Microsoft Research

fTPM: A Firmware-based TPM 2.0 Implementation

H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England,
C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon,
M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten

MSR-TR-2015-84
November 9, 2015

fTPM: A Firmware-based TPM 2.0 Implementation

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: *This paper presents the design and implementation of a firmware-based TPM 2.0 (fTPM) leveraging ARM TrustZone. The fTPM is the reference implementation used in millions of mobile devices, and was the first hardware or software implementation to support the newly released TPM 2.0 specification.*

This paper describes the shortcomings of ARM's TrustZone for implementing secure services (such as our implementation), and presents three different approaches to overcome them. Additionally, the paper analyzes the fTPM's security guarantees and demonstrates that many of the ARM TrustZone's shortcomings remain present in future trusted hardware, such as Intel's Software Guard Extensions (SGX).

1 Introduction

The Trusted Platform Module (TPM) chip is one of the most popular forms of trusted hardware. Industry has started broad adoption of TPMs for enabling security features including preventing rollback [17] (Google), protecting data at rest [30, 17] (Microsoft and Google), virtualizing smart cards [31] (Microsoft), and early-launch anti-malware [28]. At the same time, the research community has started to propose even more ambitious uses of TPMs such as secure offline data access [24], new trusted OS abstractions [40], trusted sensors [25], and protecting guest VMs from the VMM or the management VM [49, 36].

Despite their importance, many smartphones and tablets lack TPM chips. Mobile devices are constrained in terms of space, cost, and power dimensions that make the use of a discrete TPM chip difficult. Recognizing the incompatibility of TPMs with mobile device requirements, the Trusted Computing Group (TCG) has previously proposed a new standard called Mobile Trusted Module (MTM) [42]. Unfortunately, the MTM specification has lacked broad industry support, and has never been widely adopted in practice in spite of the much efforts by TCG. The absence of trusted hardware prevents mobile devices from adopting the recent security features developed by the research community and industry.

Fortunately, smartphones and tablets use ARM, an architecture that incorporates trusted computing support in hardware. ARM TrustZone offers a runtime environment isolated from the rest of the software on the platform including the OS, the applications, and most of the firmware. Any exploit or malware present in this software cannot affect the integrity and confidentiality of code and data running in ARM TrustZone. Such a level of support makes it possible to implement secure services that offer security guarantees similar to those of secure co-processors, such as TPMs.

This paper presents firmware-TPM (fTPM), an end-to-end implementation of a TPM using ARM TrustZone. Our implementation is the reference implementation used in all ARM-based Windows mobile devices including Microsoft Surface and Windows Phones, which comprises millions of mobile devices. fTPM provides security guarantees similar (although not identical) to a discrete TPM chip. fTPM was the first hardware or software implementation to support the newly released TPM 2.0 specification.

This paper makes the following contributions:

1. It provides an analysis of the ARM TrustZone's security guarantees. In the course of this analysis, we uncover a set of shortcomings of the ARM TrustZone technology needed for building secure services, whether the fTPM or others.

2. It presents the first design and implementation of a TPM 2.0 specification using the ARM TrustZone security extensions. This is the reference implementation used in millions of ARM-based Windows mobile devices.

3. It describes three techniques for overcoming ARM TrustZone's shortcomings: (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 the TrustZone's limitations. Our techniques are general and extend to building other secure services inside based on ARM TrustZone.

4. It analyzes the security guarantees of the fTPM and compares them with those of a discrete TPM chip counterpart.

5. Finally, it demonstrates that many of the shortcomings of ARM TrustZone technology remain present in

future trusted hardware, such as the up and coming Intel Software Guard Extensions (SGX) technology [20].

2 Trusted Platform Module: An Overview

Trusted Platform Modules (TPMs) are enjoying a resurgence of interest from both industry and the research community. Although over a decade old, TPMs have had a mixed history due to a combination of factors. One of the scenarios driving TPM adoption was digital rights management (DRM), a scenario often labelled as users giving 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 [43] was shown to be vulnerable to an unsophisticated attack, known as the *PIN reset* attack [41]. Over time, however, TPMs have been able to overcome their mixed reputation, and become 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. The remainder of 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, virtual smart cards, early launch anti-malware (ELAM), and key and device health attestations.

BitLocker [30] is a full-disk encryption system that uses the TPM to lock the encryption keys. Because the decryption 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 [31] 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 outside of the TPM, and the cryptography is isolated from the rest of the system.

ELAM [28] enables Windows to load anti-malware before all third-party boot drivers and applications. The anti-malware software can be first-party (e.g., Microsoft's Windows Defender) or third-party (e.g., Symantec's Endpoint Protection). Finally, Windows also uses the TPM to construct attestations of cryptographic keys and device boot parameters [29]. Enterprise IT managers use these attestations to assess the health of the devices they manage. A common use is gating access to high-value network resources based on the current state of a machine.

Google. Modern versions of Chrome OS [17] 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 allows a remote party (e.g., Google) to update the firmware or the OS in Chrome devices. Such updates are vulnerable to "remote rollback attacks", in which a remote attacker replaces newer software, through a hard-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 [11] is a disk encryption system used by Chrome OS to protect user data. Chrome OS uses the TPM to make parallelized attacks and password brute-forcing on eCryptfs's symmetric (AES) keys difficult. Any attempt at guessing the AES keys requires the use of a TPM, a single-threaded device that is relatively slow. The TPM allows Chrome OS to acquire a level of brute-force protection because it effectively throttles the rate at which guesses can be made.

A Chrome device can be booted in four different modes, corresponding to the settings of two switches (physical or virtual) at power on. They are the developer switch and the recovery switch. They 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 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 [17].

2.2 TPM-based Secure Systems In Research

The use of TPMs in novel secure systems has exploded in the research community 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 [49] and Credo [36] are virtualization-approaches that protect the privacy and integrity of customer's VMs on commodity cloud infrastructure, even when facing a total compromise of the virtual machine monitor (VMM) and the management VM. These systems require TPMs to attest to cloud customers the secure configuration of the physical nodes running their VMs.

Secure applications, OSs and hypervisors. Flicker [27], TrustVisor [26], Memoir [34] 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' TCB is small because they exclude the bulk of the OS.

Novel secure functionality. Pasture [24] 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 [38] is 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 nodes whose configurations match the policy. The abstraction relies on TPMs to identify a cloud node's configuration.

cTPM [8] 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 trusted sensors [14, 25] 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) [39, 47].

2.3 TPM 2.0: A New TPM Specification

The Trusted Computing Group (TCG) has defined the specification for TPM version 2.0 [45], which is the successor to TPM version 1.2 [46]. A newer TPM has been needed for two primary reasons. First, the crypto requirements of TPM 1.2 have become inadequate. For example, TPM 1.2 offers SHA-1 only, but not SHA-2; SHA-1 is now considered weak and cryptographers are reluctant to use it any longer. Another example is the introduction of ECC to TPM 2.0.

The second reason for TPM 2.0 is the lack of an universally-accepted reference implementation of the TPM 1.2 specification. As a result different implementations of TPM 1.2 exist with, arguably, slightly different behaviors. Another problem is that the lack of a reference implementation has made TPM 1.2 specification ambiguous. It is difficult to specify the exact behavior of cryptographic protocols in English. Instead, TPM 2.0 specification itself is the same as the reference implementation. TPM 2.0 comes with several documents that describe the behavior of the codebase, but these documents are in fact derived from TPM 2.0 codebase itself. This removes the need for creating alternative implementations of TPM 2.0, a step towards behavior uniformization.

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 functionalities. 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 could break (brick the PC) if the TPM chip would be turned into a TPM 2.0-only chip. A list of differences between the two versions is provided by the TCG [44].

3 Modern Trusted Computing Hardware

Recognizing the increasing demand for security, modern hardware has started to incorporate features specifically designed for trusted computing, such as ARM TrustZone [1] and Intel Software Guard Extensions (SGX) [20]. This section presents the background on ARM TrustZone (including its shortcomings); this background is important to the design of fTPM. Later in the paper, Section 13 will describe the soon-to-be-available 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 partitioned and dedicated to the secure world. Similarly, processor, caches, and controllers are often shared across worlds. Finally, I/O controllers and devices can be mapped to a one world only.

Secure monitor. The secure monitor is an ARM processor mode designed to switch between the secure and normal worlds. The ARM processor has many additional operating modes (their number varies for different ARM Cortex processors) that can be either secure or non-secure. A specially designed register determines whether the processor runs code in the secure or nonsecure worlds. When the core runs in secure monitor mode the state is considered secure regardless of the state of this register.

ARM has separate banked copies 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 (e.g., an access violation error is raised). However, the secure monitor can access nonsecure 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 had a chance to run. For example, the firmware allocates memory for the 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 must use a special ARM instruction called *smc* (secure monitor call), to call back into the secure world. When the CPU executes the *smc* instruction, the hardware switches into a 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 system bus. This signal corresponds to an extra bit (a 33rd-bit on a 32-bit architecture) called the *nonsecure* 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, the specification is silent on how most other resources should be protected. This has led to fragmentation – SoCs offer various forms of protecting different hardware resources for the TrustZone, or no protection at all.

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

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

Crypto needs. Most trusted systems make use of cryptography. However, the specification is silent on offering a secure entropy source or a monotonically increasing counter. As a result, most SoCs lack an entropy pool that can be read from the secure world, or 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 should be done using virtualization techniques. Although ARM offers virtualization extensions [2], the ARM TrustZone specification does not mandate them. As a result, most ARM-based SoCs used in mobile devices today lack virtualization support. Virtualizing commodity operating systems (e.g., Windows) on an ARM platform lacking hardware-assistance for virtualization is very difficult.

Lack of secure clock (and other peripherals). Secure systems often require a secure clock. While TrustZone access to protected memory and interrupts is a step forward to offering secure peripherals, it is often insufficient without protecting the bus controllers that can talk to these peripherals. It is hard to reason about the security

ARM TrustZone Shortcomings
No trusted storage
No secure entropy source
Lack of virtualization
No secure clock
No secure peripherals
Lack of firmware access

Figure 1. **The shortcomings of ARM TrustZone.**

guarantees of a peripheral whose controller can be programmed by the normal world, even when its interrupts and memory region are mapped to the secure world only. 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 the TrustZone. In our experience, this has seriously impeded the adoption of the 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.

Figure 1 summarizes the list of shortcomings of the ARM TrustZone architecture when building secure systems.

4 High-Level Architecture

Leveraging ARM TrustZone, we implemented a trusted execution environment (TEE) that acts as a basic operating system for the secure world and runs the fTPM.

4.1 Trusted Execution Environment (TEE)

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. Figure 2 illustrates this architecture. The shaded boxes represent system's TCB

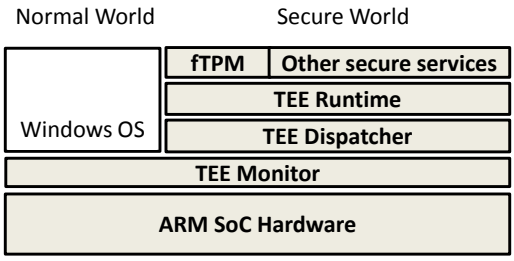

Figure 2. **The architecture of the fTPM.** This schematic is not to scale.

that comprises the ARM SoC hardware, the TEE layers, and the fTPM.

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:

• *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 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.

4.2 Threat Model and Assumptions

A primary assumption is that the commodity OS running in the ARM's Normal World is untrusted and potentially compromised. This OS could mount various attacks to code running in the TrustZone, such as making invalid calls to the TrustZone (or setting invalid parameters), not responding to requests coming from the 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 returns 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 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.

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 [22]. However, we do not defend against power analysis or any other type of side-channel attacks that require access to hardware or hardware modifications.

We turn our focus on the approaches taken to overcome TrustZone's shortcomings in the fTPM. We leave the details of the TEE implementation to Section 9.

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 because of its drastic impact on device usability. TPMs are used to measure the software (including the firmware) booting on a device. A mobile device would then require cloud connectivity to boot up in order to download the fTPM's state and start measuring the booting software. The requirement of having cloud connectivity in order to boot up a smartphone was not a viable option.

We discovered instead that many mobile devices come equipped with an eMMC storage controller that has a replay-protected memory block (RPMB). The RPMB's presence (combined with encryption) ensures that TEE can offer storage that meets all the fTPM's security property, and formed our *first hardware requirement* for TEE.

Second, we required the presence of a hardware fuse available to the secure world only. A hardware fuse is a write-once storage location. At provisioning time (before being release to a store), our mobile devices provision this secure hardware fuse with a secure key unique per device. Finally, we also required an entropy source that can be read from the secure world. The TEE uses the combination of the secure key and entropy source to generate cryptographic keys at boot time.

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

Approach #2: Design Compromises. Another big concern was long-running TEE commands. Running inside the 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 from the hardware. Speaking to the hardware vendors, we learned that it is unlikely virtualization will be added to their platforms any time soon.

Instead, we compromised on the TEE design and required that no TEE code path can execute for a long period of time. This translated into a requirement for the fTPM – no TPM 2.0 command can be long running. Our measurements of TPM commands revealed that no TPM 2.0 commands are long running except one: generating RSA keys. Section 7 will present the compromise made to the fTPM design when issued an RSA key generation command.

Approach #3: Reducing the TPM 2.0 Semantics. Lastly, we required 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 Approach #1: Hardware Requirements

6.1 eMMC with RPMB

The term *eMMC* is short 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 [10]. eMMC consists of the MMC (multimedia card) interface, the flash memory, and the flash memory controller. eMMC offers a replay-protected memory block (RPMB) partition. Like its name suggests, RPMB is a mechanism for storing data in an authenticated and replay-protected manner.

The RPMB offers two storage primitives: authenticated writes and authenticated reads.

Authenticated Writes: An authenticated write request comprises of multiple dataframes carrying data followed by a result-read-request dataframe. An authenti-

cated write request has an HMAC computed over all the data (i.e., all the blocks); the HMAC is added to the last dataframe carrying data. Each dataframe also includes the address where the data should be written on the partition as well as a nonce.

Once all dataframes carrying data have been issued, the caller must issue a result-read-request to determine whether the write has been successful or not. There are many reasons why the write could have failed, including an integrity check failure (i.e., the HMAC did not compute properly), a write counter reaching its maximum value, receiving a high-priority interrupt during the write, or a general hardware failure. Thus, an authenticated write is made of one or more dataframes carrying data followed by a result read request dataframe which will return an authenticated write response.

Authenticated Reads: An authenticated read request is made of just one dataframe that can issue a read call of many 256-byte blocks. Once this dataframe is issued, a number of dataframes carrying data can be read. The numbers of dataframes to be read is equal to the number of blocks specified in the read call.

6.1.1 RPMB Mechanism

RPMB's replay protection comprises of a set of 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 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. Once issued, a result read request dataframe must be also issued to check that the programming step has been successful. Access to the RPMB is not possible until the authentication key has been programmed. Any authenticated 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 value for the number of authenticated write requests made to RPMB. This is a 32-bit counter and is initially set to 0. Once, it reaches its maximum value, the counter will not be incremented further and a special bit will be turned on in all dataframes to indicate that the write counter has expired permanently. 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 nonces that are reflected in the read responses. These nonces ensure that reads are fresh.

6.1.2 Protection against replay attacks

A dataframe includes a 16-byte nonce field. The nonce is used only in two operations: authenticated read and read counter value. The nonce is not used during authenticated write, nor during programming the RPMB key.

The role of the nonce in the two read operations protects them against replay attacks. The secure world and the eMMC controller share a secret (the RPMB authentication key). Whenever a read operation is issued, a nonce is included to ensure the freshness of its result.

Authenticated writes make no use of nonces. Instead, they include a write counter value whose integrity is protected by the authentication key. The read request dataframe that ends an authenticated write returns a dataframe the incremented counter value, whose integrity is protected by the shared secret (the RPMB authentication key). This ensures that the write request has been successfully written to storage.

6.2 Requirement #2: Secure World Hardware Fuse

We required a hardware fuse that can be read from the secure world only. The fuse is 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 10 will describe in-depth how the seed is used in deriving secret fTPM keys, such as the secure storage key (SSK).

6.3 Requirement #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 could take some time until it has gathered "enough" bits of entropy for a seed.

We required 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 of 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 a 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 due to two reasons. First, it requires searching for two large prime numbers, and such a search can theoretically take an unbounded amount of time. Although many optimizations on how to search RSA keys efficiently exist [33], 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 the entropy required for seeding 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 level of entropy required for seeding 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 a test called the Miller-Rabin probabilistic primality test [33]. 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 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 targeted several different ARM platforms (from smartphones to tablets), most of which lacked virtualization support. The lack of virtual support required the transitions to TEE and back to be very short to ensure that the commodity OS would remain stable. We were faced with a new fTPM requirement: no long-running TEE commands. Unfortunately, creating an RSA key is a very long process, often taking in excess of 10 seconds on our early hardware tablets.

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

Faced with this challenge, we added *cooperative checkpointing* to the fTPM. Whenever a TPM command takes too long, the TPM 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 OS can cancel any TPM command even when in the command is in a checkpointed state.

Cooperative checkpointing enabled us to bypass the lack of virtualization support in ARM, yet continue to offer long-running TPM commands, such creating RSA keys.

8 Reducing TPM 2.0 Semantics

8.1 Secure Clock

TPMs use secure clocks for two reasons. First use is to measure lockout durations. Lockouts are time periods during which 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.

8.1.1 The Requirement 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 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/shutdown/bootup automatically expires all time-bound authorizations).

8.1.2 Reducing the Semantics of Secure Clock

We reduced the semantics of the clock functionality of the TPM 2.0 specification. While the fTPM's clock can measure lockout durations securely (e.g., the fTPM refuses service for the next k seconds), it cannot be used for time-bound authorizations (e.g., the fTPM allows service for the next k seconds). This distinction stems from the TEE's inability to guarantee that the secure clock moves forward. A compromised OS could stop the clock allowing time-bound authorization to continue to be valid indefinitely.

Fortunately, Windows only requires a secure implementation of lockout from the fTPM, and does not make use of time-bound authorizations. Based on our understanding, Chrome devices also do not appear to use time-bound authorizations. The fTPM clock implementation can guarantee a secure implementation of lockout meeting Chromium and Windows's needs.

8.2 Dark Periods

The diversity of mobile device manufacturers raised an additional challenge to 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 finished running, but before the OS started 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 unsealed 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 guessing the unseal authorization (e.g., Windows lets users to enter a PIN number to properly unseal the OS loader using BitLocker). The TPM maintains a counter

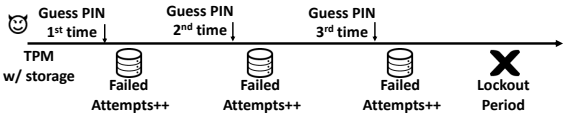

Figure 3. **TPM with storage.** TPM enters lockout if adversary makes too many guess attempts. This sequence of steps is secure

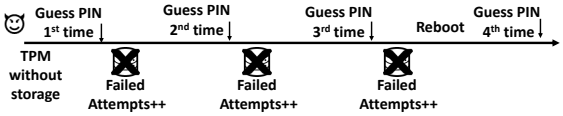

Figure 4. **TPM without stable storage is insecure.** Without storing the failed attempts counter, the adversary can simply reboot and avoid TPM lockout. This sequence of steps is insecure.

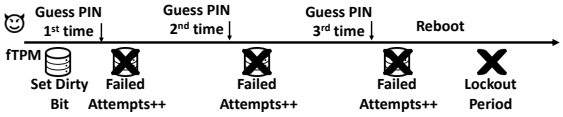

Figure 5. **fTPM.** fTPM sets the dirty bit before entering a dark period. If reboot occurs during the dark period, fTPM enters lockout automatically. This sequence of steps is secure.

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, respectively.

8.2.1 Reducing the Semantics of the Failed Tries Counter

We addressed the lack of storage during a dark period by making a slight change in how the TPM 2.0 interprets the failed tries counter. Before entering a dark period, the fTPM persists a *dirty bit*. If the dark period is entered and the unseal succeeds, the OS would 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 initalizes 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 legit device reboot during a dark period puts the TPM in lockout. The TPM cannot unseal until the lockout duration expires (typically set to several minutes).

Alerted to this problem, the TPM 2.0 designers have added a form of the *dirty bit* to their specification, called *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 TEE Implementation

TEE Monitor. The monitor layer consists of the lowest level, most privileged code that runs on the platform. Its role is to implement context switching between the normal and secure worlds. Context switching occurs when an interrupt arrives or when the normal world issues an *smc* instruction to switch to the secure world. The monitor implements two types of context switches: full and lightweight.

Full context switches save the all normal world's processor register state for all processor modes and restores the state of secure world. On each entry or exit from the secure world, 768 bytes are saved or restored. All commands issued to the fTPM use full context switches.

Normal world uses lightweight context switches when it needs to issue a command to a resource managed by the secure world. For example, the untrusted OS must invoke TEE for L2 cache maintenance operations (the secure world manages the L2 cache). These operations do not need to invoke any of the secure services in the TEE. Lightweight context switches do not perform any memory save or restore operations and are therefore very fast.

Implementation Details: ARM registers r0-r3 are volatile and used for parameter passing. In compliance with ARM calling convention, the caller is responsible for saving these registers if needed. Currently, we do not support parameter passing via stack. Instead, the monitor maintains a context switch area in which registers are saved and from which are restored. Our current implementation uses r0 as a service ID (SID) to multiplex between services. This allows us to multiplex on which path to take in monitor based on the SID, whether to invoke a lightweight context switch or a full one.

In the future, the monitor can implement strong isolation between TEE services. For example, the monitor can schedule and serve two mutually distrusting services that both run inside the secure world but within address spaces that are strongly isolated.

TEE Dispatcher. On a full context switch to the secure world, the dispatcher further de-multiplexes requests to the appropriate service using r0 as the secure process ID (SID). The dispatcher passes registers r1-r3 to the service specific callback routine. The dispatcher returns back the normal world using the smc instruction, with r0 set to the return code. Contents of r1-r3 are undefined.

TEE Runtime. TEE has a minimal UEFI-based runtime environment that provides functionality and interfaces to secure services. It implements a heap for runtime, and dynamic memory allocation for all secure services including the fTPM. It also provides runtime libraries that implement each of the design requirements listed above, in the previous section.

UEFI Initialization. On device power-on, the BootROM loads the UEFI Firmware Device (FD) image and verifies its integrity using a public key stored on the read-only flash-backed partition. If the image is corrupt (for example, an attacker replaced the image on stable storage), an error is returned and the device refuses booting. Otherwise, the FD is loaded in the the first 32MB of main memory of the secure world and begins executing.

The FD code sets up the interrupt controller to mark interrupts secure and insecure, and the memory controller to protect the first 32MB of RAM (denoted as Secure RAM). It then sets up page tables to use the secure mode for Secure RAM and non-secure mode for the rest of RAM. Next, it then initializes all services that would execute in the secure world using Secure RAM, such as the fTPM.

Once the dispatcher is loaded, the UEFI switches to the normal world. From now on, all returns back to the dispatcher are treated as explicit state switches from the normal world in response to the smc instruction. The normal world performs the rest of the UEFI startup that initializes the platform's I/O devices and boots their firmware.

On multi-core systems, every core other than core 0 is kept parked until CPU0 finishes secure world's initialization. For the remaining cores, an exception table in the monitor is set with a minimal SMC handler. This minimal handler implements the functionality required for lightweight context switches (which must be served by these cores), and returns an error for all full context switches.

Secure Clock. The TEE uses a read-only microsecond counter. Once the platform is up and running, this counter cannot be modified. However, this time source suffers from four limitations: (1) it is a 32-bit counter, and thus rolls over every 72 minutes; (2) this timer cannot be programmed to generate interrupts; (3) the counter does not increment when the platform is powered off; and (4) the counter is reset when the platform enters "deep sleep" (corresponding to LP0) because the SoC loses power. Figure 6 illustrates how the TEE updates its clock.

The TEE clock can fall behind the wall clock because it cannot detect when the counter has wrapped. Clock increments are persisted only when the TEE is scheduled to run. The normal world can (maliciously) starve the secure world preventing the clock from advancing.

Another challenge is handling the timer counter resets when the platform transitions to the LP0 sleep state. If left unhandled, this might artificially inflate the elapsed time because the TEE cannot distinguish between a counter reset due to LP0 and a counter wrap-around. Fortunately, the ARM architecture transitions to the secure world as the last step before entering LP0, and starts in the secure world on resume. When entering LP0 sleep, the TEE saves the counter value to RAM to ensure that it can restore the correct clock value on resume. This works because RAM is refreshed during LP0 sleep.

10 Providing Storage to Secure Services

The combination of encryption, the RPMB, and the hardware fuse 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:

$$SSK := KDF(HF, DK, UUID) \quad (1)$$

where KDF is a one-way derivation function, HF is the value read from the hardware fuse, 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 the 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.

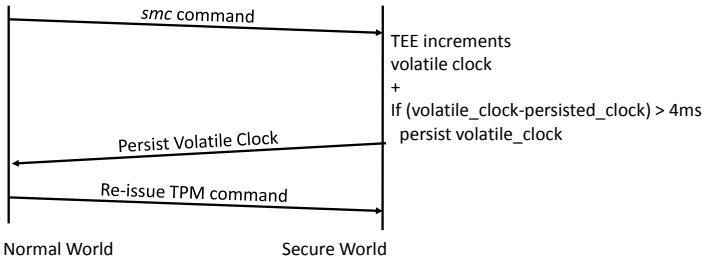

Figure 6. **fTPM clock update.**

Figure 7. **RMPB blocks.** Bit vector mechanism used for atomic updates.

10.1 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 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 non-committed) 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 write operation of the bit vector.

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 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 7 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 are planning 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

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.**

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.

11 Performance Evaluation

Our performance evaluation seeks two important questions:

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

2. What is the performance overhead of typical fTPM commands, and how do they compare to a discrete TPM chip implementation? TPM chips have notoriously slow microcontrollers [27]. In contrast, fTPM commands execute on fully-fledged ARM Cortex cores.

11.1 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 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 Used 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, a more interested reader can found all data gathered in our benchmarks in the Appendix.

- **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 Seal TPM 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 Unseal TPM 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.
- **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.

11.2 Overhead of RSA Keys Creation

Figure 8 shows the latency of a TPM create RSA-2048 keys command across all our seven devices. First, 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. Without it, creating RSA keys would have likely left the OS

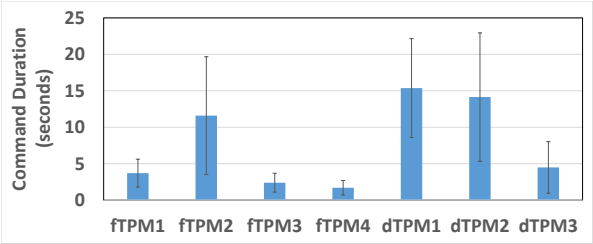

Figure 8. **Latency of create RSA-2048 keys on various fTPM and dTPM platforms.**

unstable due to remaining suspended for several seconds at a time.

Second, the performance of creating keys can be quite different across these devices. As we can see, fTPM_2 takes a much longer time than all other mobile 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 the Appendix), we believe that dTPM_3 likely searches for prime numbers in the background, even when no TPM command is issued, and maintains a cache of prime numbers.

Figure 8 also shows that the latency of creating keys is highly variable across all fTPMs and dTPMs. This large variability is due to how quickly prime numbers are found.

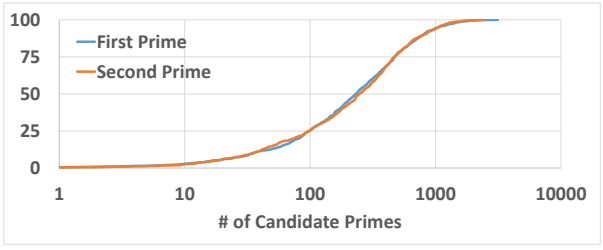

Figure 9. **Performance of searching for primes.**

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) 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 9. 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

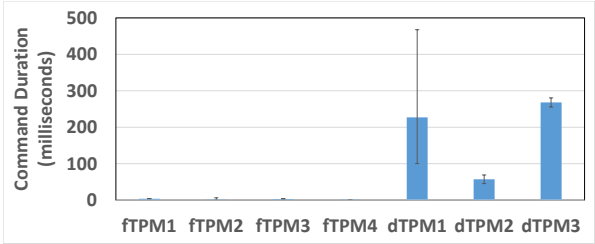

Figure 10. **Performance of TPM seal command.**

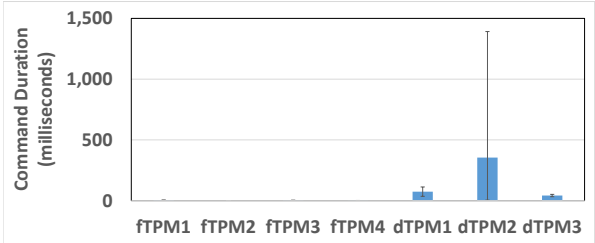

Figure 11. **Performance of TPM unseal command.**

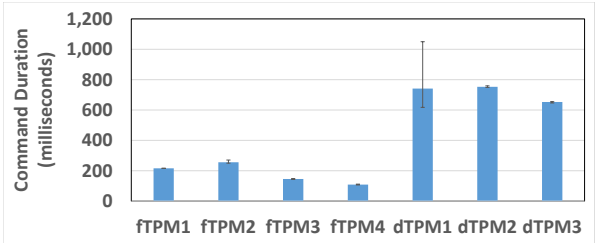

Figure 12. **Performance of TPM sign command.**

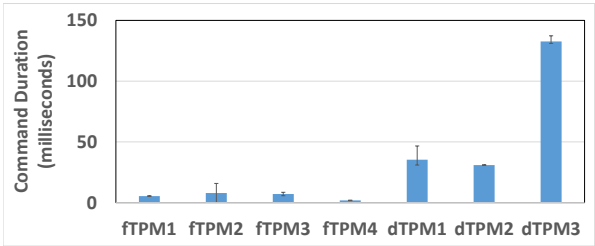

Figure 13. **Performance of TPM verify command.**

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).

11.3 Comparing fTPMs to dTPMs

Figures 10– 16 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 Appendix illustrates these vast performance improvements in even greater detail.

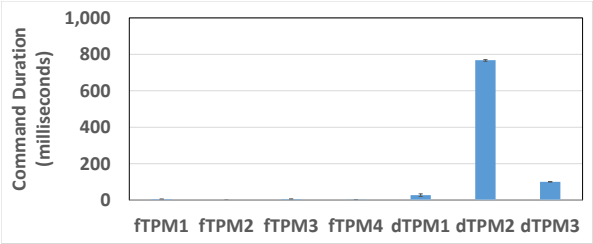

Figure 14. **Performance of TPM encrypt command.**

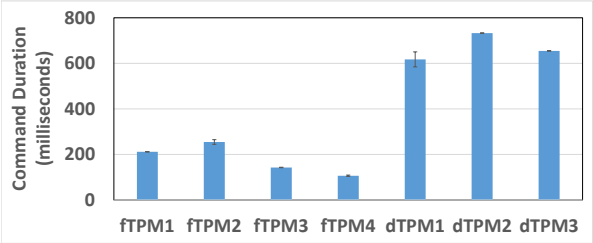

Figure 15. **Performance of TPM decrypt command.**

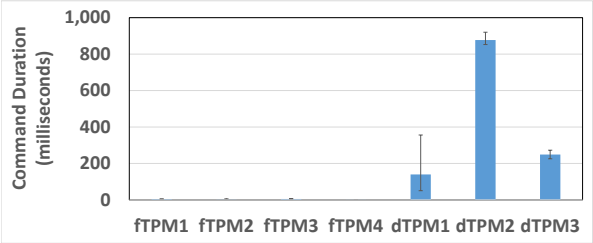

Figure 16. **Performance of TPM load command.**

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, these operations' performance are limited only by processor speeds and memory bandwidths. Furthermore, fTPMs coulbe become even faster by taking advantage of crypto accelerators. Over time, we anticipate that crypto operations would increasingly abandon the OS crypto libraries in favor of the fTPM. This provides increased security as private keys never have to leave the TrustZone's secure perimeter.

11.4 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.

12 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 [41]. 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 [18, 32], attaching a bus monitor to monitor data transfers between the CPU and system RAM [16, 12, 13], or mounting DMA attacks [6, 7, 35].

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 [21].

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 [9].

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 are 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: Side-channel attacks that exploit caches are unlikely because caches are always invalidated by hardware 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 se-

cure world.

In general, the ARM TrustZone specification takes great care to reduce the likelihood of cache-based side-channel attacks for shared resources [1].

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.

13 Looking Ahead

13.1 Intel SGX and Its Shortcomings

Intel SGX [20] is a set of extensions to the Intel processor designed to build a sandboxing mechanism for running application-level code separate from the rest of the systems. Similar to ARM TrustZone's secure world, with Intel SGX applications can create *enclaves* protected from the OS and the rest of the software running on the platform. All memory allocated to an enclave is hardware encrypted (unlike the secure world in ARM). Unlike ARM however, SGX does not offer 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 often an important stepping stone to 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 protect code running in ring 3 only. This means that the untrusted OS is left with servicing resource management tasks. This opens up a large surface for side-channel attacks. Indeed, recent work has demonstrated a number of such attacks against Intel SGX [48].

14 Related Work

The related work closest to ours is Nokia OnBoard credentials (ObC), Mobile Trusted Module (MTM), and previous implementations of earlier TPMs in software. ObC [23] 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 research effort also sought to port the .NET framework to the TrustZone [37] using techniques similar to ObC.

While the fTPM serves as the reference implementation of a firmware TPMs for ARM TrustZone, ObC is a technology proprietary to Nokia. Third-parties need to have their code signed by Nokia to take advantage of running it inside the TrustZone. In contrast, the fTPM offers TPM 2.0 primitives to any application. While TPM's primitives are less general than a full scripting language, both researchers and industry have already used TPMs in many secure systems and prototypes demonstrating its usefulness.

The Mobile Trusted Module (MTM) [42] is a specification similar to that of a TPM but aimed solely at mobile devices. Unfortunately, MTMs have never gone passed the specification stage in the Trusted Computing Group. As a result, we are unaware of any systems that made use of MTMs. If MTM were to become a reality, many of our techniques would remain relevant in building a firmware MTM.

For many years, IBM has maintained a software implementation of TPM 1.2 [19]. We are unaware of efforts to integrate this earlier implementation into commodity mobile devices.

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 [15] has also started to put together a standard for trusted runtime execution environments on various platforms, including ARM [3].

15 Conclusions

This paper presented the design and implementation of a firmware-based TPM 2.0 leveraging ARM TrustZone. The paper described the shortcomings of ARM's hardware when building practical trusted systems. Three different approaches are presented to overcome these challenges: requiring additional hardware support, making design compromises without affecting TPM's security, and slightly changing the semantics of a small number of TPM 2.0 commands. Our implementation is the reference implementation used in all ARM-based Windows mobile devices including Microsoft Surface and Windows Phones.

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] N. Asokan, J.-E. Ekberg, K. Kostianen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. Mobile Trusted Computing. *Proceedings of IEEE*, 102(1):1189–1206, 2014.

[5] A. Baumann, M. Peinado, and G. Hunt. 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] M. Becher, M. Dornseif, and C. N. Klein. FireWire - all your memory are belong to us. In *Proc. of CanSecWest Applied Security Conference*, 2005.

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

[8] C. Chen, H. Raj, S. Saroiu, and A. Wolman. 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.

[9] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proc. of 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, 2015.

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

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

[12] 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.

[13] FuturePlus System. DDR2 800 bus analysis probe. http://www.futureplus.com/download/datasheet/fs2334_ds.pdf, 2006.

[14] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. YouProve: Authenticity and Fideltiy in Mobile Sensing. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Lake District, UK, 2012.

[15] GlobalPlatform. Technical Overview. http://www.globalplatform.org/specifications.asp.

[16] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin. Reconfigurable hardware for high-security/high-performance embedded systems: The SAFES perspective. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2):144–155, 2008.

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

[18] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. of the 17th USENIX Security Symposium*, 2008.

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

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

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

[22] P. C. Kocker. 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.

[23] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala. On-board Credentials with Open Provisioning. In *Proc. of the 4th International Symposium on Information, Computer, and Communications Security (ASIA CCS)*, 2009.

[24] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. 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.

[25] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software Abstractions for Trusted Sensors. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Lake District, UK, 2012.

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

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

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

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

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

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

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

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

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

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

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

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

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

[39] S. Saroiu and A. Wolman. 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.

[40] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, , and F. B. Schneider. Logical Attestation: An Authorization Architecture For Trustworthy Computing. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

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

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

[43] 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.

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

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

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

[47] A. Wolman, S. Saroiu, and V. Bahl. Using Trusted Sensors to Monitor Patients' Habits. In *Proc. of 1st*

USENIX Workshop on Health Security and Privacy (HealthSec), Washington, DC, 2010.

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

[49] F. Zhang, J. Chen, H. Chen, and B. Zang. 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.

Appendix

This appendix presents figures containing all performance results for each device tested. The figures are placed on the following page due to formatting reasons.

Device fTPM$_1$

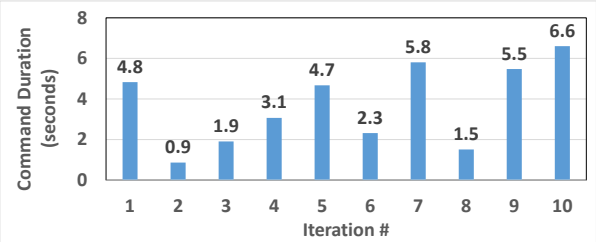

Figure 17. **TPM create RSA 2048-bit key.**

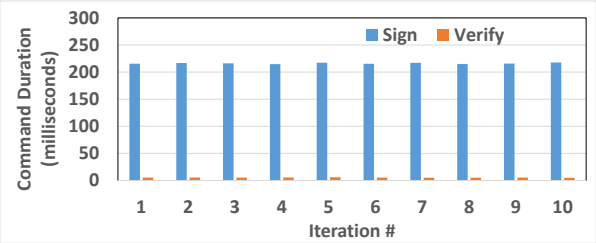

Figure 18. **TPM seal and unseal.**

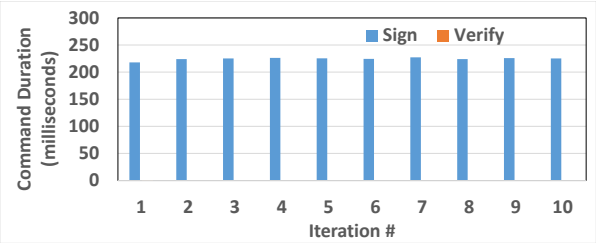

Figure 19. **TPM sign and verify.**

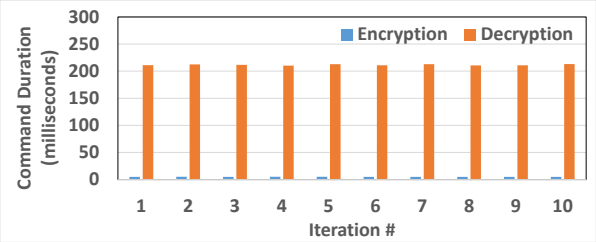

Figure 20. **TPM encryption and decryption.**

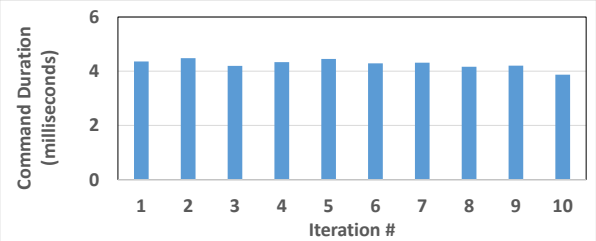

Figure 21. **TPM load.**

Device fTPM$_2$ These results were gathered with a timer with a 15 ms granularity.

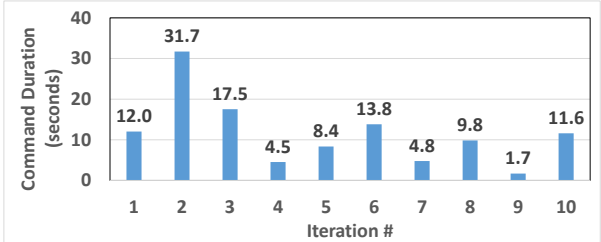

Figure 22. **TPM create RSA 2048-bit key.**

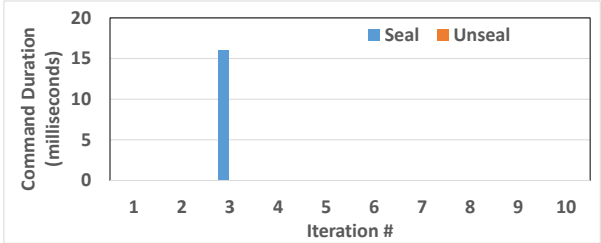

Figure 23. **TPM seal and unseal.**

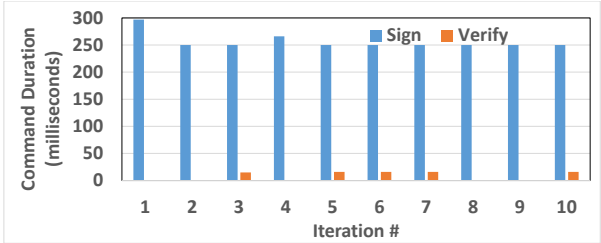

Figure 24. **TPM sign and verify.**

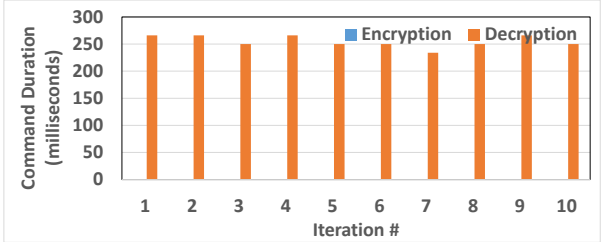

Figure 25. **TPM encryption and decryption.**

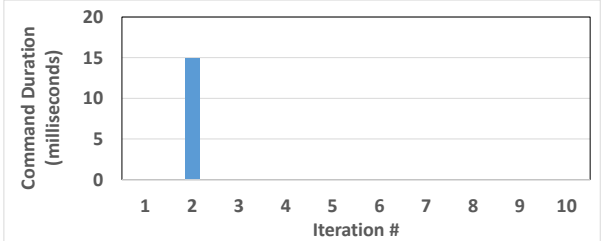

Figure 26. **TPM load.**

Device fTPM$_3$

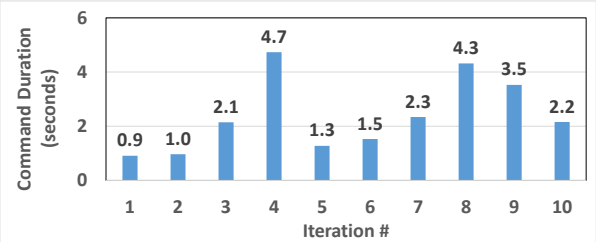

Figure 27. **TPM create RSA 2048-bit key.**

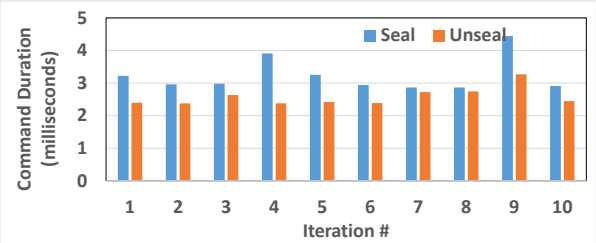

Figure 28. **TPM seal and unseal.**

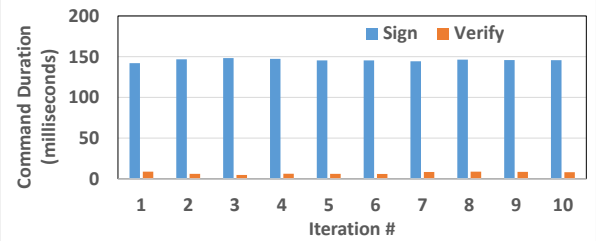

Figure 29. **TPM sign and verify.**

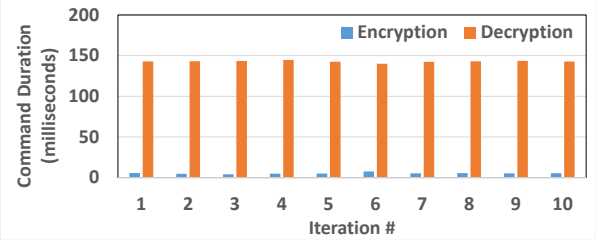

Figure 30. **TPM encryption and decryption.**

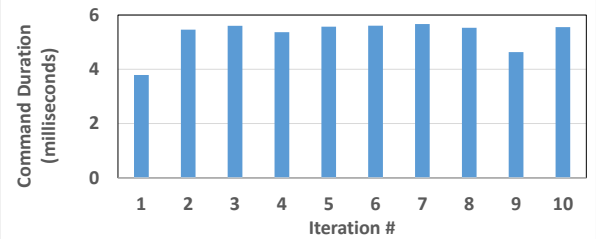

Figure 31. **TPM load.**

Device fTPM$_4$

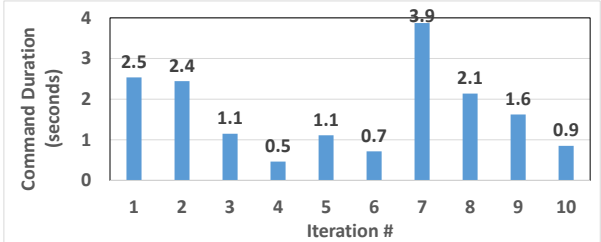

Figure 32. **TPM create RSA 2048-bit key.**

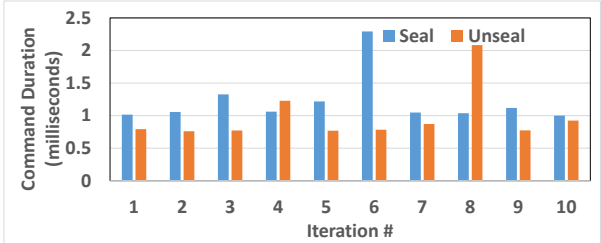

Figure 33. **TPM seal and unseal.**

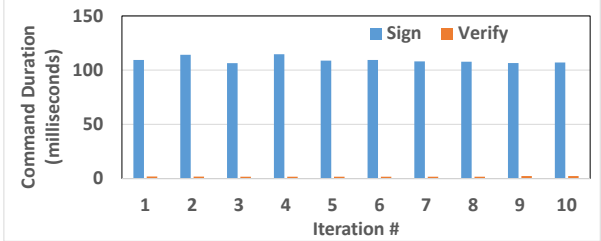

Figure 34. **TPM sign and verify.**

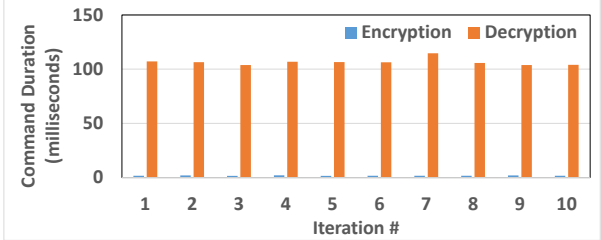

Figure 35. **TPM encryption and decryption.**

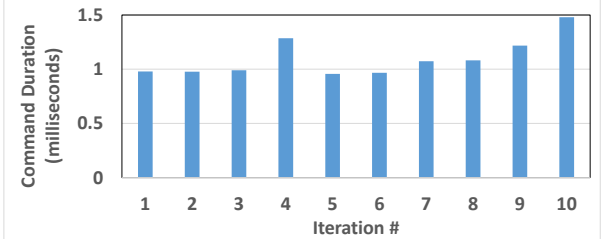

Figure 36. **TPM load.**

Device dTPM$_1$

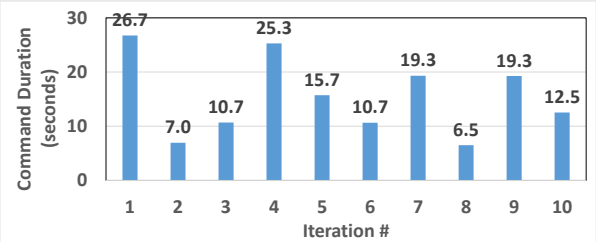

Figure 37. **TPM create RSA 2048-bit key.**

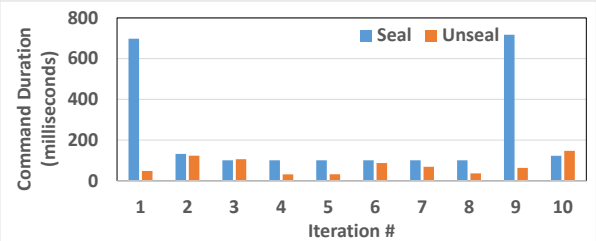

Figure 38. **TPM seal and unseal.**

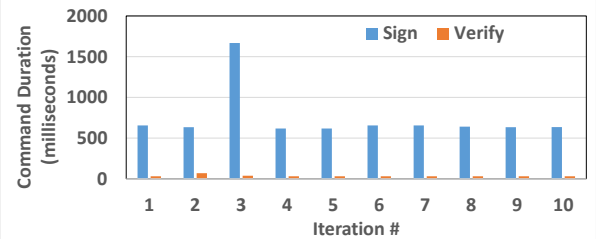

Figure 39. **TPM sign and verify.**

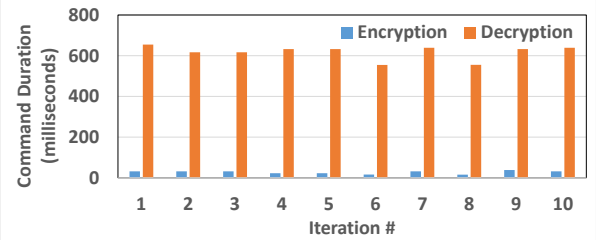

Figure 40. **TPM encryption and decryption.**

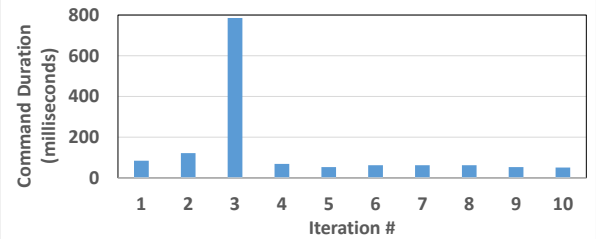

Figure 41. **TPM load.**

Device dTPM$_2$

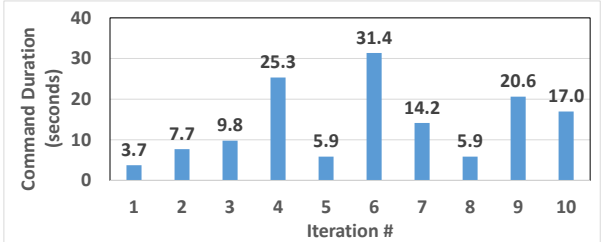

Figure 42. **TPM create RSA 2048-bit key.**

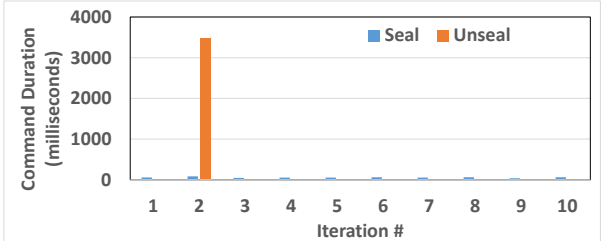

Figure 43. **TPM seal and unseal.**

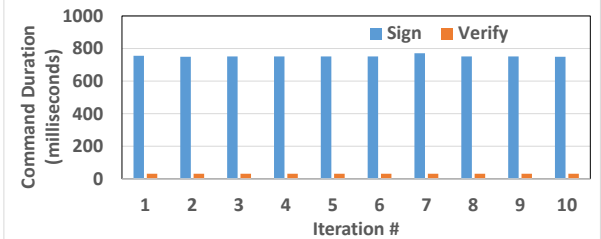

Figure 44. **TPM sign and verify.**

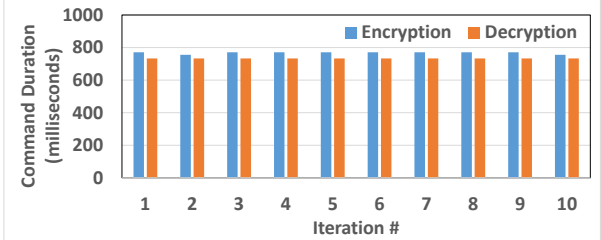

Figure 45. **TPM encryption and decryption.**

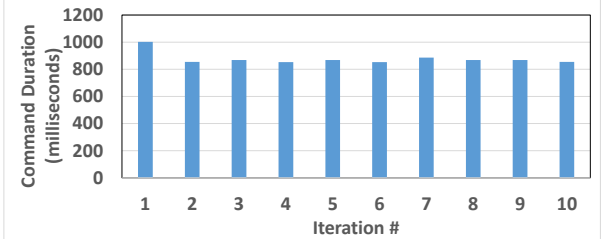

Figure 46. **TPM load.**

Device dTPM$_3$

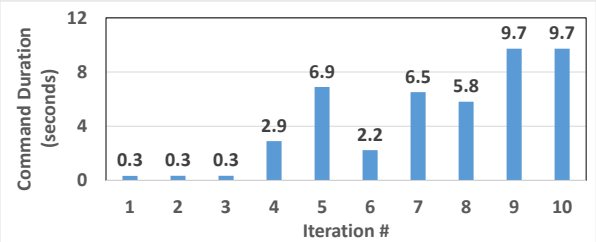

Figure 47. **TPM create RSA 2048-bit key.**

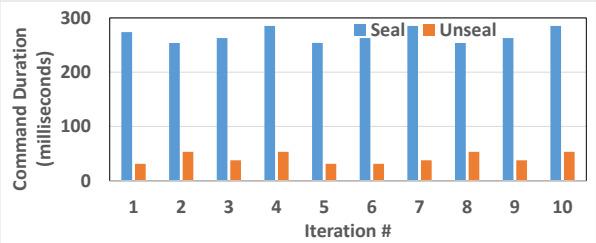

Figure 48. **TPM seal and unseal.**

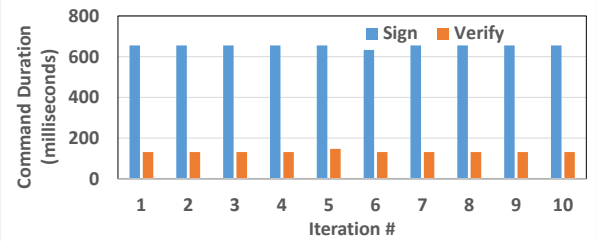

Figure 49. **TPM sign and verify.**

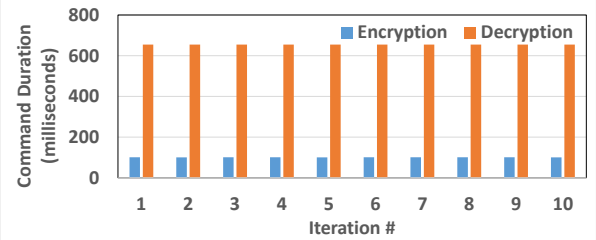

Figure 50. **TPM encryption and decryption.**

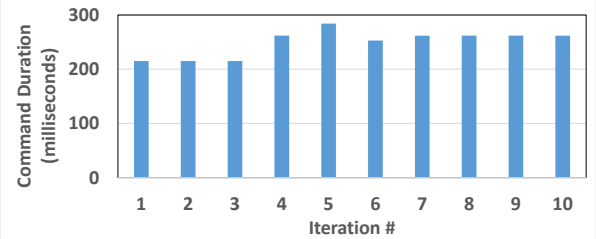

Figure 51. **TPM load.**