardware access control. The software stack can switch between the two states, referred to as "worlds". One world is called *secure world* (SW), and the other *normal world* (NW). Each world acts as a runtime environment with its own resources (e.g., memory, processor, cache, controllers, interrupts). Depending on the specifics of an individual ARM SoC, a single resource can be strongly partitioned between the two worlds, can be shared across worlds, or assigned to a single world only. For example, most ARM SoCs offer memory curtaining, where a region of memory can be dedicated to the secure world. Similarly, processor, caches, and controllers are often shared across worlds. Finally, I/O devices can be mapped to only one world, although on certain SoCs this mapping can be dynamically controlled by a trusted peripheral.

• **Secure monitor:** The secure monitor is an ARM processor mode that enables context switching between the secure and normal worlds. A special register determines whether the processor core runs code in the secure or non-secure worlds. When the core runs in monitor mode the processor is considered secure regardless of the value of this register.

An ARM CPU has separate banks of registers for each of the two worlds. Each of the worlds can only access their separate register files; cross-world register access is blocked. However, the secure monitor can access non-secure banked copies of registers. The monitor can thus implement context switches between the two worlds.

• **Secure world entry/exit:** By design, an ARM platform always boots into the secure world first. Here, the system firmware can provision the runtime environment of the secure world before any untrusted code (e.g., the OS) has a chance to run. For example, the firmware allocates secure memory for TrustZone, programs the DMA controllers to be TrustZone-aware, and initializes any secure code. The secure code eventually yields to the normal world where untrusted code can start executing.

The normal world uses a special ARM instruction called *smc* (secure monitor call) to transfer control into the secure world. When the CPU executes the *smc* instruction, the hardware switches into the secure monitor, which performs a secure context switch into the secure world. Hardware interrupts can trap directly into the secure monitor code, which enables flexible routing of those interrupts to either world. This allows I/O devices to map their interrupts to the secure world if desired.

• **Curtained memory:** At boot time, the software running in the secure monitor can allocate a range of physical addresses to the secure world only, creating the abstraction of curtained memory – memory inaccessible to the rest of the system. For this, ARM adds an extra control signal for each of the read and write channels on the main memory bus. This signal corresponds to an extra bit (a 33rd-bit on a 32-bit architecture) called the *non-secure* bit (NS-bit). These bits are interpreted whenever a memory access occurs. If the NS-bit is set, an access to memory allocated to the secure world fails.

3.2 Shortcomings of ARM TrustZone

Although the ARM TrustZone specification describes how the processor and memory subsystem are protected in the secure world and provides mechanisms for securing I/O devices, the specification is silent on how many other resources should be protected. This has led to fragmentation – SoCs offer various forms of protecting different hardware resources for TrustZone, or no protection at all. While there may be major differences between the ARM SoCs offered by different vendors, the observations below held across all the major SoCs vendors when products based on this work shipped.

• **No Trusted Storage:** Surprisingly, the ARM TrustZone specification offers no guidelines on how to implement secure storage for TrustZone. The lack of secure storage drastically reduces the effectiveness of TrustZone as trusted computing hardware.

Naively, one might think that code in TrustZone could encrypt its persistent state and store it on untrusted storage. However, encryption alone is not sufficient because (1) one needs a way to store the encryption keys securely, and (2) encryption cannot prevent rollback attacks.

• **Lack of Secure Entropy and Persistent Counters:** Most trusted systems make use of cryptography. However, the TrustZone specification is silent on offering a secure entropy source or a monotonically increasing persistent counter. As a result, most SoCs lack an entropy pool that can only be read from the secure world, and a counter that can persist across reboots and cannot be incremented by the normal world.

• **Lack of virtualization:** Sharing the processor across two different worlds in a stable manner can be done using virtualization techniques. Although ARM offers virtualization extensions [2], the ARM TrustZone specification does not mandate them. As a result, many ARM-based SoCs used in mobile devices today lack virtualization support. Virtualizing commodity operating systems on an ARM platform lacking hardware-assistance for virtualization is challenging.

• **Lack of secure clock and other peripherals:** Secure systems often require a secure clock. While TrustZone can protect memory, interrupts, and certain system buses on the SoC, this protection does extend to the ARM peripheral bus. It is hard to reason about the security guarantees of a peripheral if its controller can be programmed by the normal world, even when its interrupts and memory region are mapped into the secure world. Malicious code could program the peripheral in a way that could make it insecure. For example, some peripherals could be put in "debug mode" to generate arbitrary readings that do not correspond to the ground truth.

• **Lack of access:** Most SoC hardware vendors do not provide access to their firmware. As a result, many developers and researchers are unable to find ways to deploy their systems or prototypes to TrustZone. In our experience, this has seriously impeded the adoption of TrustZone as a trusted computing mechanism.

SoC vendors are reluctant to give access to their firmware. They argue that their platforms should be "locked down" to reduce the likelihood of "hard-to-remove" rootkits. Informally, SoC vendors also perceive firmware access as a threat to their competitiveness. They often incorporate proprietary algorithms and code into their firmware that takes advantage of the vendor-specific features offered by the SoC. Opening firmware to third parties could expose more details about these features to their competitors.

4 High-Level Architecture

Leveraging ARM TrustZone, we implement a trusted execution environment (TEE) that acts as a basic operating system for the secure world. Figure 1 illustrates our architecture, and our system's trusted computing base (TCB) is shown in the shaded boxes.

At a high-level, the TEE consists of a monitor, a dispatcher, and a runtime where one or more trusted services (such as the fTPM) can run one at a time. The TEE exposes a single trusted service interface to the normal world using shared memory. Our system's TCB comprises the ARM SoC hardware, the TEE layers, and the fTPM service.

By leveraging the isolation properties of ARM TrustZone, the TEE provides *shielded execution*, a term coined by previous work [5]. With shielded execution, the TEE offers two security guarantees:

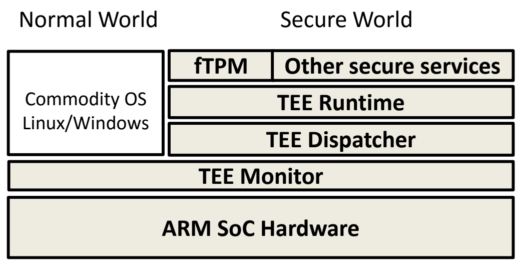

Figure 1: **The architecture of the fTPM.** This diagram is not to scale.

• *Confidentiality:* The whole execution of the fTPM (including its secrets and execution state) is hidden from the rest of the system. Only the fTPM's inputs and outputs, but no intermediate states, are observable.

• *Integrity*: The operating system cannot affect the behavior of the fTPM, except by choosing to refuse execution or to prevent access to system's resources (DoS attacks). The fTPM's commands are always executed correctly according to the TPM 2.0 specification.

4.1 Threat Model and Assumptions

A primary assumption is that the commodity OS running in the Normal World is untrusted and potentially compromised. This OS could mount various attacks to code running in TrustZone, such as making invalid calls to TrustZone (or setting invalid parameters), not responding to requests coming from TrustZone, or responding incorrectly. In handling these attacks, it is important to distinguish between two cases: (1) not handling or answering TrustZone's requests, or (2) acting maliciously.

The first class of attacks corresponds to *refusing service*, a form of Denial-of-Service attacks. DoS attacks are out of scope according to the TPM 2.0 specification. These attacks cannot be prevented as long as an untrusted commodity OS has access to platform resources, such as storage or network. For example, a compromised OS could mount various DoS attacks, such as erasing all storage, resetting the network card, or refusing to call the *smc* instruction. Although our fTPM will remain secure (e.g., preserves confidentiality and integrity of its data) in the face of these attacks, the malicious OS could starve the fTPM leaving it inaccessible.

However, the fTPM must behave correctly when the untrusted OS makes incorrect requests, returns unusual values (or fails to return at all), corrupts data stored on stable storage, injects spurious exceptions, or sets the platform clock to an arbitrary value.

At the hardware level, we assume that the ARM SoC (including ARM TrustZone) itself is implemented correctly, and is not compromised. An attacker cannot mount hardware attacks to inspect the contents of the ARM SoC, nor the contents of RAM memory on the platform. However, the adversary has full control beyond the physical boundaries of the processor and memory. They may read the flash storage and arbitrarily alter I/O including network traffic or any sensors found on the mobile device. In other work, we address the issue of physical attacks on the memory of a mobile device [10].

We defend against side-channel attacks that can be mounted by malicious software. Cache collision attacks are prevented because all caches are flushed when the processor context switches to and from the Secure World. Our fTPM implementation's cryptography library uses constant time cryptography and several other timing attack preventions, such as RSA blinding [27]. However, we do not defend against power analysis or other side-channel attacks that require physical access to hardware or hardware modifications.

We turn our focus on the approaches taken to overcome TrustZone's shortcomings in the fTPM.

5 Overcoming TrustZone Shortcomings

We used three approaches to overcome the shortcomings of ARM TrustZone's technology.

• **Approach #1: Hardware Requirements.** Providing secure storage to TEE was a serious concern. One option was to store the TEE's secure state in the cloud. We dismissed this alternative as not viable because of its drastic impact on device usability. TPMs are used to measure the boot software (including the firmware) on a device. A mobile device would then require cloud connectivity at boot time in order to download the fTPM's state and start measuring the boot software.

Instead, we imposed additional hardware requirements on device manufacturers to ensure a minimum level of hardware support for the fTPM. Many mobile devices already come equipped with an embedded Multi-Media Controller (eMMC) storage controller that has an (off-SoC) replay-protected memory block (RPMB). The RPMB's presence, combined with encryption, ensures that TEE can offer storage that meets the needs of all the fTPM's security properties. Thus, our first hardware requirement for TEE is an eMMC controller with support for RPMB.

Second, we require the presence of hardware fuses accessible only from the secure world. A hardware fuse provides write-once storage. At provisioning time (before being released to a retail store), manufacturers provision our mobile devices by setting the secure hardware fuses with a secure key unique per device. We also require an entropy source accessible from the secure world. The TEE uses both the secure key and the entropy source to generate cryptographic keys at boot time.

Section 6 provides in-depth details of these three hardware requirements.

• **Approach #2: Design Compromises.** Another big concern was long-running TEE commands. Running inside TrustZone for a long time could jeopardize the stability of the commodity OS. Generally, sharing the processor across two different worlds in a stable manner should be done using virtualization techniques. Unfortunately, many of the targeted ARM platforms lack virtualization support. Speaking to the hardware vendors, we learned that it is unlikely virtualization will be added to their platforms any time soon.

Instead, we compromised and require that no TEE code path can execute for long periods of time. This translates into an fTPM requirement – no TPM 2.0 command can be long running. Our measurements of TPM commands revealed that only one TPM 2.0 command is long running: generating RSA keys. Section 7 presents the compromise made in the fTPM design when an RSA key generation command is issued.

• **Approach #3: Modifying the TPM 2.0 Semantics.** Lastly, we do not require the presence of a secure clock from the hardware vendors. Instead, the platform only has a secure timer that ticks at a pre-determined rate. We thus determined that the fTPM cannot offer any TPM commands that require a clock for their security. Fortunately, we discovered that some (but not all) TPM commands can still be offered by relying on a secure timer albeit with slightly altered semantics. Section 8 will describe all these changes in more depth.

6 Hardware Requirements

6.1 eMMC with RPMB

eMMC stands for embedded Multi-Media Controller, and refers to a package consisting of both flash memory and a flash memory controller integrated on the same silicon die [11]. eMMC consists of the MMC (multimedia card) interface, the flash memory, and the flash memory controller. Later versions of the eMMC standard offer a replay-protected memory block (RPMB) partition. As the name suggests, RPMB is a mechanism for storing data in an authenticated and replay-protected manner.

RPMB's replay protection utilizes three mechanisms: an authentication key, a write counter, and a nonce.

RPMB Authentication Key: A 32-byte one-time programmable authentication key register. Once written, this register cannot be over-written, erased, or even read. The eMMC controller uses this authentication key to compute HMACs (SHA-256) to protect data integrity.

Programming the RPMB authentication key is done by issuing a specially formatted dataframe. Next, a result read request dataframe must be also issued to check that the programming step succeeded. Access to the RPMB is prevented unless the authentication key has been programmed. Any write/read requests will return a special error code indicating that the authentication key has yet to be programmed.

RPMB Write Counter: The RPMB partition also maintains a counter for the number of authenticated write requests made to RPMB. This is a 32-bit counter initially set to 0. Once it reaches its maximum value, the counter will no longer be incremented and a special bit will be turned on in all dataframes to indicate that the write counter has expired. The correct counter value must be included in each dataframe written to the controller.

Nonce: RPMB allows a caller to label its read requests with 16-byte nonces that are reflected in the read responses. These nonces ensure that reads are fresh.

6.1.1 Protection against replay attacks

To protect writes from replay attacks, each write includes a write counter value whose integrity is protected by an authentication key (the RPMB authentication key), a shared secret provisioned into both the secure world and the eMMC controller. The read request dataframe that verifies a write operation returns the incremented counter value, whose integrity is protected by the RPMB authentication key. This ensures that the write request has been successful.

The role of nonces in read operations protects them against replay attacks. To ensure freshness, whenever a read operation is issued, the request includes a nonce and the read response includes the nonce signed with RPMB authentication key.

6.2 Secure World Hardware Fuses

We required a set of hardware fuses that can be read from the secure world only. These fuses are provisioned with a hard-to-guess, unique-per-device number. This number is used as a seed in deriving additional secret keys used by the fTPM. Section 9 will describe in-depth how the seed is used in deriving secret fTPM keys, such as the secure storage key (SSK).

6.3 Secure Entropy Source

The TPM specification requires a true random number generator (RNG). A true RNG is constructed by having an entropy pool whose entropy is supplied by a hardware oscillator. The secure world must manage this pool because the TEE must read from it periodically.

Generating entropy is often done via some physical process (e.g., a noise generator). Furthermore, an entropy generator has a *rate of entropy* that specifies how many bits of entropy are generated per second. When the platform is first started, it can take some time until it has gathered "enough" bits of entropy for a seed.

We require the platform manufacturer to provision an entropy source that has two properties: (1) it can be managed by the secure world, and (2) its specification lists a conservative bound on its rate of entropy; this bound is provided as a configuration variable to the fTPM. Upon a platform start, the fTPM waits to initialize until sufficient bits of entropy are generated. For example, the fTPM would need to wait at least 25 seconds to initialize if it requires 500 bits of true entropy bits from a source whose rate is 20 bits/second.

Alerted to this issue, the TPM 2.0 specification has added the ability to save and restore any accumulated but unused entropy across reboots. This can help the fTPM reduce the wait time for accumulating entropy.

7 Design Compromises

7.1 Background on Creating RSA Keys

Creating an RSA key is a resource-intensive operation for two reasons. First, it requires searching for two large prime numbers, and such a search is theoretically unbounded. Although many optimizations exist on how to search RSA keys efficiently [40], searching for keys is still a lengthy operation. Second, the search must be seeded with a random number, otherwise an attacker could attempt to guess the primes the search produced. Thus the TPM cannot create an RSA key unless the entropy source has produced enough entropy to seed the search.

The TPM can be initialized with a *primary* storage root key (SRK). The SRK's private portion never leaves the TPM and is used in many TPM commands (such as TPM *seal* and *unseal*). Upon TPM initialization, our fTPM waits to accumulate the entropy required to seed the search for large prime numbers. The fTPM also creates RSA keys upon receiving a *create RSA keys* command[1].

TPM 2.0 checks whether a number is prime using the Miller-Rabin probabilistic primality test [40]. If the test fails, the candidate number is not a prime. However, upon passing, the test offers a probabilistic guarantee – the candidate is likely a prime with high probability. The TPM repeats this test a couple of times to increase

[1]This corresponds to the TPM 2.0 TPM2_Create command.

the likelihood the candidate is prime. Choosing a composite number during RSA key creation has catastrophic security consequences because it allows an attacker to recover secrets protected by that key. TPM 2.0 repeats the primality test five times for RSA-1024 keys and four times for all RSA versions with longer keys. This reduces the likelihood of choosing a false prime to a probability lower than 2^{-100}.

7.2 Cooperative Checkpointing

Our fTPM targets several different ARM platforms (from smartphones to tablets) that lack virtualization support, and the minimal OS in our TEE lacks a preemptive scheduler. Therefore, we impose a requirement on services running in the TEE that the transitions to TEE and back must be short to ensure that the commodity OS remains stable. Unfortunately, creating an RSA key is a very long process, often taking in excess of 10 seconds on our early hardware tablets.

Faced with this challenge, we added *cooperative checkpointing* to the fTPM. Whenever a TPM command takes too long, the fTPM checkpoints its state in memory, and returns a special error code to the commodity OS running in the Normal World.

Once the OS resumes running in the Normal World, the OS is free to call back the TPM command and instruct the fTPM to resume its execution. These "resume" commands continue processing until the command completes or the next checkpoint occurs. Additionally, the fTPM also allows all commands to be cancelled. The commodity OS can cancel any TPM command even when in the command is in a checkpointed state. Cooperative checkpointing lets us bypass the lack of virtualization support in ARM, yet continue to offer long-running TPM commands, such as creating RSA keys.

8 Modifying TPM 2.0 Semantics

8.1 Secure Clock

TPMs use secure clocks for two reasons. The first use is to measure lockout durations. Lockouts are time periods when the TPM refuses service. Lockout are very important to authorizations (e.g., checking a password). If a password is incorrectly entered more than *k* times (for a small *k*), the TPM enters lockout and refuses service for a pre-determined period of time. This thwarts dictionary attacks – guessing a password incorrectly more than *k* times puts the TPM in lockout mode.

The second use of a secure clock in TPMs is for time-bound authorizations, such as the issuing an authorization valid for a pre-specified period of time. For example, the TPM can create a key valid for an hour only. At the end of an hour, the key becomes unusable.

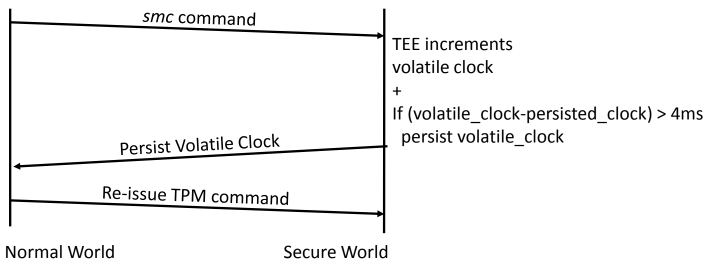

Figure 2: **fTPM clock update.**

8.1.1 Requirements of the TPM 2.0 Specification

A TPM 2.0 requirement is the presence of a clock with millisecond granularity. The TPM uses this clock only to measure intervals of time for time-bound authorizations and lockouts. The volatile clock value must be persisted periodically to a specially-designated non-volatile entry called *NVClock*. The periodicity of the persistence is a TPM configuration variable and cannot be longer than 2^{22} milliseconds (˜70 minutes).

The combination of these properties ensures that the TPM clock offers the following two guarantees: 1. *the clock advances while the TPM is powered*, 2. *the clock never rolls backwards more than NVClock update periodicity.* The only time when the clock can roll backward is when the TPM loses power right before persisting the NVClock value. Upon restoring power, the clock will be restored from NVClock and thus rolled back. The TPM also provides a flag that indicates the clock may have been rolled back. This flag is cleared when the TPM can guarantee the current clock value could not have been rolled back.

Given these guarantees, the TPM can measure time only while the platform is powered up. For example, the TPM can measure one hour of time as long as the platform does not reboot or shutdown. However, the clock can advance slower than wall clock *but only due to a reboot*. Even in this case time-bound authorizations are secure because they do not survive reboots by construction: in TPM 2.0, a platform reboot automatically expires all time-bound authorizations.

8.1.2 Fate Sharing

The main difficulty in building a secure clock in the fTPM is that persisting the clock to storage requires the cooperation of the (untrusted) OS. The OS could refuse to perform any writes that would update the clock. This would make it possible to roll back the clock arbitrarily just by simply rebooting the platform.

The fate sharing model suggests that it is acceptable to lose the clock semantics of the TPM as long as the TPM itself becomes unusable. Armed with this principle, we designed the fTPM's secure clock to be the first piece of functionality the fTPM executes on *all* commands. The fTPM refuses to provide any functionality until the clock is persisted. Figure 2 illustrates how the fTPM updates its clock when the TEE is scheduled to run.

The fTPM implementation does not guarantee that the clock cannot be rolled back arbitrarily. For example, an OS can always refuse to persist the fTPM's clock for a long time, and then reboot the platform effectively rolling back the clock. However, fate sharing guarantees that the fTPM services commands *if and only if* the clock behaves according to the TPM specification.

8.2 Dark Periods

The diversity of mobile device manufacturers raised an additional challenge for implementing the fTPM. A mobile device boot cycle starts by running firmware developed by one (of the many) hardware manufacturers, and then boots a commodity OS. The fTPM must provide functionality throughout the entire boot cycle. In particular, both Chrome and Windows devices issue TPM *Unseal* commands after the firmware finishes running, but before the OS starts booting. These commands attempt to unseal the decryption keys required for decrypting the OS loader. At this point, the fTPM cannot rely on external secure storage because the firmware has unloaded its storage drivers while the OS has yet to load its own. We refer to this point as a "dark period".

TPM Unseal uses storage to record a failed unseal attempt. After a small number of failed attempts, the TPM enters lockout and refuses service for a period of time. This mechanism rate-limits the number of attempts to guess the unseal authorization (e.g., Windows lets users enter a PIN number to unseal the OS loader using BitLocker). The TPM maintains a counter of failed attempts and requires persisting it each time the counter increments. This eliminates the possibility of an attacker brute-forcing the unseal authorization and rebooting the platform without persisting the counter. Figures 3, 4, and 5 illustrate three timelines: a TPM storing its failed attempts counter to stable storage, a TPM without stable storage being attacked with by a simple reboot, and the fTPM solution to dark periods based on the dirty bit.

8.2.1 Modifying the Semantics of Failed Tries

We address the lack of storage during a dark period by making a slight change in how the TPM 2.0 interprets the failed tries counter. At platform boot time, before entering any possible dark periods, the fTPM persists

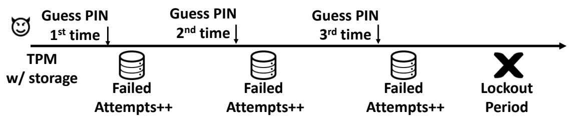

Figure 3: **TPM with storage (no dark period).** TPM enters lockout if adversary makes too many guess attempts. This sequence of steps is secure.

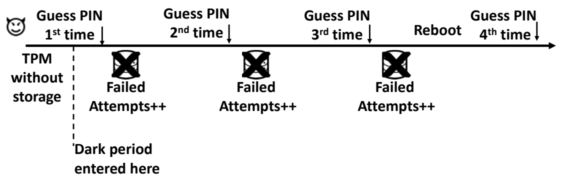

Figure 4: **TPM during a dark period (no stable storage).** Without storing the failed attempts counter, the adversary can simply reboot and avoid TPM lockout. This sequence of steps is insecure.

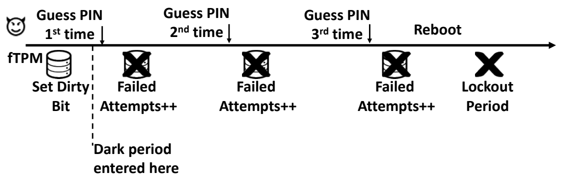

Figure 5: **fTPM during a dark period (no stable storage).** fTPM sets the dirty bit before entering a dark period. If reboot occurs during the dark period, fTPM enters lockout automatically. This sequence of steps is secure.

a *dirty bit*. If for any reason the fTPM is unable to persist the dirty bit, it refuses to offer service. If the dark period is entered and the unseal succeeds, the OS will start booting successfully and load its storage drivers. Once storage becomes available again, the dirty bit is cleared. However, the dirty bit remains uncleared should the mobile device reboot during a dark period. In this case, when the fTPM initializes and sees that the bit is dirty, the fTPM cannot distinguish between a legitimate device reboot (during a dark period) and an attack attempting to rollback the failed tries counter. Conservatively, the fTPM assumes it is under attack, the counter is immediately incremented to the maximum number of failed attempts, and the TPM enters lockout.

This change in semantics guarantees that an attack against the counter remains ineffective. The trade-off is that a legitimate device reboot during a dark period puts the TPM in lockout. The TPM cannot unseal until the lockout duration expires (typically several minutes).

Alerted to this problem, the TPM 2.0 designers have added a form of the *dirty bit* to their specification, called the *non-orderly* or *unorderly bit* (both terms appear in the specification). Unfortunately, they did not adopt the idea of having a small number of tries before the TPM enters lockout mode. Instead, the specification dictates that the TPM enters lockout as soon as a failed unsealed attempt cannot be recorded to storage. Such a solution impacts usability because it locks the TPM as soon as the user has entered an incorrect PIN or password.

9 Providing Storage to Secure Services

The combination of encryption, the RPMB, and hardware fuses is sufficient to build trusted storage for the TEE. Upon booting the first time, TEE generates a symmetric RPMB key and programs it into the RPMB controller. The RPMB key is derived from existing keys available on the platform. In particular, we construct a secure storage key (SSK) that is unique to the device and derived as following:

$$SSK := KDF(HF, DK, UUID) \tag{1}$$

where KDF is a one-way key derivation function, HF is the value read from the hardware fuses, DK is a device key available to both secure and normal worlds, and UUID is the device's unique identifier.

The SSK is used for authenticated reads and writes of all TEE's persistent state (including the fTPM's state) to the device's flash memory. Before being persisted, the state is encrypted with a key available to TrustZone only. Encryption ensures that all fTPM's state remains confidential and integrity protected. The RPMB's authenticated reads and writes ensure that fTPM's state is also resilient against replay attacks.

9.1 Atomic Updates

TEE implements atomic updates to the RPMB partition. Atomic updates are necessary for fTPM commands that require multiple separate write operations. If these writes are not executed atomically, TEE's persistent state could become inconsistent upon a failure that leaves the secure world unable to read its state.

The persisted state of the fTPM consists of a sequence of blocks. TEE stores two copies of each block: one representing the committed version of the state block and one its shadow (or uncommitted) version. Each block id X has a corresponding block whose id is $X+N$, where N is the size of fTPM's state. The TEE also stores a bit vector in its first RPMB block. Each bit in this vector indicates which block is committed: if the i bit is 0 then the ith block committed id is X, otherwise is $X+N$. In this way, all pending writes to shadow blocks are committed using a single atomic write operation of the bit vector.

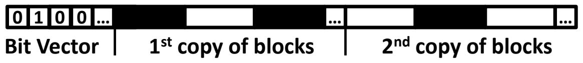

Figure 6: **RMPB blocks.** Bit vector mechanism used for atomic updates.

Allocating the first RPMB entry to the bit vector limits the size of the RPMB partition to 256KB (the current eMMC specification limits the size of a block to 256 bytes). If that size is insufficient, an extra layer of indirection can extend the bit vector mechanism to support up to 512MB ($256*8*256*8 = 1,048,576$ blocks).

Figure 6 illustrates the bit vector mechanism for atomic updates. On the left, the bit vector shows which block is committed (bit value 0) and which block is shadow (bit value 1). The committed blocks are shown in solid color.

In the future, we plan to improve the fTPM's performance by offering transactions to fTPM commands. All writes in a transaction are cached in memory and persisted only upon commit. The commit operation first updates the shadow version of changed blocks, and then updates the metadata in a single atomic operation to make shadow version for updated blocks the committed version. A command that updates secure state must either call commit or abort before returning. Abort is called implicitly if commit fails, where shadow copy is rolled back to the last committed version, and an error code is returned. In this scenario, the command must implement rollback of any in-memory data structure by itself.

10 Performance Evaluation

This paper answers two important questions on performance[2]:

1. What is the overhead of long-running fTPM commands such as *create RSA keys*? The goal is to shed light on the fTPM implementation's performance when seeking prime numbers for RSA keys.

2. What is the performance overhead of typical fTPM commands, and how does it compare to the performance of a discrete TPM chip? TPM chips have notoriously slow microcontrollers [33]. In contrast, fTPM commands execute on full-fledged ARM cores.

10.1 Methodology

To answer these questions, we instrumented four off-the-shelf commodity mobile devices equipped with fTPMs and three machines equipped with discrete TPMs. We keep these devices' identities confidential, and refer to

[2]The fTPM technical report presents additional results of the performance evaluation [44].

fTPM Device	Processor Type
Device # $fTPM_1$	1.2 GHz Cortex-A7
Device # $fTPM_2$	1.3 GHz Cortex-A9
Device # $fTPM_3$	2 GHz Cortex-A57
Device # $fTPM_4$	2.2 GHz Cortex-A57

Table 1: **Description of fTPM-equipped devices used the evaluation.**

them as $fTPM_1$ through $fTPM_4$, and $dTPM_1$ through $dTPM_3$. All mobile devices are commercially available both in USA and the rest of the world and can be found in the shops of most cellular carriers. Similarly, the discrete TPM 2.0 chips are commercially available. Table 1 describes the characteristics of the mobile ARM SoC processors present in the fTPM-equipped devices. The only modifications made to these devices' software is a form of device unlock that lets us load our own test harness and gather the measurement results. These modifications do not interfere with the performance of the fTPM running on the tablet.

Details of TPM 2.0 Commands. To answer the questions raised by our performance evaluation, we created a benchmark suite in which we perform various TPM commands and measure their duration. We were able to use timers with sub-millisecond granularity for all our measurements, except for device $fTPM_2$. Unfortunately, device $fTPM_2$ only exposes a timer with a 15-ms granularity to our benchmark suite, and we were not able to unlock its firmware to bypass this limitation.

Each benchmark test was run ten times in a row. Although this section presents a series of graphs that answer our performance evaluation questions, an interested reader can find all the data gathered in our benchmarks in the fTPM technical report [44].

• **Create RSA keys:** This TPM command creates an RSA key pair. When this command is issued, a TPM searches for prime numbers, creates the private and public key portions, encrypts the private portion with a root key, and returns both portions to the caller. We used 2048-bit RSA keys in all our experiments. We chose 2048-bit keys because they are the smallest key size still considered secure (1024-bit keys are considered insecure and their use has been deprecated in most systems).

• **Seal and unseal:** The TPM Seal command takes in a byte array, attaches a policy (such as a set of Platform Configuration Register (PCR) values), encrypts with its own storage key, and returns it to the caller. The TPM Unseal command takes in an encrypted blob, checks the policy, and decrypts the blob if the policy is satisfied by the TPM state (e.g., the PCR values are the same as at seal time). We used a ten-byte input array to Seal, and we set an empty policy.

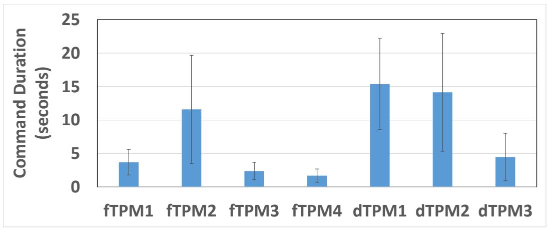

Figure 7: **Latency of create RSA-2048 keys on various fTPM and dTPM platforms.**

• **Sign and verify:** These TPM commands correspond to RSA sign and verify. We used a 2048-bit RSA key for RSA operations and SHA-256 for integrity protection.

• **Encryption and decryption:** These TPM commands correspond to RSA encryption and decryption. We used a 2048-bit RSA key for RSA operations, OAEP for padding, and SHA-256 for integrity protection.

• **Load:** This TPM command loads a previously-created RSA key into the TPM. This allows subsequent command, such as signing and encryption, to use the preloaded key. We used a 2048-bit RSA key in our TPM Load experiments.

10.2 Overhead of RSA Keys Creation

Figure 7 shows the latency of a TPM create RSA-2048 keys command across all our seven devices. As expected, creating RSA keys is a lengthy command taking several seconds on all platforms. These long latencies justify our choice of using cooperative checkpointing (see Section 7) in the design of the fTPM to avoid leaving the OS suspended for several seconds at a time.

Second, the performance of creating keys can be quite different across devices. $fTPM_2$ takes a much longer time than all other devices equipped with an fTPM. This is primarily due to the variations in the firmware performance across these devices – some manufacturers spend more time optimizing the firmware running on their platforms than others. Even more surprisingly, the discrete TPM 2.0 chips also have very different performance characteristics: $dTPM_3$ is much faster than $dTPM_1$ and $dTPM_2$. Looking at the raw data (shown in [44]), we believe that $dTPM_3$ searches for prime numbers in the background, even when no TPM command is issued, and maintains a cache of prime numbers.

Figure 7 also shows that the latency of creating keys has high variability due to how quickly prime numbers are found. To shed more light into the variability of finding prime numbers, we instrumented the fTPM codebase to count the number of prime candidates considered when creating an RSA 2048 key pair. For each test, all candidates are composite numbers (and thus discarded)

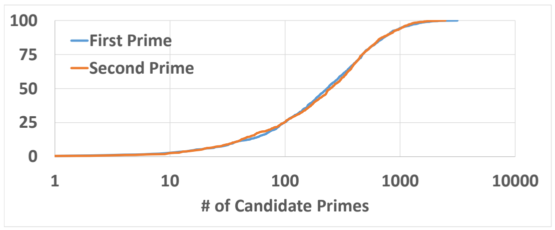

Figure 8: **Performance of searching for primes.**

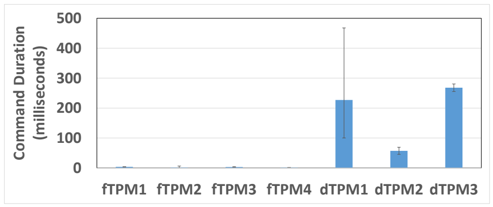

Figure 9: **Performance of TPM seal command.**

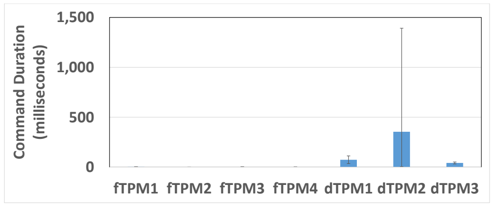

Figure 10: **Performance of TPM unseal command.**

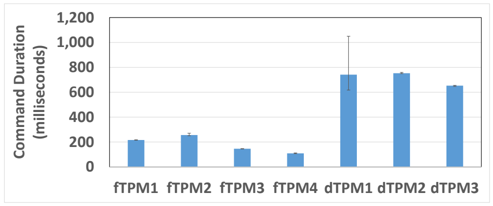

Figure 11: **Performance of TPM sign command.**

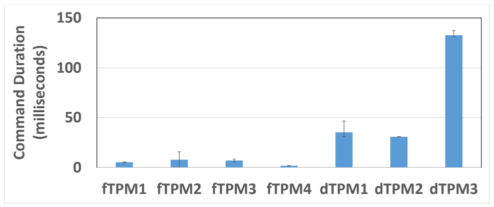

Figure 12: **Performance of TPM verify command.**

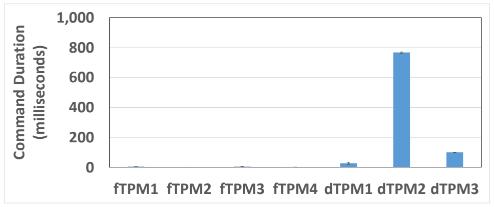

Figure 13: **Performance of TPM encrypt command.**

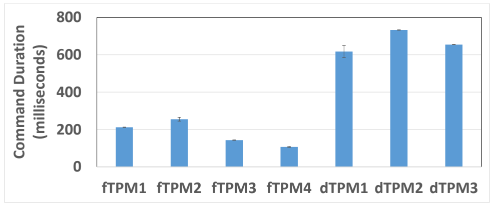

Figure 14: **Performance of TPM decrypt command.**

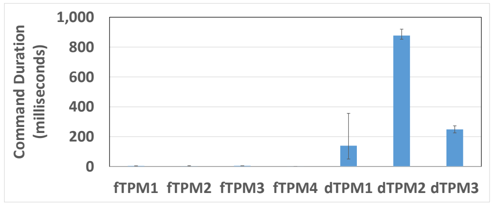

Figure 15: **Performance of TPM load command.**

except for the last number. We repeated this test 1,000 times. We plot the cumulative distribution function of the number of candidates for each of the two primes (p and q) in Figure 8. These results demonstrate the large variability in the number of candidate primes considered. While, on average, it takes about 200 candidates until a prime is found (the median was 232 and 247 candidates for p and q, respectively), sometimes a single prime search considers and discards thousands of candidates (the worst case was 3,145 and 2,471 for p and q, respectively).

10.3 Comparing fTPMs to dTPMs

Figures 9–15 show the latencies of several common TPM 2.0 commands. The main result is that fTPMs are much faster than their discrete counterparts. On average, the slowest fTPM is anywhere between 2.4X (for decryption) and 15.12X (for seal) faster than the fastest dTPM. This is not surprising because fTPMs run their code on ARM Cortex processors, whereas discrete chips are relegated to using much slower microprocessors. The fTPM technical report illustrates these vast performance improvements in even greater detail [44].

These performance results are encouraging. Traditionally, TPMs have not been used for bulk data cryptographic operations due to their performance limitations. With firmware TPMs however, the performance of these operations is limited only by processor speed and memory bandwidth. Furthermore, fTPMs could become even faster by taking advantage of crypto accelerators. Over time, we anticipate that crypto operations will increasingly abandon the OS crypto libraries in favor of the fTPM. This provides increased security as private keys never have to leave TrustZone's secure perimeter.

10.4 Evaluation Summary

In summary, our evaluation shows that (1) the firmware TPM has better performance than discrete TPM chips, and (2) creating RSA keys is a lengthy operation with high performance variability.

11 Security Analysis

The fTPM's security guarantees are not identical to those of a discrete TPM chip. This section examines these differences in greater depth.

On- versus off-chip. Discrete TPM chips connect to the CPU via a serial bus; this bus represents a new attack surface because it is externally exposed to an attacker with physical access to the main board. Early TPM chips were attached to the I^2C bus, one of the slower CPU buses, that made it possible for an attacker to intercept and issue TPM commands [49]. Modern TPM specifications have instructed the hardware manufacturers to attach the TPM chip to a fast CPU bus and to provide a secure *platform reboot signal*. This signal must guarantee that the TPM reboots (e.g., resets its volatile registers) *if and only if* the platform reboots.

In contrast, by running in the device's firmware, the fTPM sidesteps this attack surface. The fTPM has no separate bus to the CPU. The fTPM reads its state from secure storage upon initialization, and stores all its state in the CPU and the hardware-protected DRAM.

Memory attacks. By storing its secrets in DRAM, the fTPM is vulnerable to a new class of physical attacks – memory attacks that attempt to read secrets from DRAM. There are different avenues to mount memory attacks, such as cold boot attacks [23, 39], attaching a bus monitor to monitor data transfers between the CPU and system RAM [21, 17, 18], or mounting DMA attacks [6, 8, 42].

In contrast, discrete TPM chips do not make use of the system's DRAM and are thus resilient to such attacks. However, there is a corresponding attack that attempts to remove the chip's physical encasing, expose its internal dies, and thus read its secrets. Previous research has already demonstrated the viability of such attacks (typically referred to as *decapping* the TPM), although they remain quite expensive to mount in practice [26].

The fTPM's susceptibility to memory attacks has led us to investigate inexpensive counter-measures. Sentry is a prototype that demonstrates how the fTPM can become resilient to memory attacks. Sentry retrofits ARM-specific mechanisms designed for embedded systems but still present in today's mobile devices, such as L2 cache locking or internal RAM [10]. Note that in constrast with TrustZone, Intel SGX [25] provides hardware encryption of DRAM, which protects against memory attacks.

Side-channel attacks. Given that certain resources are shared between the secure and normal worlds, great care must be given to side-channel attacks. In contrast, a discrete TPM chip is immune to side-channel attacks that use caching, memory, or CPU because these resources are not shared with the untrusted OS.

a. Caches, memory, and CPU: The ARM TrustZone specification takes great care to reduce the likelihood of cache-based side-channel attacks for shared resources [1]. Cache-based side-channel attacks are difficult because caches are always invalidated during each transition to and from the secure world. Memory is statically partitioned between the two worlds at platform initialization time; such a static partitioning reduces the likelihood of side-channel attacks. Finally, the CPU also invalidates all its registers upon each crossing to and from the secure world.

b. Time-based attacks: The TPM 2.0 specification takes certain precautions against time-based attacks. For example, the entire cryptography subsystem of TPM 2.0 uses constant time functions – the amount of computation needed by a cryptographic function does not depend on the function's inputs. This makes the fTPM implementation as resilient to time-based side-channel attacks as its discrete chip counterpart.

12 Discussion

Most of ARM TrustZone's shortcomings stem from the nature of this technology: it is a standalone CPU-based security mechanism. CPU extensions alone are insufficient in many practical scenarios. As described earlier in Section 3.2, trusted systems need additional hardware support, such as support for trusted storage, secure counters, and secure peripherals.

Unfortunately, CPU designers continue to put forward CPU extensions aimed at building trusted systems that suffer from similar limitations. This section's goal is to describe these limitations in the context of a new, up-and-coming technology called Intel Software Guard Extensions (SGX). In the absence of additional hardware support for trusted systems, our brief discussion of SGX will reveal shortcomings similar to those of TrustZone.

12.1 Intel SGX Shortcomings

Intel SGX [25] is a set of extensions to Intel processors designed to build a sandboxing mechanism for running application-level code isolated from the rest of the system. Similar to ARM TrustZone's secure world, with Intel SGX applications can create *enclaves* protected from the OS and the rest of the platform software. All memory allocated to an enclave is hardware encrypted (unlike the secure world in ARM). Unlike ARM TrustZone, SGX does not offer any I/O support; all interrupts are handled by the untrusted code.

SGX has numerous shortcomings for trusted systems such as the fTPM:

1. Lack of trusted storage. While code executing inside an enclave can encrypt its state, encryption cannot protect against rollback attacks. Currently, the Intel SGX specification lacks any provision to rollback protection against persisted state.

2. Lack of a secure counter. A secure counter is important when building secure systems. For example, a rollback-resilient storage system could be built using encryption and a secure counter. Unfortunately, it is difficult for a CPU to offer a secure counter without hardware assistance beyond the SGX extensions (e.g., an eMMC storage controller with an RPMB partition).

3. Lack of secure clock. SGX leaves out any specification of a secure clock. Again, it is challenging for the CPU to offer a secure clock without extra hardware.

4. Side-channel dangers. SGX enclaves only protect code running in ring 3. This means that an untrusted OS is responsible for resource management tasks, which opens up a large surface for side-channel attacks. Indeed, recent work has demonstrated a number of such attacks against Intel SGX [57].

13 Related Work

Previous efforts closest to ours are Nokia OnBoard credentials (ObC), Mobile Trusted Module (MTM), and previous software implementations of TPMs. ObC [29] is a trusted execution runtime environment leveraging Nokia's implementation of ARM TrustZone. ObC can execute programs written in a modified variant of the LUA scripting language or written in the underlying runtime bytecode. Different scripts running in ObC are protected from each other by the underlying LUA interpreter. A more recent similar effort ported the .NET framework to TrustZone [45, 46] using techniques similar to ObC.

While the fTPM serves as the reference implementation of a firmware TPM for ARM TrustZone, ObC is a technology proprietary to Nokia. Third-parties need their code signed by Nokia to allow it to run inside TrustZone. In contrast, the fTPM offers TPM 2.0 primitives to any application. While TPM primitives are less general than a full scripting language, both researchers and industry have already used TPMs in many secure systems demonstrating its usefulness. Recognizing the TPM platform's flexibility, ObC appears to have recently started to offer primitives more compatible with those of the TPM specification [15].

The Mobile Trusted Module (MTM) [51] is a specification similar to a TPM but aimed solely at mobile devices. Previous work investigated possible implementations of MTM for mobile devices equipped with secure hardware, such as ARM TrustZone, smartcards, and Java SecureElements [12, 13]. These related works acknowledged upfront that the limitations of ARM TrustZone for implementation MTM remain future work [12]. Unfortunately, MTMs have not gone past the specification stage in the Trusted Computing Group. As a result, we are unaware of any systems that make use of MTMs. If MTMs were to become a reality, our techniques would remain relevant in building a firmware MTM.

A more recent article presents a high-level description of the work needed to implement TPM 2.0 both in hardware and in software [34]. Like the fTPM, the article points out the need of using a replay-protected memory block partition to protect against replay attacks. However, this article appeared much later, after the fTPM was launched in mobile devices. It is unclear whether any implementation of their architecture exists.

IBM has been maintaining a software implementation of TPM 1.2 [24]. An independent effort implemented a TPM 1.2 emulator without leveraging any secure hardware [50]. This emulator was aimed at debugging scenarios and testbeds. We are unaware of efforts to integrate any of these earlier implementations into mobile devices.

Another area of related work is building virtualized TPM implementations. Virtual TPMs are needed in virtualized environments where multiple guest operating systems might want to share the physical TPM without having to trust each other. Several designs of virtual TPMs have been proposed [7, 16].

Finally, a recent survey describes additional efforts in building trusted runtime execution environments for mobile devices based on various forms of hardware, including physically uncloneable functions, smartcards, and embedded devices [4]. A recent industrial consortium called GlobalPlatform [20] has also started to put together a standard for trusted runtime execution environments on various platforms, including ARM [3].

14 Conclusions

This paper demonstrates that the limitations of CPU-based security architectures, such as ARM TrustZone, can be overcome to build software systems with security guarantees similar to those of dedicated trusted hardware. We use three different approaches to overcome these challenges: requiring additional hardware support, making design compromises without affecting security, and slightly changing command semantics.

This paper describes a software-only implementation of a TPM chip. Our software-only TPM requires no application-level changes or changes to OS components (other than drivers). Our implementation is the reference implementation of TPM 2.0 used in millions of smartphones and tablets.

Acknowledgements We would like to thank Andrew Baumann, Weidong Cui, Roxana Geambasu, Jaeyeon Jung, and Angelos Keromytis for feedback on earlier drafts of this paper. We are also grateful to Jiajing Zhu for her help with the TPM 2.0 simulator, and numerous other collaborators who contributed to the firmware TPM effort. Finally, we would like to thank the anonymous reviewers for their feedback on the submission.

References

[1] ARM Security Technology – Building a Secure System using TrustZone Technology. ARM Technical White Paper, 2005-2009.

[2] Virtualization is Coming to a Platform Near You. ARM Technical White Paper, 2010-2011.

[3] ARM. GlobalPlatform based Trusted Execution Environment and TrustZone Ready. http://community.arm.com/servlet/JiveServlet/previewBody/8376-102-1-14233/GlobalPlatform%20based%20Trusted%20Execution%20Environment%20and%20TrustZone%20Ready%20-%20Whitepaper.pdf.

[4] ASOKAN, N., EKBERG, J.-E., KOSTIANEN, K., RAJAN, A., ROZAS, C., SADEGHI, A.-R., SCHULZ, S., AND WACHSMANN, C. Mobile Trusted Computing. *Proceedings of IEEE 102*, 1 (2014), 1189–1206.

[5] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications from an Untrusted Cloud with Haven. In *Proc. of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, CO, 2014).

[6] BECHER, M., DORNSEIF, M., AND KLEIN, C. N. FireWire - all your memory are belong to us. In *Proc. of CanSecWest Applied Security Conference* (2005).

[7] BERGER, S., CCERES, R., GOLDMAN, K. A., PEREZ, R., SAILER, R., AND VAN DOORN, L. vtpm: Virtualizing the trusted platform module. In *Proc. of the 15th USENIX Security Symposium* (2006).

[8] BOILEAU, A. Hit by a Bus: Physical Access Attacks with Firewire. In *Proc. of 4th Annual Ruxcon Conference* (2006).

[9] CHEN, C., RAJ, H., SAROIU, S., AND WOLMAN, A. cTPM: A Cloud TPM for Cross-Device Trusted Applications. In *Proc. of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Seattle, WA, 2014).

[10] COLP, P., ZHANG, J., GLEESON, J., SUNEJA, S., DE LARA, E., RAJ, H., SAROIU, S., AND WOLMAN, A. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proc. of 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Istanbul, Turkey, 2015).

[11] DATALIGHT. What is eMMC. http://www.datalight.com/solutions/technologies/emmc/what-is-emmc.

[12] DIETRICH, K., AND WINTER, J. Implementation Aspects of Mobile and Embedded Trusted Computing. *Proc. of 2nd International Conference on Trusted Computing and Trust in Information Technologies (TRUST), LNCS 5471* (2009), 29–44.

[13] DIETRICH, K., AND WINTER, J. Towards Customizable, Application Specific Mobile Trusted Modules. In *Proc. of 5th ACM Workshop on Scalable Trusted Computing (STC)* (Chicago, IL, 2010).

[14] ECRYPTFS. eCryptfs – The enterprise cryptographic filesystem for Linux. http://ecryptfs.org/.

[15] EKBERG, J.-E. Mobile information security with new standards: GlobalPlatform/TCG/Nist-root-of-trust. Cyber Security and Privacy, EU Forum, 2013.

[16] ENGLAND, P., AND LÖSER, J. Para-virtualized tpm sharing. *Proc. of 1st International Conference on Trusted Computing and Trust in Information Technologies (TRUST), LNCS 4968* (2008), 119–132.

[17] EPN SOLUTIONS. Analysis tools for DDR1, DDR2, DDR3, embedded DDR and fully buffered DIMM modules. http://www.epnsolutions.net/ddr.html. Accessed: 2014-12-10.

[18] FUTUREPLUS SYSTEM. DDR2 800 bus analysis probe. http://www.futureplus.com/download/datasheet/fs2334_ds.pdf, 2006.

[19] GILBERT, P., JUNG, J., LEE, K., QIN, H., SHARKEY, D., SHETH, A., AND COX, L. P. YouProve: Authenticity and Fideltiy in Mobile Sensing. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)* (Lake District, UK, 2012).

[20] GLOBALPLATFORM. Technical Overview. http://www.globalplatform.org/specifications.asp.

[21] GOGNIAT, G., WOLF, T., BURLESON, W., DIGUET, J.-P., BOSSUET, L., AND VASLIN, R. Reconfigurable hardware for high-security/high-performance embedded systems: The SAFES perspective. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16*, 2 (2008), 144–155.

[22] GOOGLE. The Chromium Projects. http://www.chromium.org/developers/design-documents/tpm-usage.

[23] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: Cold boot attacks on encryption keys. In *Proc. of the 17th USENIX Security Symposium* (2008).

[24] IBM. Software TPM Introduction. http://ibmswtpm.sourceforge.net/.

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

[26] JACKSON, W. Engineer shows how to crack a 'secure' TPM chip. http://gcn.com/Articles/2010/02/02/Black-Hat-chip-crack-020210.aspx, 2010.

[27] KOCKER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. of 16th Annual International Cryptology Conference (CRYPTO)* (Santa Barbara, CA, 1996).

[28] KOSTIAINEN, K., ASOKAN, N., AND EKBERG, J.-E. Practical Property-Based Attestation on Mobile Devices. *Proc. of 4th International Conference on Trusted Computing and Trust in Information Technologies (TRUST), LNCS 6470* (2011), 78–92.

[29] KOSTIAINEN, K., EKBERG, J.-E., ASOKAN, N., AND RANTALA, A. On-board Credentials with Open Provisioning. In *Proc. of the 4th International Symposium on Information, Computer, and Communications Security (ASIA CCS)* (2009).

[30] KOTLA, R., RODEHEFFER, T., ROY, I., STUEDI, P., AND WESTER, B. Pasture: Secure Offline Data Access Using Commodity Trusted Hardware. In *Proc. of 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywoood, CA, 2012).

[31] LIU, H., SAROIU, S., WOLMAN, A., AND RAJ, H. Software Abstractions for Trusted Sensors. In *Proc. of 10th International*

Conference on Mobile Systems, Applications, and Services (MobiSys) (Lake District, UK, 2012).

[32] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of IEEE Symposium on Security and Privacy* (Oakland, CA, May 2010).

[33] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)* (Glasgow, UK, 2008).

[34] McGill, K. N. Trusted Mobile Devices: Requirements for a Mobile Trusted Platform Module. *Johns Hopkings Applied Physical Laboratory Technical Digest 32*, 2 (2013).

[35] Microsoft. Early launch antimalware. http://msdn.microsoft.com/en-us/library/windows/desktop/hh848061(v=vs.85).aspx.

[36] Microsoft. HealthAttestation CSP. https://msdn.microsoft.com/en-us/library/dn934876%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396.

[37] Microsoft. Help protect your files with BitLocker Driver Encryption. http://windows.microsoft.com/en-us/windows-8/using-bitlocker-drive-encryption.

[38] Microsoft. Understanding and Evaluating Virtual Smart Cards. http://www.microsoft.com/en-us/download/details.aspx?id=29076.

[39] Müller, T., and Spreitzenbarth, M. FROST - forensic recovery of scrambled telephones. In *Proc. of the International Conference on Applied Cryptography and Network Security (ACNS)* (2013).

[40] NIST. Digital Signature Standard (DSS). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[41] Parno, B., Lorch, J. R., Douceur, J. R., Mickens, J., and McCune, J. M. Memoir: Practical State Continuity for Protected Modules. In *Proc. of IEEE Symposium on Security and Privacy* (Oakland, CA, 2011).

[42] Piegdon, D. R. Hacking in physically addressable memory - a proof of concept. Presentation to the Seminar of Advanced Exploitation Techniques, 2006.

[43] Raj, H., Robinson, D., Tariq, T., England, P., Saroiu, S., and Wolman, A. Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor. Tech. Rep. MSR-TR-2011-130, Microsoft Research, 2011.

[44] Raj, H., Saroiu, S., Wolman, A., Aigner, R., Jeremiah Cox, a. P. E., Fenner, C., Kinshumann, K., Loeser, J., Mattoon, D., Nystrom, M., Robinson, D., Spiger, R., Thom, S., and Wooten, D. fTPM: A Firmware-based TPM 2.0 Implementation. Tech. Rep. MSR-TR-2015-84, Microsoft, 2015.

[45] Santos, N., Raj, H., Saroiu, S., and Wolman, A. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proc. of 12th Workshop on Mobile Computing Systems and Applications (HotMobile)* (Phoenix, AZ, 2011).

[46] Santos, N., Raj, H., Saroiu, S., and Wolman, A. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *Proc. of 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Salt Lake City, UT, 2014).

[47] Santos, N., Rodrigues, R., Gummadi, K. P., and Saroiu, S. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proc. of the 21st USENIX Security Symposium* (Bellevue, WA, 2012).

[48] Saroiu, S., and Wolman, A. I Am a Sensor and I Approve This Message. In *Proc. of 11th International Workshop on Mobile Computing Systems and Applications (HotMobile)* (Annapolis, MD, 2010).

[49] Sparks, E., and Smith, S. W. TPM Reset Attack. http://www.cs.dartmouth.edu/~pkilab/sparks/.

[50] Strasser, M., and Stamer, H. A Software-Based Trusted Platform Module Emulator. *Proc. of 1st International Conference on Trusted Computing and Trust in Information Technologies (TRUST), LNCS 4968* (2008), 33–47.

[51] Trusted Computing Group. Mobile Trusted Module Specification. http://www.trustedcomputinggroup.org/resources/mobile_phone_work_group_mobile_trusted_module_specification.

[52] Trusted Computing Group. TCPA Main Specification Version 1.1b. http://www.trustedcomputinggroup.org/files/resource_files/64795356-1D09-3519-ADAB12F595B5FCDF/TCPA_Main_TCG_Architecture_v1_1b.pdf.

[53] Trusted Computing Group. TPM 2.0 Library Specification FAQ. http://www.trustedcomputinggroup.org/resources/tpm_20_library_specification_faq.

[54] Trusted Computing Group. TPM Library Specification. http://www.trustedcomputinggroup.org/resources/tpm_library_specification.

[55] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2, Revision 116. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[56] Wolman, A., Saroiu, S., and Bahl, V. Using Trusted Sensors to Monitor Patients' Habits. In *Proc. of 1st USENIX Workshop on Health Security and Privacy (HealthSec)* (Washington, DC, 2010).

[57] Xu, Y., Cui, W., and Peinado, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proc. of the 36th IEEE Symposium on Security and Privacy (Oakland)* (2015).

[58] Zhang, F., Chen, J., Chen, H., and Zang, B. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proc. of Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, 2011).