UNSW
AUSTRALIA

School of Computer Science and Engineering
Faculty of Engineering
The University of New South Wales

Secure Boot and TPM Attestation for seL4-Based Systems

By Nicholas Berridge-Argent

z5208292

Supervised by Ihor Kuz and Ben Leslie (Breakaway Consulting)
Assessed by Gernot Heiser

Thesis C Report — Term 3, 2021
Submitted *24th November 2021*

Thesis submitted as a requirement for the degree of
Bachelor of Engineering in Software Engineering (Honours)

Abstract

Secure boot and TPM attestation are two important technologies for validating the integrity of a platform. In this thesis, we will evaluate current approaches to validating platform integrity in seL4-based systems, and develop an approach to integrating hardware-backed secure boot and TPM attestation into seL4-based systems.

We find that TPM attestation is possible on ARM platforms, a common platform of choice for seL4-based systems, and demonstrate a system which provides access to a TPM and TPM attestation on ARM systems for seL4. We also find that there are some flaws with this approach, and conclude by proposing some designs which take advantage of seL4 to provide better security properties.

Acknowledgements

I would like to thank Ihor Kuz and Ben Leslie for their guidance as supervisors for this project, as well as Gernot Heiser for his guidance and feedback as assessor. I would also like to thank my Trustworthy Systems thesis peers and fellow students, for being available to discuss ideas with in the regular student meetings.

Finally, I would like to thank my family and Julie, for all their support during both my degree and this thesis.

Contents

Introduction

1.1 Motivation

The year is 2011 — and over 4 million Windows PCs are participating in a large peer-to-peer botnet, with their users and administrators none the wiser about it [Ars Technica 2011].

This botnet is formed with a unique piece of malware — the *Alureon* rootkit — which made itself known for its ability to infect systems by modifying their *Master Boot Record* (MBR) and running code before the operating system had a chance to start [Goodin 2010]. This allowed Alureon to easily evade anti-malware software, which often relies on the operating system to provide its functionality.

Figure 1.1 outlines the structure of traditional malware and traditional rootkits, comparing them to Alureon. Traditional malware runs alongside normal applications, making it easier to design, but easy to defend against with operating system-level security features. Rootkits embed themselves in the operating system to avoid detection, but usually run alongside the operating system and thus can be detected or prevented with some clever programming. Alureon is able to circumvent these by running before the operating system is even loaded.

Such an attack was already theorised by security researchers, but this malware and others like it highlighted the importance of protecting a computer before the operating system has a chance to. Our current best defences against this are *secure boot* and *attestation*.

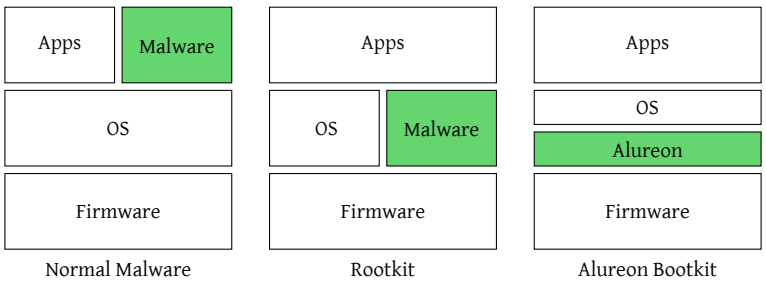


Figure 1.1: Comparing normal malware, rootkits, and the Alureon bootkit.

1.1.1 Application to seL4

seL4 is a *microkernel*, with strong security properties guaranteed by a unique design, small *Trusted Computing Base* (TCB), and comprehensive *formal verification* [Klein et. al. 2009]. It is popular in embedded systems where a high degree of security or reliability is required. However — like all security solutions — seL4 has a pre-defined scope and is not designed to protect against attacks from outside this scope. For seL4, examples of some of these out-of-scope attacks would be:

- *Physical attacks.* For example, an attacker attaching a signal analyser to the memory bus and reading memory transactions.
- *Improper setup.* For example, having an untrustworthy web server share an address space with an important storage driver.
- *Userspace vulnerabilities.* For example, running a publicly accessible database server in userspace with the password 'password'.
- *Platform attacks.* For example, an attacker modifying the boot image to be a malicious version of seL4.

This last attack vector is the one we are interested in, and is similar to the Alureon rootkit described above. Although seL4 would prevent malicious software from accessing the hardware in a privileged way, if the boot image were somehow compromised by a hardware vulnerability or by some limited physical access, then seL4-based systems would still be vulnerable to these kinds of attacks.

One possible defence would be to boot seL4 directly from *Read-Only Memory* (ROM). This would successfully defend against some platform-based attacks, however:

- There may still be ways to erase and re-program this ROM, depending on how it was made (for example, if it was *Electronically Eraseable Programmable Read-Only Memory*).
- There may be ways to boot code from other sources.
- This would make it impossible to upgrade seL4 to enable new features (such as the recently introduced MCS kernel).
- If this approach was chosen by manufacturers for other operating systems (e.g. Linux) it would be impossible to install seL4 onto these systems.

Secure boot and attestation provide a more flexible and robust defence against platform-based attacks, and address all of the points listed above.

1.2 Thesis Problem Statement

Secure boot and TPM attestation are two important technologies for validating the integrity of a platform. In this thesis, we will evaluate current approaches to validating platform integrity in seL4-based

systems, and develop an approach to integrating hardware-backed secure boot and TPM attestation into seL4-based systems. The result will be an seL4-based system protected by secure boot, with a CAmkES module which allows the userspace to access the TPM and provide local and remote attestation.

Background

First, we will need to examine some of the technologies and cryptographic primitives required to understand secure boot and attestation, then we can start to look at how secure boot and attestation work on a theoretical level and their more common implementations.

2.1 Platform Security

The *platform* of a system refers to the foundational hardware and firmware that a system runs on. Depending on the context, it may also include some portion of the system's software stack. *Platform security* refers to the security of this platform. In particular, where any weaknesses might exist in this platform, and mechanisms to protect or validate it.

Vulnerabilities in the platform could be used to compromise a system in spite of many protections which are applied at a higher level (in the case of Alureon, this was anti-malware software). Therefore, good *defence in depth* measures need to take steps to ensure that the platform is also secured.

2.1.1 Booting a System

A large responsibility of the platform is to *boot* the system. On modern systems, this will involve several stages as outlined in Figure 2.2:

1. The firmware will initialise parts of the hardware, and load some lower level drivers and the bootloader.
2. The bootloader will start to configure and load the operating system. In some systems there may be multiple stages of bootloaders.
3. The operating system will load some higher level drivers, and the rest of userspace, which the user will actually interact with.

The user of a modern system will rarely interact with the stages before the operating system. In some cases they may never even see or notice them. However, each of them presents a challenge for platform security.

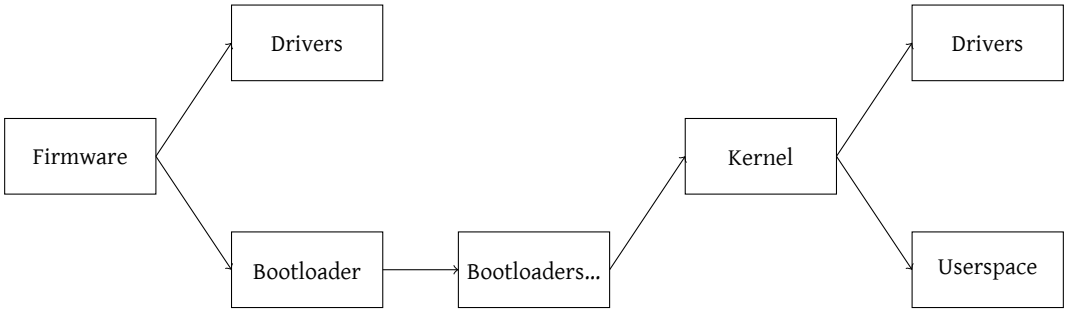


Figure 2.2: Boot Process for Modern Systems

2.1.2 Evil Maid Attack

The *evil maid attack* is a hypothetical threat model proposed by security researcher Joanna Rutkowska [Rutkowska 2009] which works based on limited physical access to a system. It describes a situation where:

1. Someone (the target) is staying at a hotel, and leaves their laptop in their room while they go out.
2. While they are gone, the evil maid enters their room and accesses their laptop.
3. The evil maid boots the laptop's firmware and installs a new piece of malware to the MBR.
4. The malware runs transparently underneath the operating system, and its only purpose is to collect passwords and encryption keys.

At this point the system is already compromised, although in the full threat model the maid returns the next day after the laptop has been used for one night and uninstalls the malware, leaving no traces.

Secure boot and *attestation* both provide defences against the evil maid attack, and similar attacks targeting the platform.

2.1.3 Secure Boot

Secure boot (also occasionally called trusted boot) is a technique for ensuring that only trusted code is ever booted on a system. To do this, it makes use of *cryptographic signatures*.

Figure 2.3 outlines the changes to the boot process with secure boot. Each stage of the boot process checks a cryptographic signature for the next stage against a list of known trusted public keys. If this signature is invalid or untrusted, that stage of the boot will refuse to boot the next stage. This forms a *chain of trust*, in which each stage of the boot has established trust in the next, up to the userspace.

But this poses a question: If a system boots with secure boot, is it secure? Secure boot only produces a binary output of success or failure, a system either boots or does not boot. If it does boot, there may still be reasons why it is not secure:

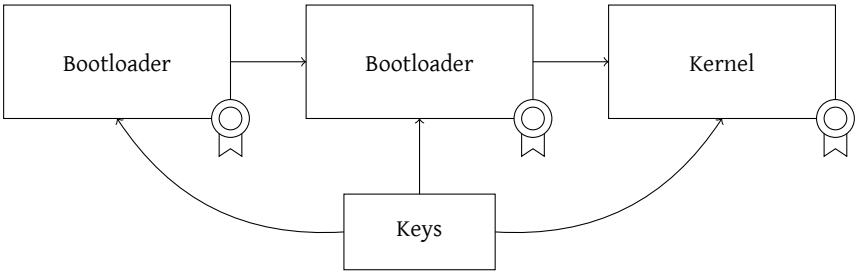


Figure 2.3: A high-level overview of secure boot.

- The private keys used for the cryptographic signatures may have been compromised.
- The system may have been set up deliberately to boot an older and insecure version of an operating system, which would still have a valid signature.
- Secure boot may have been turned off, either through a software vulnerability or some limited physical access.
- There may be a bug in the secure boot implementation itself (see *boothole* [Shkatov and Michael 2020]).

Attestation can provide a more flexible solution to platform security.

2.1.4 Attestation

Attestation (also occasionally called *measured boot*) is a technique for securely measuring the state of a system's platform and reporting that to a trusted third party (local or remote). To do this, it makes use of *cryptographic hashes*, *asymmetric encryption* and some *secure hardware*.

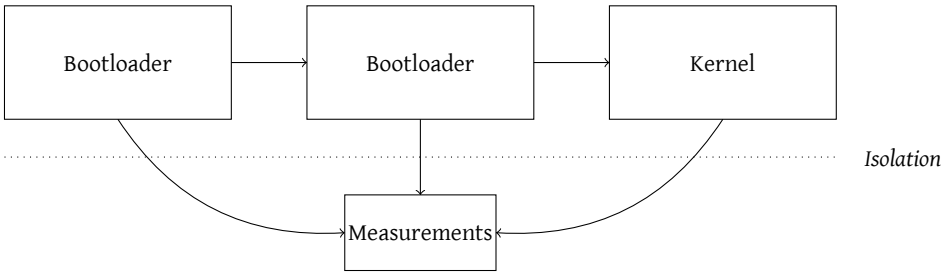


Figure 2.4: A high-level overview of attestation.

Figure 2.4 outlines the boot process with attestation. Each stage of the boot process computes a cryptographic hash of the code and the configuration for the next stage. These are then stored in *secure storage* that is backed by either hardware or firmware to prevent certain kinds of modification. This also forms a chain of trust as with secure boot, but also produces a trusted *measurement* of the state of the system. This measurement can then be sent via asymmetric encryption to a trusted third party to be verified against some known secure values.

This is ideal compared to secure boot because even if an attacker could somehow disable these security measures, they would not be able to fake the produced measurements, and so would not be able to properly attest the system to a third party verifier.

However, unlike secure boot which can be done entirely within the system it protects and can stop an insecure system from running in the first place, attestation does require a third party to verify it and cannot necessarily stop an insecure system from booting.

2.2 Cryptographic Primitives

In order to fully understand how secure boot and attestation work and how they are implemented on most systems, we need to understand a few of the cryptographic primitives being used.

2.2.1 Cryptographic Hashes

A *cryptographic hash function* is a function $H(x)$ which takes a sequence of bytes x and produces another fixed-length sequence of bytes called a hash (also called a digest) which "summarises" the input bytes. Ideal cryptographic hash functions have two properties which we are interested in:

- Given only the hash of a sequence of bytes $H(x)$, it should be impossible to find the original sequence of bytes x except from by using brute force. In this way, the hash function is said to be a *one-way function*.
- Given a sequence of bytes x and its hash $H(x)$, it should be impossible to find another sequence of bytes y such that $H(x) = H(y)$. In this case x and y are said to be colliding. In practice due to the pigeon hole principle each sequence of bytes has an infinite number of collisions (for instance in SHA256, there are "only" 2^256 possible hash values, but many more possible input sequences), but it should be computationally difficult to find a useful collision for a given sequence of bytes.

The properties of hash functions are demonstrated in Figure 2.5. An example of a modern, secure cryptographic hash function is *SHA256*.

2.2.2 Asymmetric Encryption

Asymmetric encryption is a form of encryption where encrypting and decrypting require different keys. In particular, we have two keys: a *public key* (usually denoted K^+) which is shared publicly, and a *private key* (usually denoted K^-) which is kept secret. These two keys are generated in pairs, and can only be used as that particular pair. With these two keys, we have the following rules:

- If some bytes are encrypted using a public key, they can only be decrypted using the corresponding private key.
- Similarly, if some bytes are encrypted using a private key, they can only be decrypted using the corresponding public key.

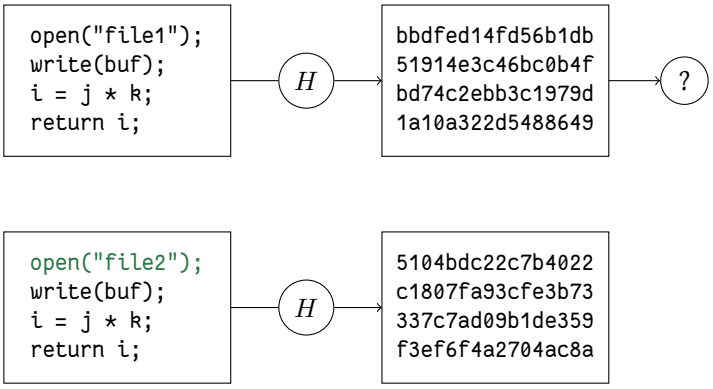


Figure 2.5: Properties of cryptographic hashes. *Top:* One we have a hash, it is not possible to find the original data. *Bottom:* If the input data changes, the resulting hash changes significantly.

Figure 2.6: Properties of asymmetric encryption. Note the switched use of public and private keys.

As with other encryption, it should be impossible to decrypt data without having the required key (public or private).

The properties of asymmetric encryption are demonstrated in Figure 2.6. An example of a modern, secure asymmetric encryption algorithm is *RSA*.

2.2.3 Cryptographic Signatures

Cryptographic signatures make use of both cryptographic hashes and asymmetric encryption to be able to "sign" a blob of data, ensuring that the data is known to be correct and valid by some trusted third party.

Suppose that person A produces a sequence of bytes they wish to sign. They take that sequence of bytes, compute the cryptographic hash of that sequence of bytes, and encrypt it with their private key. This encrypted cryptographic hash is the *signature* and is distributed with that sequence of bytes.

Person B now wants to verify the integrity of that sequence of bytes. They take the sequence of bytes

which they received and compute the same cryptographic hash. Then, they take the signature that was distributed with that sequence of bytes, separately request person A's public key, and decrypt the signature. If the cryptographic hashes match, we know that sequence of bytes was definitely produced by person A and was not modified.

Without a copy of person A's private key, it would not be possible to modify the sequence of bytes *and* produce a signature which is valid when using person A's public key.

This does introduce some problems:

- We still need to trust person A to give us the correct sequence of bytes. This is particularly important when person A is only responsible for signing bytes, and not creating them.
- If an attacker steals person A's private key, they can produce valid signatures for any sequence of bytes.
- We need a way to securely obtain person A's public key. If they are both sent over the same communications channel, an attacker could replace the sequence of bytes, signature *and* public key with their own.

2.2.4 Blockchain

These days, when we think of the word "blockchain", we think of cryptocurrency. However, a *blockchain* is actually a simpler concept than that, it is a series of blocks of data each linked using a *cryptographic hash*.

Each *block* starts with the hash of the previous block, meaning that modifying any one block will cause the rest of the blockchain to also be modified. By checking the hash and value of the last block, we effectively have an ordered cryptographic *summary* of an entire sequence of blocks.

In some implementations, attestation uses a similar construct to a blockchain to measure the state of the platform. The properties of blockchains are demonstrated in Figure 2.7.

2.3 UEFI Secure Boot

The *Unified Extensible Firmware Interface* (UEFI) is a common firmware interface for many x86 systems. Of particular interest to us, it standardises the most common implementation of secure boot, using cryptographic signatures [UEFI 2020]. This implementation of secure boot is shown in more depth in Figure 2.8.

In this implementation of secure boot, any bootable or low-level code (bootloaders, drivers, the operating system) needs to include a trusted signature in order to be booted. To solve the public key distribution problem, these signatures are verified against keys which are pre-installed onto the system at the manufacturing time. Most systems include a copy of *Microsoft's* public key, but some systems

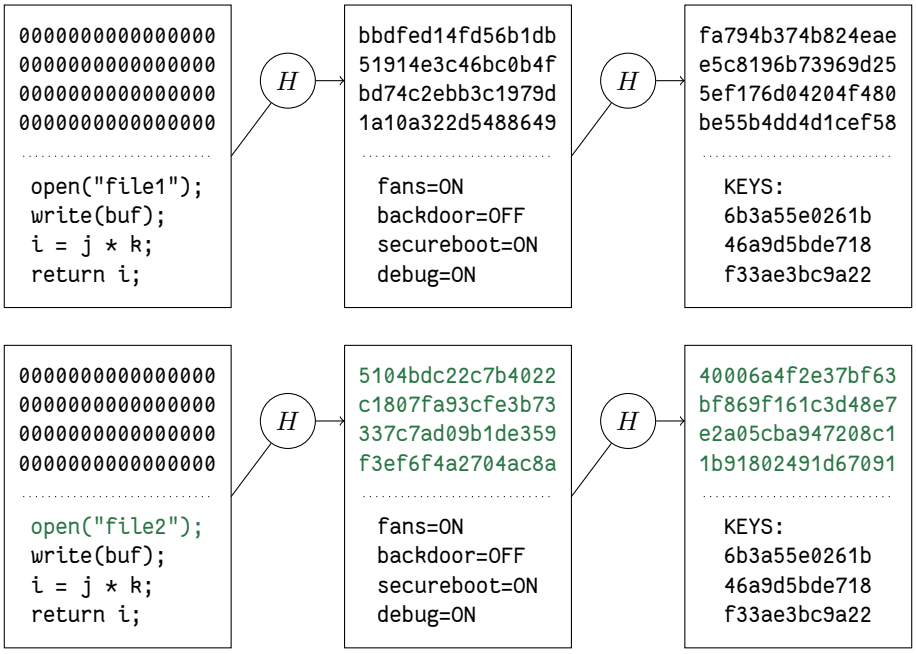


Figure 2.7: Properties of blockchains. If an earlier input changes, the results are seen in the final hash value.

allow the end users to *enrol* their own extra public keys under certain circumstances.

2.4 Trusted Platform Module

Attestation requires some form of secure hardware or firmware to work properly. On larger systems, this comes from a dedicated *trusted platform module* (TPM) such as the one shown in . A TPM is a module in hardware or firmware which implements an interface standardised by the *trusted computing group* (TCG) as part of two separate specifications: TPM 1.2 and TPM 2.0 [TCG 2019].

A TPM might provide any number of useful functions:

- A secure clock, secure counter, and / or secure random number source, all of which can be used to prevent replay attacks in secure protocols.
- Secure encryption and decryption functionality, so that cryptography can be performed separately from the rest of the system.
- Secure key storage, where private keys may be set up to never leave the TPM, or may be set up to only be used if certain conditions are met.

The feature we are most interested in are the *platform configuration registers* (PCRs).

Embedded systems may not be able to justify the cost of a discrete hardware TPM and often have their

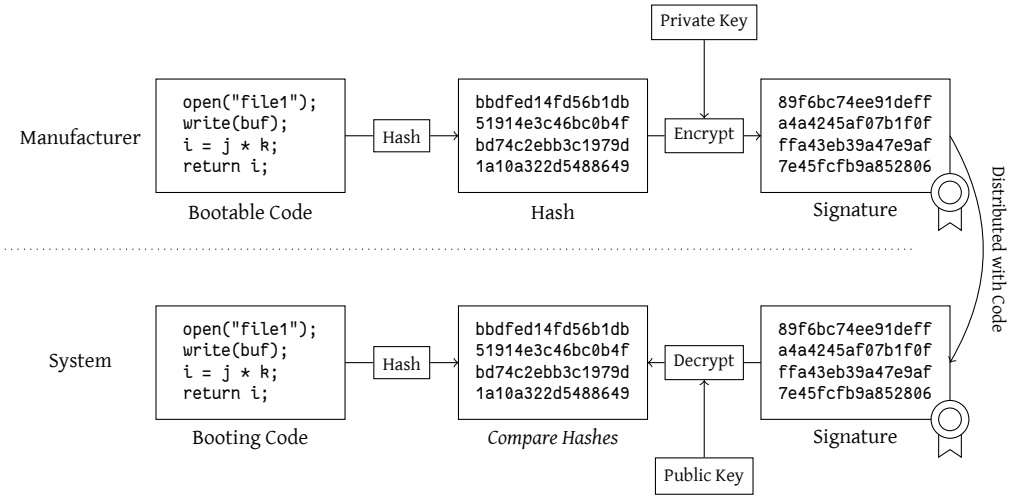


Figure 2.8: Implementation of secure boot in UEFI. Implementation in other systems will be similar.

own security features implemented. However, there are ways to implement a firmware TPM on some embedded systems, which we will look at in section 3.1.

2.4.1 Platform Configuration Registers

The *platform configuration registers* (PCRs) are a special kind of secure storage available on a TPM. In particular, they can be read at any time, but can only be modified using a special Extend operation:

$$\text{Extend}\,(P, x)\ :\ P := H(P \parallel x)$$

This forms what is effectively a *blockchain* within a given PCR. If one PCR P has Extend called with values x_1, x_2 and x_3, the resulting value is:

$$P = H(H(H(x_1) \parallel x_2) \parallel x_3)$$

If any one of the x_i are changed, each of the hashes would change, resulting in a different final value for the PCR once it gets read.

A TPM normally has several PCRs, each for the code and configuration elements of a different stage in the boot process. Some are also typically reserved for future use, or available to be used by other userspace processes. Table 2.1 shows an example PCR allocation for 24 PCRs. Not all TPMs will have the same number of PCRs, and the allocation may vary depending on the exact platform and software stack.

Although specific PCR values imply that specific events took place during the boot process, the PCR

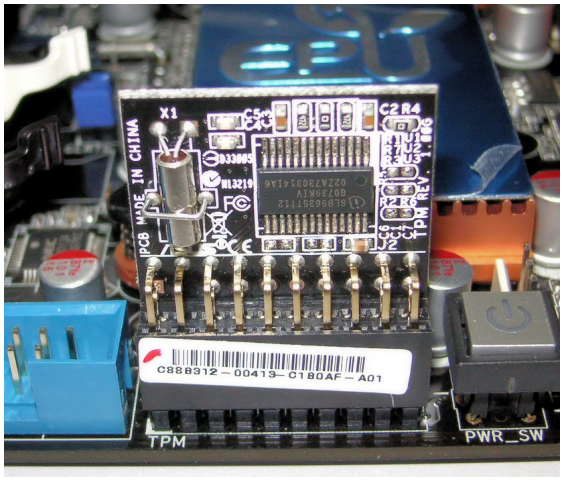


Figure 2.9: A discrete hardware TPM installed on a PC motherboard. This particular TPM appears to be a SLB 9635 TT 1.2 from Infineon, implementing the older TPM 1.2 standard. Image by Wikimedia user *FxJ*. Source: `https://en.wikipedia.org/wiki/Trusted_Platform_Module#/media/File:TPM_Asus.jpg`

values do not reveal the whole boot process themselves. In order to make understanding the boot process easier, TPMs may also include a slightly less secured but more verbose *event log* which lists all events that are used to extend the PCRs. In this way, if the PCR values are incorrect or unexpected, it is possible to see why.

Another critical detail about the PCRs is that the values in them are dependent on the order they are written. For example, if a system once loads x_1, x_2 and x_3 in that order, and then later loads x_1, x_3 and x_2 in that order, the PCR values would not match since:

$$H(H(H(x_1) \parallel x_2) \parallel x_3) \neq H(H(H(x_1) \parallel x_3) \parallel x_2)$$

Because of this, great care needs to be taken to initialise the system in a pre-defined order, and no steps can be parallelised.

2.4.2 TPM Attestation

TPM attestation is an implementation of attestation that uses the hardware / firmware security features provided by the TPM.

Starting at a small, trusted piece of software referred to as the *Core Root of Trust Measurement* (CRTM), each stage of the boot will calculate a cryptographic hash of the code and the configuration of the next stages of the boot process, and will store these hashes in a PCR.

Once the system has booted into userspace, the PCR values contain a cryptographic summary of the

PCR Numbers	Hashed Contents
0 & 1	BIOS & BIOS configuration
2 & 3	Low-level firmware & associated configuration
4 & 5	MBR & MBR configuration
6	State transitions & wake events
7	Manufacturer specific measurements
8–15	Operating-system specific measurements
16	Debugging information
10	Linux kernel
11	Linux `initramfs`
12	Linux kernel command line
13	GRUB commands & configuration
14	Secure boot shim keys & state
15	Unused
16	Debug measurements
17–22	Unused
23	Application specific measurements

Table 2.1: An example PCR allocation for Linux on an x86 PC, from [Arthur et. al. 2015].

state of the platform up to that point. To ensure that the correct values of the PCRs can be read by the third party verifier, the TPM first encrypts them with a private key that is stored securely inside the TPM. The manufacturer of the TPM publishes the corresponding public key, and the third party verifier can then decrypt the PCR values on a trusted system and check them against known good PCR values.

An attacker cannot modify part of the boot process, as an earlier part of the boot process would extend a PCR with a different value, which would cause the set of PCR values to differ. An attacker also cannot just modify the operating system's TPM driver to return fake PCR values, since the PCR values are encrypted first using an inaccessible private key. The TPM's secure clock / counter / random number source can also be used here to prevent replay attacks.

2.4.3 Remote Attestation

Remote attestation refers to the specific variant of TPM attestation where the trusted third party is another computer accessible via. the network. The encrypted PCR values are then sent to this remote verifier, which then decrypts them and verifies they are correct.

The remote verifier can then use this information in a number of ways. For example, if the PCR values are incorrect, it could [Garrett 2019]:

- Prevent that system from accessing the network, or sensitive network resources such as security cameras or network attached storage.
- Prevent that system from *being* accessed via. the network, to stop it from impersonating a trusted host on the network.

- Notify a system administrator.

However, notably, it cannot stop the system itself from starting unless it has control over that system's power supply. It also cannot do any of these things unless there is a connection between it and the system being verified.

Figure 2.10 shows an overview of how remote attestation works using a TPM. A more detailed description including the precise commands is given later in subsection 2.4.6.

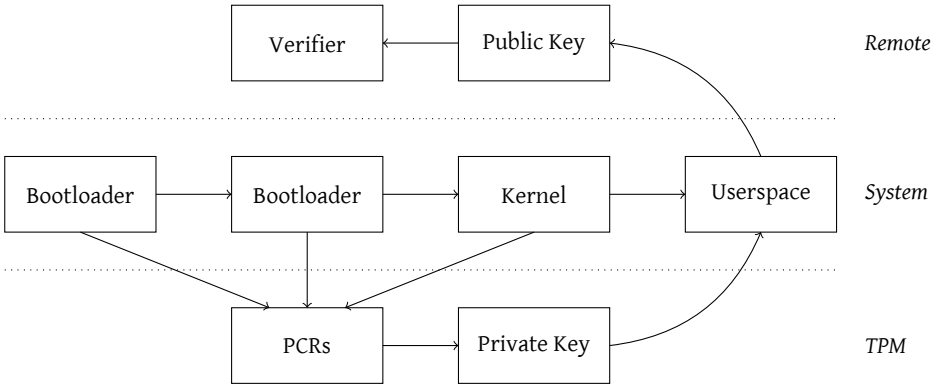


Figure 2.10: A high-level overview of remote attestation.

2.4.4 Local Attestation

Local attestation is also possible using a TPM, in which the trusted third party is a human operator with physical access to the computer. This works slightly differently to remote attestation and makes use of a TPM's *data sealing* functionality.

A TPM can be asked to keep encrypted data and the corresponding keys in secure storage, and will then only decrypt and provide this data to the userspace if the PCR values match a known good value that was set when the data was originally stored. This process is known as *sealing* data.

Figure 2.11 gives a high-level overview of how local attestation works. Local attestation can be implemented by sealing a small passphrase which can be manually inspected inside the TPM, as is done in [Constable et. al. 2018]. When the system boots, the system will present the passphrase to the user and ask them to verify that the passphrase is correct. If any parts of the boot process have been modified, the TPM will not return the passphrase to the userspace, and the system cannot present the correct passphrase.

This differs from remote attestation in that it requires a human operator to be physically present when the system boots, but does not require any kind of remote access to a third party verifier. This would make it ideal for slightly different use cases, such as air-gapped systems.

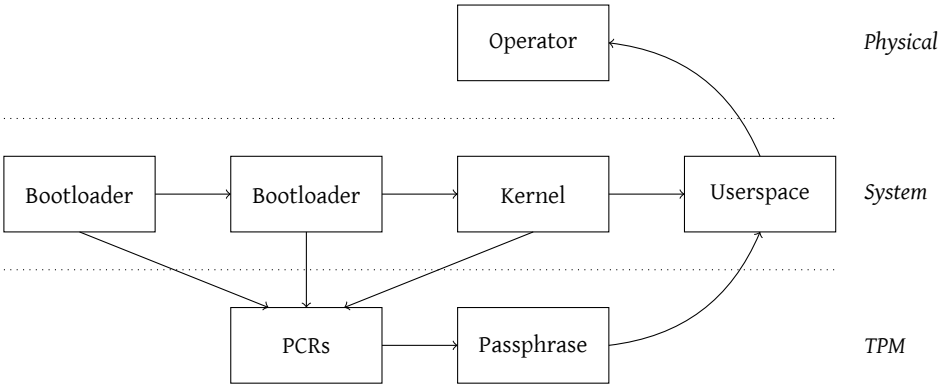


Figure 2.11: A high-level overview of local attestation.

2.4.5 Storage Hierarchy

To implement any kind of attestation, the TPM needs to store an asymmetric *attestation key*, which is used to produce the signatures described above. TPMs are lightweight devices, and have a limited amount of memory and non-volatile storage. To effectively store keys, TPMs rely on a *storage hierarchy* in which keys are derived from other keys.

To begin with, the TPM stores only a *primary seed* for each hierarchy. This is a large securely generated random number which never leaves the TPM and never changes. The primary seed we are interested in is the primary seed for the *storage hierarchy*, or the *storage primary seed*. Other hierarchies (endorsement and platform hierarchies) exist for other TPM use cases.

The storage primary seed is used to generate an asymmetric *storage key*. Since the storage key is generated by the fixed storage primary seed, the storage key always has the *same* public and private components and can be considered to be effectively stored inside the TPM.

The storage key is used to securely store and persist objects outside the TPM. When an asymmetric key is generated in the storage hierarchy of the TPM, the private component is encrypted using the storage key and returned to the system. In this way, the system cannot read the private component of the generated key, since it does not have the storage key. When the TPM is restarted, the storage key is regenerated with the same values, and the system can reload the same generated key.

The attestation key is generated inside the storage hierarchy during provisioning, and must be provided to the TPM whenever attestation is required. If a compromised system chooses not to load the correct attestation key, then the TPM cannot produce a valid attestation. If the attestation key is somehow lost, then a new one can be generated, but the verifier will also need an updated copy of the public key.

Because TPMs also support other more general cryptographic functions, it is also possible to restrict generated keys. In the case of an attestation key, it is suggested to:

- Restrict it to *only producing signatures*, to avoid it being used for general enryption / decryption.

- Restrict it to *only sign TPM generated digests*. Otherwise, an attacker could simply request the TPM to use the attestation key to sign their own, forged attestation.

2.4.6 TPM Commands

In each of the TPM standards set out by the TCG, usage of the TPM is done entirely through commands and responses. These commands can initialise the TPM, create and manage objects stored on the TPM, and perform operations on the TPM such as generating an attestation.

As far as the standard attestation process is concerned, there are eight commands we are interested in [TCG 2019]:

- `TPM2_Startup` initialises the TPM and must be called before any TPM usage.
- `TPM2_Shutdown` cleanly shuts down the TPM, optionally preserving some state. This can be used to shut down the TPM between boot stages while preserving PCR values.
- `TPM2_CreatePrimary` creates an object directly from a primary seed. For attestation, we can use this to generate the storage key.
- `TPM2_Create` creates an object inside a hierarchy and — where appropriate — returns the object data. For attestation ,we can use this to generate the attestation key during provisioning.
- `TPM2_Load` loads an object into a hierarchy. For attestation, we can use this to load the generated attestation key during normal operation.
- `TPM2_PCR_Extend` extends a PCR with a digest.
- `TPM2_PCR_Read` reads the value of a PCR.
- `TPM2_Quote` produces a signed 'quote' for some information on the TPM. For attestation, this is a quote of the PCR values, and is the value we are interested in.

Although `TPM2_PCR_Read` is not strictly required for attestation, it provides a significant quality-of-life improvement to the attestation process. The quote structure returned by `TPM2_Quote` contains a digest of *all* selected PCRs. If these values are wrong, it can be difficult to tell which of the PCRs has changed.

Each of these commands have several bytes of parameters and several bytes of information in their responses, which must be marshalled and unmarshalled by any software using the TPM. Figure 2.12 gives an example of one of these structures, and Table 2.2 shows the result of unmarshalling this structure.

```
ff544347801800220000bd31f2da8ab07884d351fe03b49384d5e30626ff03253
a098cb7331b1e305c1d900000000000003f131f0aa6945c32517169e01484e3f
b5d3db225500000001000b0301000000202e0b4afe025424000e0b7edb8ecd9b
f1a40e5609078c5eaeb7c334fb9d36ddc9
```

Figure 2.12: Example of the raw bytes in a quote response.

Value	Explanation
`ff544347`	Magic signature (`\xffTCG`)
`8018`	Quote type (PCR Quote)
`0022 .. c1d9`	Name (digest) of signing key
`0000`	Extra data (none)
`0000000003f131f0`	Clock value
`aa6945c3`	Reset counter
`2517169e`	Restart counter
`01`	Safe flag (the clock value is safe)
`484e3fb5d3db2255`	Firmware version identifier
`00000001`	PCR selection count (one)
`000b`	PCR digest algorithm (SHA256)
`03010000`	Select PCR 0 only
`0020 .. ddc9`	Digest of selected PCRs

Table 2.2: Unmarshalling the quote response in Figure 2.12. The entire quote structure is signed, and the signature follows in the full response.

With these commands, we can outline a concrete process for remote attestation, as adapted from [Chiang 2021]. Parts of this process are shown in Figure 2.13. Firstly, during provisioning:

1. Use `TPM2_Startup` to initialise the TPM.
2. Use `TPM2_CreatePrimary` with the storage hierarchy to initialise the storage key.
3. Use `TPM2_Create` to create the attestation key in the storage hierarchy. Save the returned public and encrypted private components.

Then, during normal operation of the TPM:

1. Use `TPM2_Startup` to initialise the TPM.
2. Use `TPM2_PCR_Extend` several times to measure any required data.
3. Use `TPM2_CreatePrimary` with the storage hierarchy to initialise the storage key.
4. Use `TPM2_Load` to load the attestation key's public and encrypted private components.
5. Use `TPM2_PCR_Read` to read the measured PCR values (optional).
6. Use `TPM2_Quote` to generated a signed quote of the PCR values.
7. Repeat `TPM2_Quote` as many times as required to prevent replay attacks.

In practice, systems will continually generate new quotes. This helps to detect unexpected resets and changes in the running software, as well as helps to prevent against replay attacks.

Local attestation uses a smaller set of commands, but can still be done using only a TPM. `TPM2_Create` can be used to create the sealed 'passphrase' in the storage hierarchy, and `TPM2_Unseal` can be used to

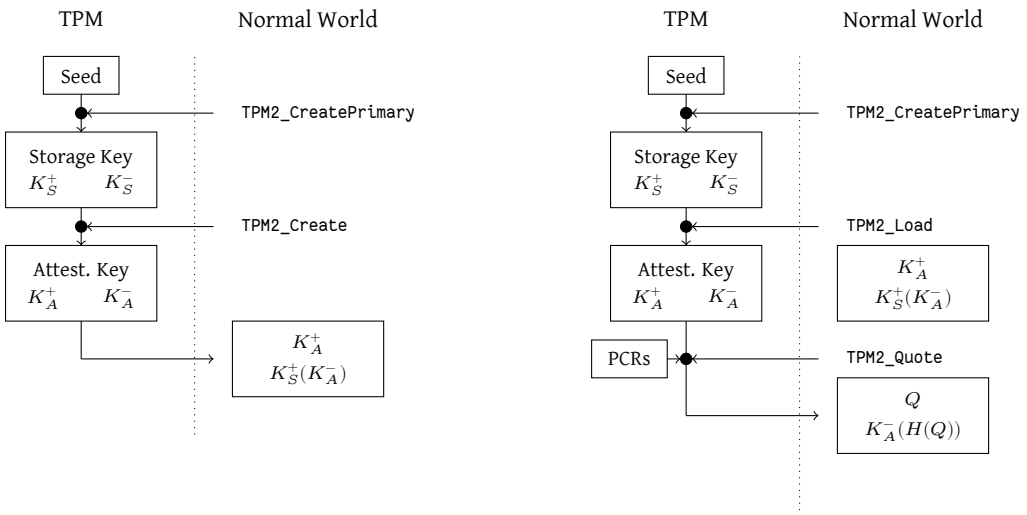


Figure 2.13: TPM command sequence for provisioning (left) and attestation (right).

unseal the passphrase and provide an attestation.

2.5 Platform Support

Secure boot and TPM attestion are both backed by hardware and firmware security features. Thus, their support depends on the platform being used. We will consider mainly x86 and ARM systems as a whole, since their features supported are usually similar.

2.5.1 x86 Systems

x86 systems are typically large desktop, laptop or server machines running powerful Intel and AMD processors. seL4 can be used on these systems to provide a secure enclave while virtualising Linux-based operating systems, but this is a less common use case.

Most x86 systems will be running on a UEFI platform, which will therefore support secure boot. However, not all x86 systems will allow users to enrol their own secure boot keys. This presents a problem since often the only key installed by default is Microsoft's secure boot public key, and Microsoft will refuse to sign GPL licenced code [Tremblay 2021].

Linux is still able to boot on some of these systems using a *UEFI secure boot shim* [Debian 2021] — a small piece of code signed by Microsoft which uses another public key to verify the bootloader and Linux image.

In some cases, secure boot may need to be turned off entirely to boot other code (i.e. code not signed by Microsoft) on these systems, which completely disables the protections of secure boot for non-Windows users.

Windows 10 also requires systems to have a TPM installed in order to boot. This means that most modern x86 systems will include some form or firmware or hardware TPM in order to be able to boot Windows 10.

2.5.2 ARM Systems

ARM systems are typically low-power embedded or mobile devices that could be running processors or *system on chips* (SoCs) from a wide range of manufacturers that have licenced ARM's IP. seL4 is most commonly used on these systems, as these systems are most commonly used for embedded applications.

Modern ARM processors include *ARM TrustZone*, a firmware supported *Trusted Execution Environment* (TEE) that provides a variety of security features. The specification for TrustZone is very minimal and only specifies the trusted execution environment — this on its own is not enough to implement secure boot or TPM attestation. However, most SoC manufacturers will include secure storage and either a secure clock, secure random number generator, or secure counter. These features can be used to implement secure boot and attestation.

In particular, in order to run Windows 10 on ARM devices (such as the Microsoft Surface), Microsoft has implemented a firmware TPM entirely on ARM TrustZone, with a minimal amount of extra hardware requirements. This is explored later in section 3.1.

2.5.3 Other Platforms

Other up and coming platforms for seL4 are based on the RISC-V architecture, which shows great promise as a high-performance, low-power, open source processor architecture. Unfortunately since RISC-V is still fairly new compared to x86 and ARM, support for secure boot and attestation are still under active development.

In future it might be possible to revise this thesis and extend the support to RISC-V, as seL4 becomes more closely associated with RISC-V.

2.6 seL4 and Microkernels

Operating systems can be categorised based on the structure of their *kernel* — the portion of code that runs at a privileged execution level and interfaces directly with hardware. Most operating systems currently use either *monolithic kernels* (e.g. Linux) that encompass almost all of the operating system's functionality, or *hybrid kernels* (e.g. Darwin, Windows NT) that export some of the operating system's functionality to userspace.

Microkernels offer a different design approach, by reducing the size of the kernel as much as possible, and providing most of the operating system's functionality in userspace. This gives different performance and design characteristics, but allows for greater security and is ideal for embedded systems.

seL4 is a *third generation* microkernel based on the popular L4 family of microkernels. It is currently the fastest microkernel, and also offers strong security guarantees by means of *formal verification*. In 2009 after a great research effort, the seL4 microkernel was formally verified [Klein et. al. 2009], this includes:

- A *functional correctness* proof that proves the implementation of seL4 is correct to a higher-level abstract specification.
- A proof of certain properties on top of this high-level specification, to ensure that certain performance and security guarantees are met.
- A proof of correct *binary translation* that ensures the produced machine code matches the behaviour modelled by the implementation.

The exact scope of the verification depends on the platform being used, but all platforms can benefit from seL4's design for security and performance.

These features make seL4 ideal for embedded applications where a very high level of security is required. As such, these systems would be a great target for secure boot and attestation to be used with. The task of providing this level of platform security to seL4 is still an active area of research.

2.6.1 Capabilities

seL4 uses *capability-based access control* [Trustworthy Systems 2020]. This means that access to objects provided by the kernel is mediated through *capabilities* to those objects. Any thread has an associated *capability space* (CSpace), and each of the capabilities has an associated set of access rights (read, write, grant and grant reply). Capabilities can also be copied, or *minted / mutated* into copies with slightly weaker access rights. These access rights have different meanings, depending on the kind of object the capability refers to.

Threads in userspace ask the kernel to act on their behalf by *invoking* a capability. One capability may support a number of different operations, which can be selected using parameters to the same all-powerful `seL4_Call` system call. The operations available also depend on the kind of object the capability refers to.

One example might be a *page* object. These are architecture-dependent, but exist in some form on every architecture to enable memory mapping. Read permissions on a page capability grant the thread the ability to map the page as readable, write permissions grant the thread the ability to map it as writeable. Invoking the page capability allows the thread to map or unmap the page.

2.6.2 CAmkES

The *Component Architecture for microkernel-based Embedded Systems* is a framework which is designed to tackle the complexity of building modular systems on top of seL4. It allows developers to describe a sys-

tem as a set of components and component interfaces in a high-level language (the *Architecture Description Language* (ADL)) and then automatically generates the infrastructure to connect these components with seL4 primitives (capabilities, communication endpoints, etc.).

This means that any userspace modules intended to be used for seL4-based systems would do well to be defined as CAmkES components, as they can be more easily integrated into existing systems.

2.6.3 seL4 Boot Process

How seL4 boots depends largely on the platform. However, the common theme is that the seL4 microkernel, the *root server* image (that is, the first userspace thread, which may load more), and a special program called the *ELF loader*; are built into one large boot image. The system booting seL4 initially boots the ELF loader, which loads seL4 and the root server into memory and jumps into seL4. seL4 then initialises itself and starts the root server.

Figure 2.14 outlines the boot process for seL4. On ARM systems, the ELF loader is usually started through U-Boot. U-Boot is a popular open source boot loader for embedded systems, providing drivers for anything necessary to boot the system. This could include power drivers (for system initialisation), storage drivers (to load images), network drivers (to pull images from the network), and even TPM drivers (for secure measurements).

U-Boot is split into a *secondary program loader* (SPL) and *tertiary program loader* (TPL). The SPL is more simple, and is only responsible for loading the TPL, which supports more advanced methods of system initialisation.

U-Boot supports a command line system before booting, and boots either according to a preset *boot script* or according to commands entered manually over a serial console. These commands can be extended by vendors and allow developers, administrators and end users to customise their boot process.

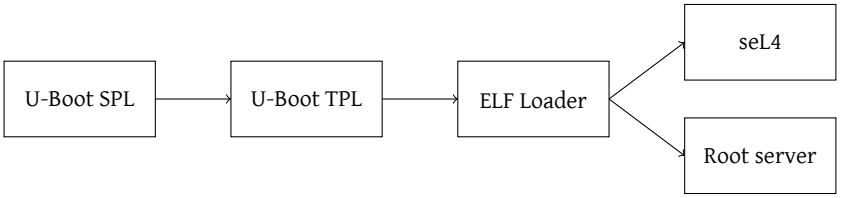


Figure 2.14: A high-level overview of the seL4 boot process, assuming U-Boot is the bootloader.

2.7 ARM TrustZone

ARM TrustZone is a firmware supported means of isolation on modern ARM processors. Figure 2.15 gives an overview of a system running with TrustZone. It separates execution and memory on the processor into the *secure world* and *normal world*. The normal world hosts the familiar operating system and userspace programs, the secure world can run secure world services (also called *trusted applications*)

which are isolated from the normal world. This isolation is backed by a *coarse context switch*: all memory is either marked as secure or non-secure, and secure memory can only be accessed in the secure world.

This kind of isolation could be achieved in theory with regular processes. However, because the secure world is available *before* the normal world and because it exists concurrently to the normal world, it can be particularly useful for platform security. Measurements can be made in the bootloader, stored using TrustZone, then used from the operating system. TrustZone can also be used to measure parts of the normal world before they have a chance to boot.

Additionally, on ARM platforms, some hardware can be configured to only be accessible from TrustZone. This forms the foundation of the secure storage required to implement a TPM.

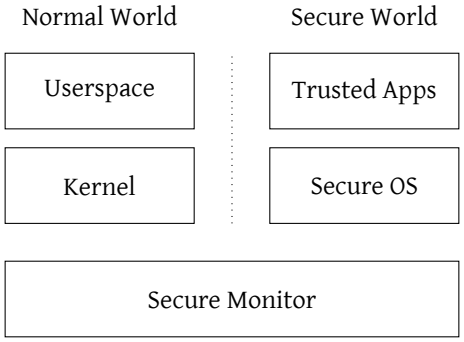


Figure 2.15: A high-level overview of a system with TrustZone.

2.7.1 Secure Monitor

Underneath any system with TrustZone is the *secure monitor*. This is a small piece of software which acts as the aforementioned Core Root of Trust Measurement (CRTM), and mediates the context switches between the normal and secure world.

Communication with the secure monitor (and thus context switches between the normal and secure world) are achieved using the `smc` instruction. This follows a particular calling convention, in which parameters to calls to the normal or secure world are placed in the general purpose registers (`x0`—`x7` by default).

One caveat of the `smc` instruction is that it can only be executed in *privileged mode*. This is a design issue for microkernels, as it means that the equivalent of a driver for TrustZone cannot be written fully in userspace. The only option is to somehow move the `smc` instruction into the kernel. Since the secure world can access and change normal world memory, if the secure world is not trusted, the `smc` instruction also cannot be trusted.

2.7.2 Secure World Operating System

On top of the secure monitor, the secure world has analogues of an operating system and processes. The secure world operating system is designed to act as an operating system for secure world services (processes), mediating their access to hardware and providing simple abstractions for them to use. These secure world services provide a service which takes advantage of the isolation provided by the secure world.

The secure world operating system should be as minimal as possible, this makes seL4 a prime candidate. However, the current most well-supported secure world operating system is *OP-TEE*. To ensure a minimal trusted computing base, OP-TEE is supported in the normal world by a *supplicant*. This supplicant is used to co-ordinate shared memory between the normal and secure worlds, and to offload any functionality onto the normal world which can be offloaded. For example, access to the secure storage.

2.8 Secure Storage

Multi-Media Card (MMC) is a storage standard which is aimed at embedded systems. It covers both removable cards (SD/microSD) and embedded cards, which are present as chips on the device. Some embedded chips will implement a *Replay-Protected Memory Block* (RPMB), which can be used to provide a means of secure storage only accessible to the secure world.

The RPMB is available as a special partition with one logical block on the MMC device. RPMB *commands* can be issued by writing to this block, and responses received by reading from this block, much like a special device file on UNIX. The RPMB is equipped with a key during manufacturing or provisioning, with the idea that this key is only accessible to the RPMB and to the secure world. This key allows the secure world to read and write to the replay-protected memory via the normal world without the normal world being able to intercept the requests. Programming this key is a one-time operation.

The key can be restricted to the secure world by a number of means. The simplest is to use re-write the secure world system to use a proprietary method of generating the key and restrict the source code for the secure world. While simple, this method relies on security by obscurity. A more secure way is to take advantage of devices restricted to the secure world and program the RPMB key into an electronic fuse only accessible to the secure world.

Related Work

Now we can explore some existing work which could make our approach easier. In particular, we look at *fTPM* — an implementation of a firmware TPM using ARM's TrustZone [Raj et. al. 2016]; *HYDRA* — an implementation of a "hybrid" TPM using seL4 and a small amount of secure hardware [Eldefrawy et. al. 2017]; and *SABLE* — a formally verified bootloader which provides secure boot and local attestation using a TPM [Constable et. al. 2018].

SABLE and HYDRA are both solutions for securely booting and attesting seL4-based systems, with different characteristics. Table 3.3 compares SABLE and HYDRA on a number of key points, which we will use later to discuss our own approach. fTPM is a different kind of existing work in that it could help us to design our solution, but is not necessarily a solution by itself.

	SABLE	HYDRA
Platform Support	x86	ARM
Hardware Requirements	Discrete TPM	None
Secure Boot Support	Yes	Yes
Attestation Support	Local only	Local or remote
Attestation Scope	Whole platform	Userspace
Root of Trust Measurement	Dynamic	Static
Verification	Yes	Planned

Table 3.3: Comparison of SABLE and HYDRA.

3.1 Microsoft's Firmware TPM

As we mentioned before, ARM platforms do not necessarily support secure boot or attestation by hardware or firmware out of the box. Instead, modern ARM SoCs will support ARM TrustZone — usually with some extra secure hardware to increase the SoC's security capabilities. In order to securely boot and support Windows 10 systems on ARM devices Microsoft has developed *fTPM* — a TPM fully implemented in software inside ARM TrustZone's secure world.

3.1.1 Design Overview

Recall that TrustZone separates the system into a normal and secure world, separated in firmware by a coarse context switch. Microsoft's fTPM has two components: a driver that runs in the normal world and allows the operating system to interface with the fTPM, and the fTPM itself running in the secure world and implementing the TPM.

The fTPM places three extra hardware requirements on the system outside of ARM's TrustZone: a secure entropy source (that is, an entropy source only available to TrustZone), secure storage, and a number of *hardware fuses* — small portions of write-once storage that are written with secure keys during manufacturing. The secure storage comes in the form of an *embedded Multi-Media Card* (eMMC) that is equipped with an RPMB.

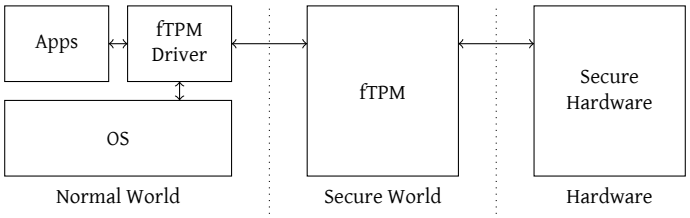


Figure 3.16: Overview of fTPM's Architecture.

3.1.2 Limitations

The biggest limitation of Microsoft's fTPM is the availability. Primarily, it depends on ARM TrustZone not only being present on the SoC, but also available for use. Although all modern ARM SoCs do contain TrustZone by specification, this is usually restricted by the manufacturer for various reasons. In particular, they may offer their own bespoke security services on top of TrustZone and may not allow these to be easily modified.

fTPM also makes some small modifications to the semantics of some TPM commands. However, these commands are still largely the same, and these commands are not ones which are important for attestation or secure boot.

3.2 HYDRA

Hybrid Design for Remote Attestation or *HYDRA* is a system which implements a kind of "hybrid" TPM using seL4 and some secure hardware, intended for platforms which do not have the full support for a hardware or firmware TPM. These would be older or smaller ARM systems, or systems with more obscure architectures. The hybrid TPM takes advantage of the isolation capabilities of seL4 and provides a software TPM as a separate seL4 task, which can then be used by remote verifiers or other seL4 tasks such as a virtualised commodity operating system.

This is not quite what we are trying to achieve, but it does demonstrate a unique architecture for a TPM that is only possible with a microkernel with strong isolation properties such as seL4. In particular, this is an architecture that was initially proposed by NICTA when seL4 was first released [Heiser et. al. 2011].

3.2.1 Design Overview

This is a system targeted specifically at remote attestation of a system running on seL4. It makes use of a higher priority attestation task running alongside the system, and a platform-supported secure boot to ensure the integrity of seL4 and the attestation task itself. In particular there are no requirements on the platform outside of this secure boot support.

By running the attestation task with a higher priority than the virtualised system, the virtualised system can be interrupted and attested to at any time, which is more flexible than normal TPM attestation. It also places a minimal amount of trust in hardware and firmware solutions, relying on software to provide isolation and security. This means it cannot defend against some hardware and firmware level attacks, but also means it is less susceptible to vulnerabilities as it can rely on formally verified software which can be updated and patched easily.

3.2.2 Limitations

The main limitation of HYDRA is the dependence of platform-level secure boot and the reliance on this to attest any firmware or the seL4 system itself. The entire attestation program *including the private key* is stored un-encrypted on regular storage. If the platform-level secure boot is compromised or disabled, then an attacker can simply read this storage and leak the private key, allowing them to fake attestation messages. In this way, the system is only secure and trustworthy if seL4 is securely loaded by the platform in the first place and can control access to storage. This means it cannot truly replace a TPM or TPM attestation, although it can provide similar functionality for the userspace system.

3.3 SABLE

The *Syracuse Assured Boot Loader Executive* or *SABLE* is a formally-verified secure bootloader, supporting secure boot and local TPM attestation. It was originally intended for seL4/Genode systems, but is theoretically capable of booting any system.

3.3.1 Details

SABLE works in much the same way as a normal bootloader with security features added (e.g. the *Grand Unified Bootloader* (GRUB)) in that it takes advantage of the platform's support for secure boot and whatever hardware or firmware TPM exists. However, it is targeted specifically for x86 platforms and as such relies on whatever firmware TPMs are provided by the processor, as opposed to using fTPM or some other third-party firmware TPM.

SABLE is unique in that it specifically makes use of a *Dynamic Root of Trust Measurement* (DRTM) as opposed to a *Static Root of Trust Measurement* (SRTM) as is typical of TPM attestation. For attestation using SRTM, the entire platform initialisation starts at the secure Core Root of Trust Measurement (CRTM) which measures the next stage and establishes a chain of trust across the entire boot process. For attestation using DRTM, once the platform is fully initialised a special CPU instruction can be issued (`skinit` for AMD platforms and `SENTER` for Intel platforms) which effectively clears all CPU state and starts a new secure loader from which new measurements are taken.

This DRTM means that the rest of platform before the secure loader can be ignored and excluded from the *Trusted Computing Base* (TCB). Minimising the TCB is a well-established security practice as it makes it easier to secure what *is* inside the TCB. However, this relies on the CPU and the platform securely and reliably clearing its state.

3.3.2 Limitations

The main limitation of SABLE is the amount of platform dependence. Firstly, by relying on hardware or native firmware TPMs, it restricts itself to platforms which have these. Most importantly however, the reliance on a DRTM and the platform-specific DRTM instructions means it can only work on platforms which support these instructions. Currently both of these restrictions limit SABLE to modern commodity x86 platforms.

It also specifically targets the TPM 1.2 standard as opposed to the TPM 2.0 standard. The TPM 2.0 standard offers improved security guarantees in a number of areas, but is *not* backwards compatible with TPM 1.2, and a non-trivial porting effort would be required to make SABLE support TPM 2.0.

Finally, the use of local attestation over remote attestation means that SABLE can only be used to provide assurance if a human verifier is present. However with the right userspace tools SABLE could be configured to also provide remote attestation, as it still produces the required TPM measurements.

Approach

The initial problem statement of introducing secure boot and attestation to seL4-based systems is quite broad, and the required design will depend highly on the chosen platform and chosen secure boot and attestation implementations. In particular, it would be impossible to have one general solution for all platforms.

To start with, before choosing a particular platform or implementation, we will lay out the desired requirements and design goals for an ideal solution. We then lay out a proposed approach for the implementation.

4.1 Requirements

4.1.1 Design Goals

We begin by choosing some over-arching design goals. These are based heavily on the design goals of seL4 [Heiser 2020], to ensure the best integration into an seL4-based system.

Requirement 1 (Security) *The design should follow the current state of the art in security practices, using as many of the available security mechanisms as possible in the correct way.*

Requirement 2 (Performance) *The implemented solution should not cause a significant overhead in the boot process and should not impede the operation of the system after the boot process unless specifically invoked.*

Requirement 3 (Policy Freedom) *The design should not require the system running on top of seL4 to adhere to a particular security policy. Designers of the 'end user' system should be able to freely use the TPM and attestation facilities in whatever way they see fit.*

Requirement 4 (Verification) *While full verification is outside the scope of this thesis, the design and implemented solution should at least be amenable to verification, with a strong specification and a relatively small codebase.*

We can now lay out some concrete, high-level requirements for the implementation. These are directly drawn from the thesis statement, and are independent of the chosen platform (as long as the platform supports some form of TPM).

Requirement 5 (Secure Boot) *The design should leverage some hardware or firmware functionality to attempt to certify a cryptographic signature for seL4 and the initial task, and should fail to boot if it cannot certify those signatures.*

Requirement 6 (TPM Attestation) *The design should leverage a hardware or firmware TPM to provide remote attestation to a trusted remote verifier, through the use of a CAmkES module.*

4.1.2 Targeted Platform

The exact nature of the solution depends on the platform chosen, since each platform has varying levels of support for secure boot and attestation and different implementations of each.

Since RISC-V does not yet have good support for these technologies, and the problem already seems solved on x86 platforms by SABLE, we will specifically be targeting *ARM* platforms. This would be a good target, as ARM is a popular choice for embedded systems, which is the largest use case for seL4.

Requirement 7 (Chosen Platform) *The implementation should work on any ARM platform which supports some level of secure boot on the firmware and fTPM via. TrustZone.*

As we have seen, the choice of ARM introduces a number of challenges. We cannot rely on DRTM as ARM lacks the appropriate instructions, we need access to TrustZone on whichever SoC is chosen along with the appropriate secure hardware, and we need some level of support for secure boot from the underlying firmware.

4.2 Design

We will start by looking at how the existing work relates to the requirements listed above. Table 4.4 lists how each of the existing solutions matches each requirement.

Requirement	SABLE	HYDRA
Security	Yes	No (unsuitable TCB)
Performance	Unknown	Unknown
Policy Freedom	No	Yes
Verification	Yes	Planned
Secure Boot	Yes	Yes
Attestation	No (local only)	Yes
Platform	No (x86 only)	Yes

Table 4.4: Evaluation of SABLE and HYDRA against the stated requirements.

As we can see from Table 4.4, neither solution exactly meets our requirements. However, by leveraging fTPM and looking into platform-specific features for secure boot, we may be close to meeting our requirements. fTPM will require some effort to port to seL4, and secure boot will be somewhat platform-

dependent, but this will fit better into the time constraints of this thesis than adapting any of the other existing work.

The proposed solution consists of the following stages:

- Investigating support for fTPM, TrustZone and secure boot on the chosen platform.
- Writing drivers to support the secure world operating system and fTPM from within seL4.
- Implementing any userspace software and other utilities to support attestation.

The new boot and attestation process will work like so:

1. The platform's secure monitor starts in the secure world, initialises fTPM, and measures U-Boot.
2. The secure monitor switches to the normal world and starts U-Boot.
3. U-Boot measures the seL4 image (including the kernel and root server), and boots into seL4.
4. The seL4 root server uses fTPM to generate an attestation to provide to the remote verifier.

This also assumes some provisioning and external utilities, which are also in the scope of this thesis:

- Provisioning the chosen platform with the necessary keys.
- Userspace utilities (e.g., a root server library) to use TPM commands to generate an attestation.
- An external verifier program which is able to verify a generated attestation.

Implementation

Due to time constraints and unforseen roadblocks, it was only possible to implement most of the TPM attestation aspect of this thesis. The implementation can be considered from the 'bottom-up' with regards to TPM attestation:

1. Choosing a suitable platform.
2. Accessing the secure monitor from within seL4.
3. Installing and accessing a secure world operating system (OP-TEE).
4. Implementing the required normal world supplicant for OP-TEE in seL4.
5. Installing and accessing fTPM as a secure world service from within seL4.
6. Provisioning fTPM with the required keys for attestation.
7. Implementing the standard attestation workflow commands within seL4.
8. Implementing an external application for verification.

Many pieces of this implementation exist in some form on some other platform, usually embedded Linux. Therefore, most of the implementation phase involved understanding and porting these onto seL4, with some adaptations where necessary. Some cases where documentation is missing or difficult to find (such as OP-TEE and fTPM) required some imagination and reverse engineering.

5.1 The Platform

The chosen platform was the *i.MX8 MQ Evaluation Kit* (henceforth referred to as "the i.MX8"), a development board based on the i.MX8 MQ applications processor shown in Figure 5.17. This is a platform well supported by seL4. Importantly for this thesis, it also has a somewhat well-documented boot process, and support for open source configurations of OP-TEE and fTPM. It also has the secure storage support required for fTPM, through the embedded MMC card.

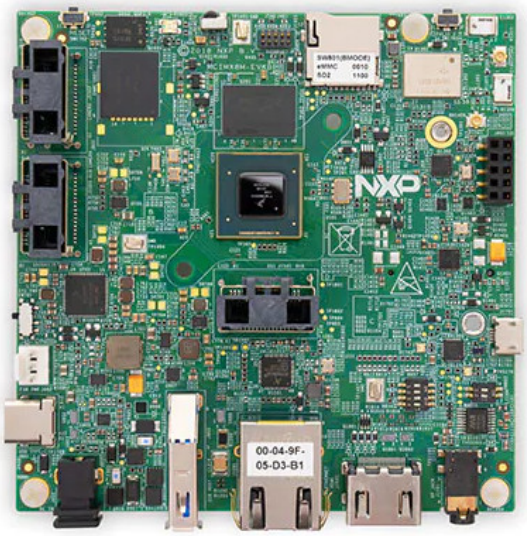


Figure 5.17: A marketing photo of the i.MX8 MQ Evaluation Kit, with the heatsink removed. The SoC is in the centre of the board, and the eMMC containing the Replay-Protected Memory Block (RPMB) is located to the top-left of the board. Image by NXP. Source: `https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/evaluation-kit-for-the-i-mx-8m-applications-processor:MCIMX8M-EVK`

5.2 `smc` Instruction

On any ARM system that supports TrustZone, the `smc` instruction can be used to access the secure world. Therefore, this is a natural prerequisite for the rest of the system.

As mentioned in the background section, the `smc` instruction can only be called from kernel mode. This means that seL4 has to be modified with an extra capability to allow `smc` calls.

To implement this thesis, we introduce an extra architecture-specific capability for Aarch64 called `seL4_ARM_SMC`. The capability has only one kind of call, `seL4_ARM_SMC_SMC`. This takes the input registers for the `smc` call, makes the `smc` call, and returns the output registers for the `smc` call. The input and output registers are marshalled and unmarshalled into structures by `libsel4`.

On Aarch64 systems, this capability is placed into the root tasks's CSpace and the BootInfo structure. The capability holds no information on the kernel side, and simply represents the ability of a thread to make an `smc` call through the kernel. The intended design is that only one or two threads acting as servers actually have the capability, and that clients wishing to interact with the secure world should have endpoints to these servers.

`smc` calls are dangerous, and in a secure system will require some kind of access control. One way to do this would be to have these access controls implemented somehow in seL4, this would provide better security guarantees and adhere to the principle of least privilege by only giving userspace threads just

enough access to the secure world [VanVossen 2021].

However, the arguments to `smc` calls can be complicated, and may vary in structure depending on the platform and what is running in the secure world.

For instance, the calling convention specifies that the function ID being called in the secure world should be placed in `x0`. In that case, seL4 may choose to filter calls based on the function ID. However, in the case of OP-TEE, most function calls use the *same* function ID (`OPTEE_SMC_CALL_WITH_ARG` (`0x32000004`)). The secure world function being called is stored in the shared memory.

To minimise the complexity of the kernel and provide true policy freedom, it is better to have an all-powerful capability restricted to a minimal amount of threads, and to implement the access controls in userspace.

One potential issue left to be addressed is long-running operations in the secure world. Some secure world operations are considered atomic and do not enable interrupts. Others are considered to be interruptable and can be paused and resumed, during which time the system may return to the normal world.

It is currently not clear to me how this would interact with the seL4 scheduling requirements, particularly MCS scheduling budgets. In particular, operations in the seL4 kernel are assumed to be short, which results in a kernel-wide lock. Since these `smc` calls take place in the kernel, this may cause system-wide pauses whenever a long-running secure world operation is started.

5.3 OP-TEE and OP-TEE Supplicant

Before accessing OP-TEE from within seL4, OP-TEE needs to be present on the system. For the i.MX8, OP-TEE is not installed out of the box, only the secure monitor. The open source OP-TEE repository [OP-TEE 2021] includes a manifest and Makefile for the i.MX8, this builds OP-TEE, U-Boot, TF-A (the open source reference secure monitor) and Linux.

The Makefile then produces an image which can be flashed onto a microSD card, which can be booted from directly on the i.MX8 by selecting the microSD on a hardware switch. This image will boot Linux by default, however we can modify the included boot script to boot an seL4 image over the network, or copy an seL4 image onto the microSD itself. The image also ensures that OP-TEE boots first as a secure world operating system.

Recall that OP-TEE is paired with a supplicant which acts on behalf of OP-TEE in the normal world. This supplicant is responsible for managing shared memory and for storage requests. For the supplicant, the Linux drivers are slightly more complex, as the supplicant is in userspace and the driver itself is in the kernel. For seL4 this can be simplified, as both are in userspace.

From there, drivers are required from within seL4 to access OP-TEE. For the driver for OP-TEE and its supplicant, these were derived from the Linux drivers. Secure storage requires MMC drivers, which I

ported from U-Boot. These required some modification to work properly within seL4.

5.3.1 OP-TEE Messages

OP-TEE messages are sent through the `smc` instruction. These only use a limited number of the available registers for the `smc` calling convention, and pass more information through shared memory objects. These shared memory objects are allocated in the normal world, and are sent by physical address to the secure world.

OP-TEE shared memory comes in a number of different forms, all of which require slightly different handling:

- Shared memory can span multiple pages. In this case, the physical address passed to OP-TEE points to a page containing a list of physical addresses of other pages. This is also to allow support for non-contiguous memory mappings.
- Shared memory can be allocated on request by OP-TEE, or allocated in advance by the normal world. For memory allocated in advance — except for memory used to hold arguments — it must be *registered* with OP-TEE. This involves making another `smc` call to OP-TEE which contains the physical address of the shared memory object.

For all of these shared memory objects, the cache must be invalidated and flushed before and after using as for any DMA operations. This is because the secure world and normal world share different mappings.

Currently, these shared memory objects are allocated using a bump pointer for testing purposes. However, since shared memory is always allocated as whole pages, this could be changed to use a separate frame allocation library with a proper free list.

The responses are delivered in the same shared memory objects. These responses may indicate success or an error, and contain some payload. However, in some cases, these responses may be a *Remote Procedure Call* (RPC) response. This is a way of OP-TEE indicating to the normal world that some action from the supplicant (e.g. allocation of shared memory, secure storage access) is required before the message can be addressed. Figure 5.18 is a sequence diagram showing how RPC responses work. Multiple RPC responses may need to be handled before OP-TEE can handle the original request.

OP-TEE messages that are sent to a secure application are associated with a *session*. These sessions are opened using OP-TEE messages themselves, and use a TA's *Universally Unique Identifier* (UUID) to select the TA associated with the session.

Figure 5.19 shows how the fTPM library initialises itself, by opening an OP-TEE session using fTPM's UUID. Figure 5.20 and Figure 5.21 show how the fTPM library can send TPM commands, using OP-TEE messages.

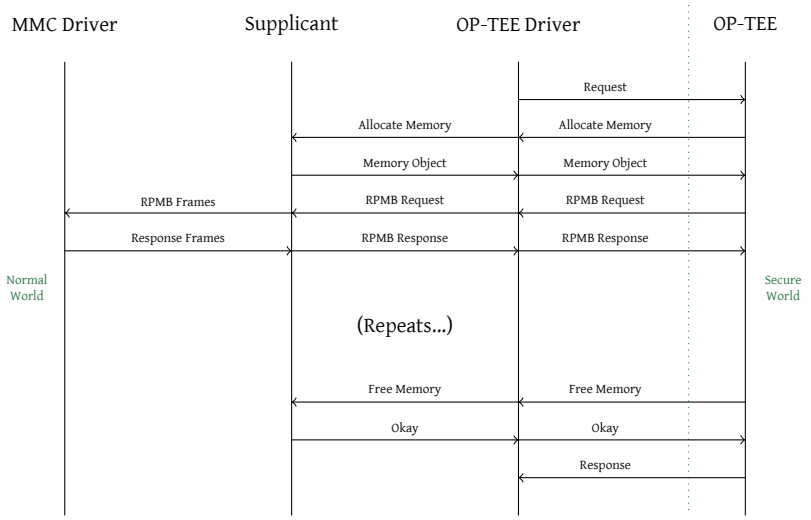


Figure 5.18: The full process for sending an OP-TEE message, including any intermediate RPC responses.

5.3.2 Secure Storage

Access to secure storage is highly platform dependent and is likely to be a problem for a small trusted computing base. Instead, OP-TEE forms platform indepenent Replay-Protected Memory Block (RPMB) commands in the secure world, and routes these through to the normal world in RPC responses. The normal world then needs an MMC driver to service these RPMB requests.

In the case of the i.MX8, the seL4 ecosystem already has a similar driver (for the i.MX6). However, U-Boot already has a driver specifically for the i.MX8, which is more suitable [Denk 2021]. I also ported the clock driver from U-Boot, as this is required for the MMC driver to work.

Before sending the RPMB requests, some initialisation is required from the MMC driver:

1. The eMMC device itself is re-initialised, to clear any settings left behind by U-Boot. I found that the device was only partially cleared at the end of the boot process.
2. Some information is requested from the eMMC device. In particular, the Card ID, reliable sector count, and RPMB partition size. These may be requested by OP-TEE at any time, usually during initialisation.
3. The eMMC device should be switched to the RPMB partition. If the eMMC is being used for storage purposes, this will need to happen every time the RPMB is needed.

With this initialisation complete, RPMB requests can be sent and received by reading and writing to logical blocks on the eMMC device.

5.4 fTPM

fTPM exists as a *trusted application* (TA) running on top of OP-TEE. Generally, OP-TEE loads a trusted application from the file system on request from the normal world. However, in the case of fTPM, we need it to be available before the normal world has loaded and before a file system is available. We can do this by building OP-TEE with fTPM included as an *early TA*. These TAs are linked directly into the OP-TEE image and are available as soon as the system starts, and more importantly do not require a file system to be used.

With fTPM linked as an early TA and with OP-TEE messages available in seL4, we have everything we need to use TPM2 commands. We start in seL4 by opening a session with fTPM. For normal trusted applications this would load the application into memory, but in the case of fTPM this simply establishes some state within OP-TEE. Then we can send TPM commands and receive TPM responses wrapped within OP-TEE messages. Figure 5.22 shows a snippet of code which interfaces with fTPM from within seL4.

The code that the fTPM library uses itself to communicate with OP-TEE (in Figure 5.19, Figure 5.20 and Figure 5.21) is also adapted from U-Boot, with some adjustments for my implementation of the OP-TEE driver.

5.5 Key Provisioning

For attestation to work, the system needs to be provisioned with an RPMB key and an attestation key, as well as any primary seeds required by the TPM. The RPMB key can be configured to be written by OP-TEE automatically if it has not been programmed yet, and fTPM will initialise the primary seeds if those have not been initialised yet. Provisioning the attestation key must be done manually using the `TPM2_Create` command.

Since U-Boot has file system support and all the relevant drivers, we can extend U-Boot's command system to include a command for creating an attestation key. This will place the public and private components in memory, and these can be written to a file on the microSD card for transfer to a remote computer.

5.6 Example Setup

For the thesis demonstration, and to evaluate this prototype, we have a small example of attestation in action. This consists of:

- A U-Boot script that pulls a complete seL4 + root server image off TFTP and measures it into PCR 0.

- An seL4-power serial console diary, which provides the value of PCR 0 as well as a signed quote of the value in PCR 0.
- A remote verifier, implemented as a web application on a separate computer.

The verifier checks, in order:

1. That the attestation structure is valid and contains a TPM quote.
2. That the PCR digests provided match those in the attestation structure.
3. That the signature is a valid signature for that attestation structure (using an OpenSSL command).
4. That the attestation structure has a safe clock value.
5. That no previous valid attestation attempts have the same or higher clock values. This means checking the clock, reset and restart values.

If these checks pass, the verifier adds this attestation attempt to the history of valid attestation attempts.

If an attacker manages to hack the TFTP server, they can replace the seL4 image. However, U-Boot and OP-TEE are both written on the microSD card, so these are not compromised. For the example, we consider three different situations:

1. The diary image is not compromised. (Figure 5.23)
2. An attacker replaces seL4 with 'eL4' — a version of seL4 with worse security guarantees than seL4. The root server image is unchanged. (Figure 5.24)
3. As above, but the attacker replaces the root server image with one that returns a hardcoded, known good PCR value and quote. (Figure 5.25)

In the first case, the verifier application will in fact successfully verify the attestation. However, the PCR value is clearly different. This presents a practical drawback of TPM attestation. If a PCR value changes, it can be difficult to know if this is from an attack, or simply a system update (for example). For this example setup, we place the responsibility of checking the PCR on the end user.

In the second case, there are two possibilities. If the verifier application has seen a quote previously with a *later* clock value than the hard-coded quote, the hard-coded quote fails to verify as the clock values are considered to be in the past. If the verifier application has not seen a quote previously with a later clock value than the hard-coded quote, the diary application is configured to re-generate the quote on pressing `@`. If the quote is hard-coded, the same quote will be produced, and the verifier application will detect that the same clock value has been used twice.

```c
#include <optee.h>

static struct {
    uint32_t session;
    struct optee_shm* shm;
    uint8_t resp[PAGE_SIZE];
} ftpm_state;

int ftpm_init(void)
{
    struct optee_open_session_arg arg;
    memset(&arg, sizeof(arg), 0);

    /* Copy in fTPM's UUID */
    const uint8_t ftpm_uuid[16] = {
        0xbc, 0x50, 0xd9, 0x71,
                0xd4, 0xc9, 0x42, 0xc4,
                0x82, 0xcb, 0x34, 0x3f,
                0xb7, 0xf3, 0x78, 0x96
    };
    memcpy(arg.uuid, ftpm_uuid, 16);

    /* Unused parameters */
    arg.clnt_login = 0;
    arg.num_params = 0;

    int rc = optee_session(&arg, NULL);
    assert(rc); // Assert no OP-TEE errors

    assert(arg.ret == 0); // Assert no fTPM error

    ftpm_state.session = arg.session;

    /* This is used for requests and responses */
    /* The arguments here are:
         reg = TRUE - Register this memory with OP-TEE
         pl = TRUE  - Use a page list as this memory may use
                      non-contiguous pages */
    ftpm_state.shm = optee_shm_alloc(PAGE_SIZE * 2, TRUE, TRUE);

    return 1;
}
```

Figure 5.19: A simplified version of the code used to open an fTPM session with OP-TEE.

```c
#include <optee.h>

static struct {
    uint32_t session;
    struct optee_shm* shm;
    uint8_t resp[PAGE_SIZE];
} ftpm_state;

struct tpm_header {
        uint16_t tag;
        uint32_t length;
        union {
                uint32_t ordinal;
                uint32_t return_code;
        };
} __attribute__((packed));

/* Swap for endianness */
static uint32_t swap_32(uint32_t i);

int ftpm_command(uint8_t* input_buf, uint8_t* output_buf,
    size_t input_length, size_t output_length)
{
    struct optee_invoke_arg arg;
    memset(&arg, sizeof(arg), 0);

    arg.func = 0;
    arg.session = ftpm_state.session;
    arg.num_params = 4;

    struct optee_msg_param params[4];
    memset(&params, 4 * sizeof(struct optee_msg_param), 0);

    /* First parameter, the input buffer */
    params[0].attr = 0x5; // Registered memory input
    params[0].u.rmem.shm_ref = (uint64_t) ftpm_state.shm;
    params[0].u.rmem.size = input_length;
    params[0].u.rmem.offs = 0;

    uint8_t* shm_buf = (uint8_t*) ftpm_state.shm->vaddr;
    memset(shm_buf, 2 * PAGE_SIZE, 0);

    /* Copy input buffer into shared memory */
    assert(input_length < PAGE_SIZE);
    memcpy(shm_buf, input_buf, input_length);

    optee_shm_clean_invalidate(ftpm_state.shm);
```

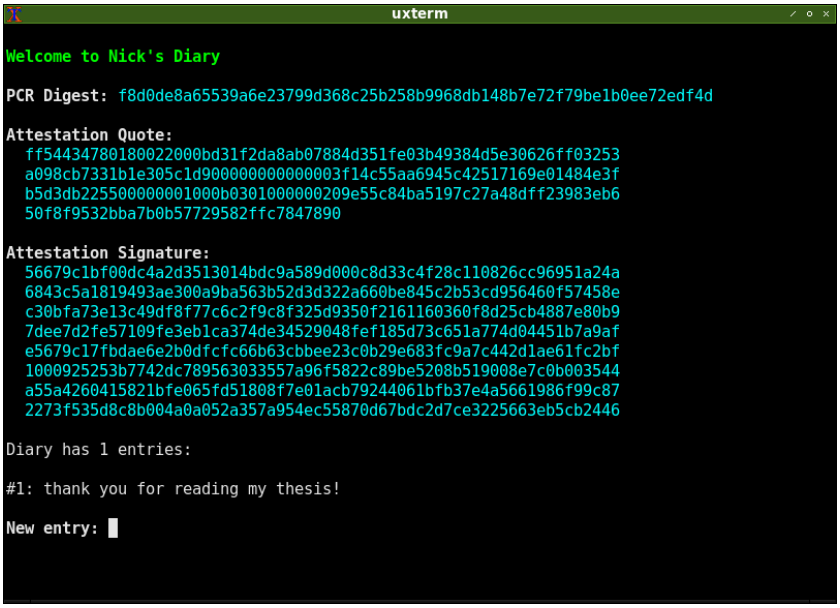
Figure 5.20: A simplified version of the code used to send TPM commands as OP-TEE messages to fTPM.

```
    /* Second parameter, the output buffer */
    params[1].attr = 0x7; // Registered memory input/output
    params[1].u.rmem.shm_ref = (uint64_t) ftpm_state.shm;
    params[1].u.rmem.size = PAGE_SIZE;
    params[1].u.rmem.offs = PAGE_SIZE;

    int rc = optee_invoke(&arg, params);
    assert(rc); // Assert no OP-TEE errors

    optee_shm_clean_invalidate(ftpm_state.shm);

    assert(arg.ret == 0); // Assert no fTPM errors

    uint8_t* tmp_buf = ftpm_state.shm->vaddr + params[1].u.rmem.offs;
    struct tpm_header* tpm_hdr = (struct tpm_header*) tmp_buf;
    size_t tmp_len = swap_32(tpm_hdr->length);

    assert(tmp_len > output_length); // Ensure we have enough space

    /* Copy output buffer from shared memory */
    memset(output_buf, output_length, 0);
    memcpy(output_buf, tmp_buf, output_length);

    return 1;
}
```

Figure 5.21: A simplified version of the code used to send TPM commands as OP-TEE messages to fTPM *(Continued from Figure 5.20).*

```c
#include <ftpm.h>
#include <optee.h>

/* Swaps the endianness, as the TPM and Aarch64 use */
/* different endianness */
static uint16_t swap_16(uint16_t i);
static uint32_t swap_32(uint32_t i);

/* Generate 32 random bytes, and copy to 'random' */
void example(uint8_t* random)
{
    /* Initialise OP-TEE, and give it an address to start */
    /* allocating shared memory from */
    optee_shm_init(0xa0000000);

    /* Initialise fTPM */
    int rc = ftpm_init();
    assert(rc); // Assert no initialisation errors

    /* Prepare the TPM request */
    struct __attribute__((__packed__)) {
        uint16_t tag;
        uint32_t length;
        uint32_t code;
    } tpm_header_req;

    tpm_header_req.tag = swap_16(0x8001);  // No sessions
    tpm_header_req.length = swap_32(12);   // 10 byte header
                                           // + 2 byte parameter
    tpm_header_req.code = swap_16(0x017b); // TPM2_GetRandom

    struct __attribute__((__packed__)) {
        uint16_t bytes;
    } tpm_getrandom_req;

    tpm_getrandom_req.bytes = swap_16(32); // Request 32 bytes

    uint8_t request[12];  // 10 byte header + 2 byte parameter
    uint8_t response[42]; // 10 byte header + 32 byte response
    memcpy(request, &tpm_header_req, sizeof(tpm_header_req));
    memcpy(request + sizeof(tpm_header_req), &tpm_getrandom_req,
        sizeof(tpm_getrandom_req));

    rc = ftpm_command(request, response, 12, 42);
    assert(rc); // Assert no OP-TEE errors

    /* Unmarshall the response */
    struct __attribute__((__packed__)) {
        uint16_t tag;
        uint32_t length;
        uint32_t code;
    } tpm_header_res;

    memcpy(&tpm_header_res, response, sizeof(tpm_header_res));
    tpm_header_res.code = swap_32(tpm_header_res.code);
    assert(tpm_header_res.code == 0); // Assert no TPM errors

    memcpy(random, response + sizeof(tpm_header_res), 32);
}
```

Figure 5.22: Using fTPM from within seL4 as a userspace library. This implements a function which initialises fTPM and OP-TEE, then uses `TPM2_GetRandom` to fetch 32 random bytes.



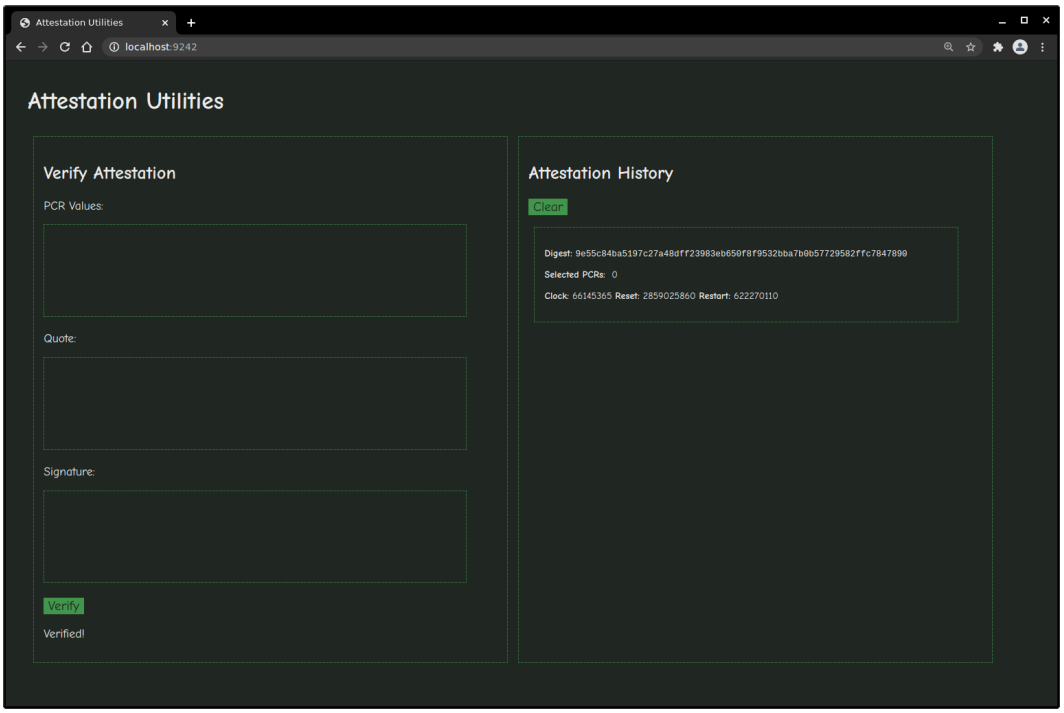
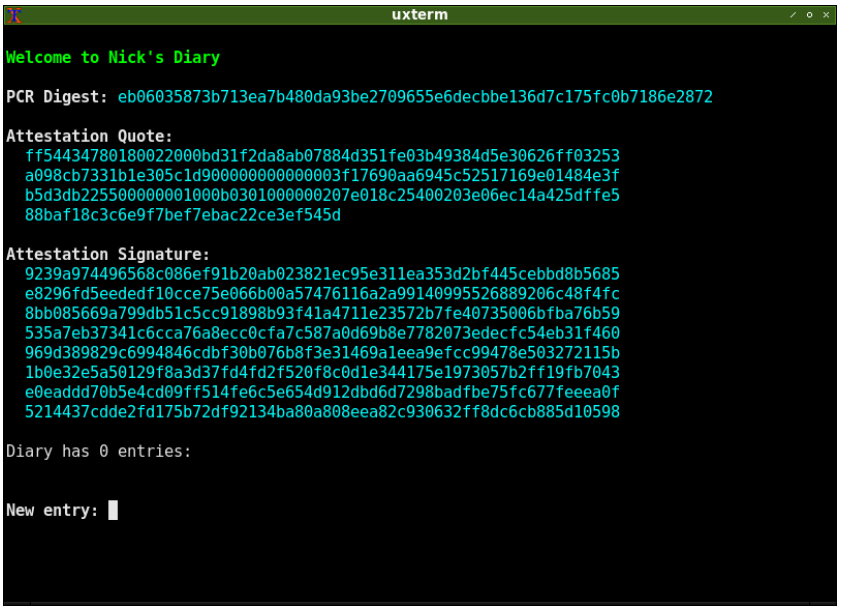


Figure 5.23: The known good version of the diary, and the result of entering the details into the verification application.



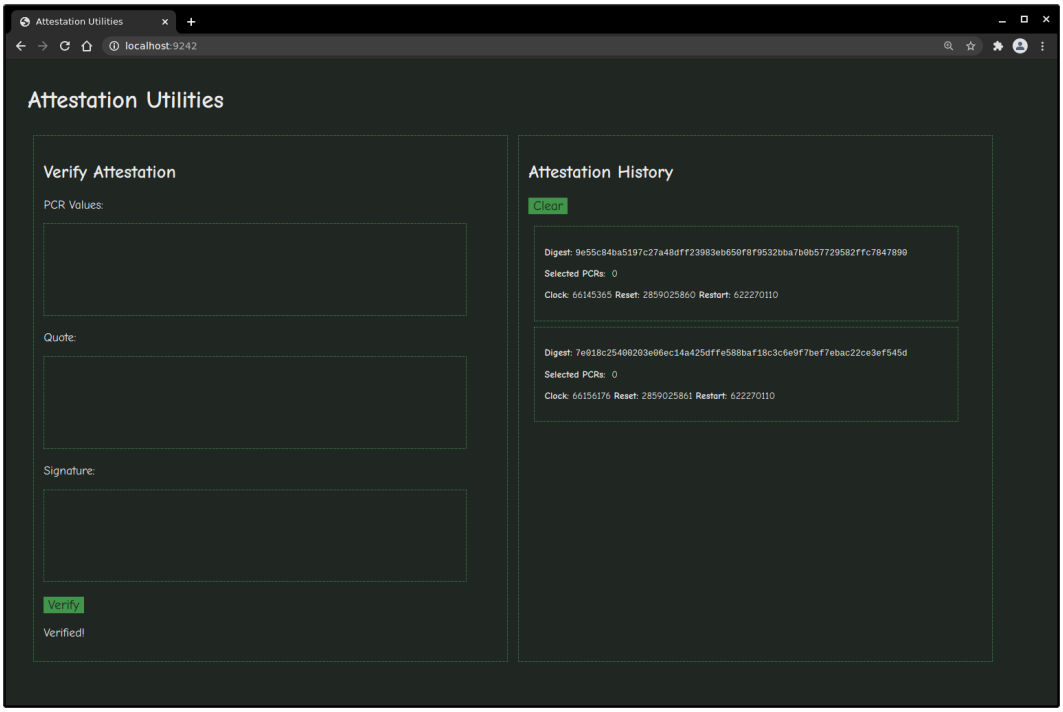
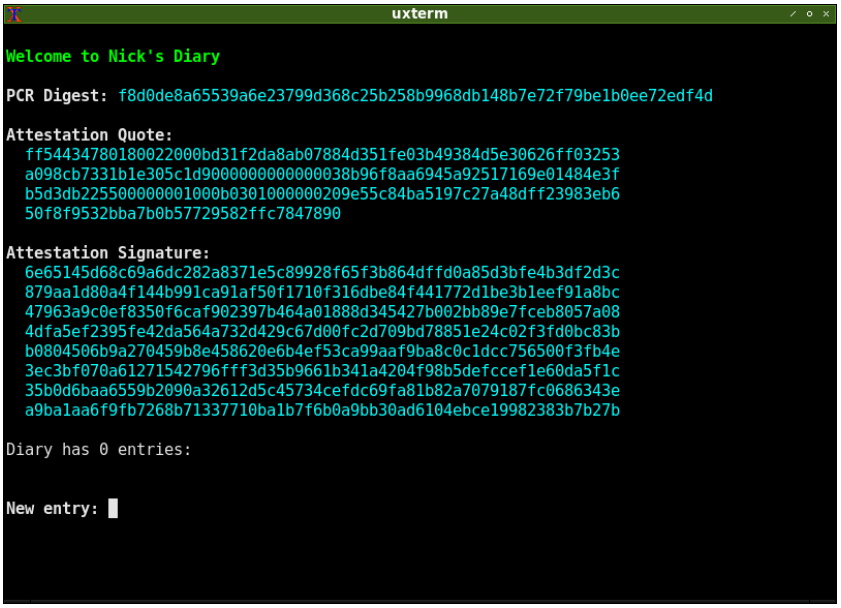


Figure 5.24: The compromised version of the diary. Note that the PCR digest has changed.



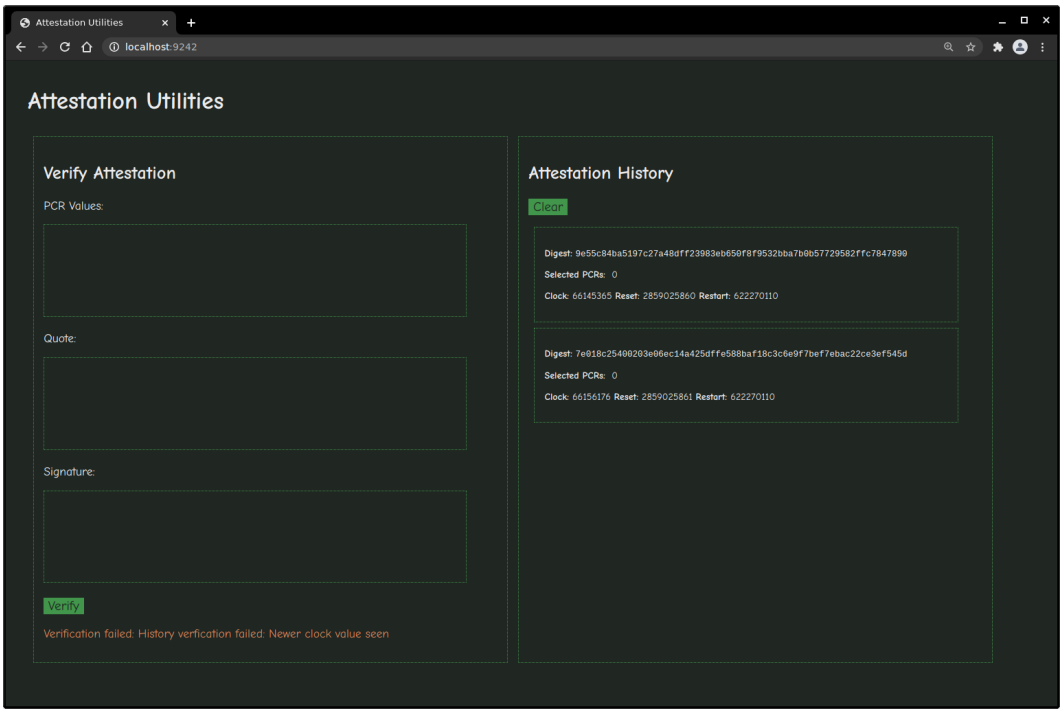


Figure 5.25: A version of the diary implementing a replay attack. The PCR digest looks good, but the verification application realises it has seen newer clock values.

Evaluation

With a system using fTPM on top of seL4, as well as the example use case from the thesis demonstration, we can now compare the solution against the requirements detailed earlier.

6.1 Security

The security requirement is stated as: *The design should follow the current state of the art in security practices, using as many of the available security mechanisms as possible in the correct way.*

For the parts of the design which have been implemented, this has been followed. The latest open source versions of fTPM and OP-TEE are used, fTPM implements as closely as possible the TPM 2.0 standard which is at the time of writing the most recent. The full access to the TPM means that the user can choose the algorithms being used in their TPM command parameters, however in the example setup only secure algorithms are being used (SHA256, 2048-bit RSA).

The only place where the current setup falls short is that there is no way to program the RPMB key into a secure fuse, instead the default derived OP-TEE key is used, which can be calculated from information read from the eMMC's manufacturing registers.

6.2 Performance

The performance requirement is stated as: *The implemented solution should not cause a significant overhead in the boot process and should not impede the operation of the system after the boot process unless specifically invoked.*

The overhead in the boot process was never measured, although we know from fTPM's paper that a firmware TPM is generally faster than a discrete TPM, since it ultimately runs on the main CPU [Raj et. al. 2016]. More importantly however, since fTPM and the secure world can only be passively invoked from the `smc` instruction (when only using fTPM) this solution should not impede the operation of the system after the boot process unless specifically invoked, and thus the latter half of this requirement is met.

However, we cannot make any serious claims about performance without doing a proper benchmark. Since there are no benchmarks of this particular solution, this requirement can only be considered partially met.

6.3 Policy Freedom

The policy freedom requirement is stated as: *The design should not require the system running on top of seL4 to adhere to a particular security policy. Designers of the 'end user' system should be able to freely use the TPM and attestation facilities in whatever way they see fit.*

Since fTPM support is fully implemented, the root server (and any other server with the `smc` capability) is free to use whichever TPM commands they like. This can be used to implement other security systems, or to make variations on the standard attestation process. Thus as far as the TPM is concerned, this requirement is met.

6.4 Verification

The verification requirement is stated as: *While full verification is outside the scope of this thesis, the design and implemented solution should at least be amenable to verification, with a strong specification and relatively small codebase.*

The nature of this solution makes this difficult to achieve. The largest targets for verification here are OP-TEE and fTPM. OP-TEE can in theory be replaced with seL4 in the long run, removing the need to verify it. However, the TPM specification is quite large and complicated, although it includes a full reference implementation.

The `smc` kernel object is difficult to verify without making strict assumptions about the secure world and any services running on top of it, and thus this solution may never be suitable for verification.

6.5 Secure Boot

The secure boot requirement is stated as: *The design should leverage some hardware or firmware functionality to attempt to certify a cryptographic signature for seL4 and the initial task, and should fail to boot if it cannot certify those signatures.*

As secure boot was not part of the final implementation nor part of the example system, this requirement is not met.

6.6 Attestation

The attestation requirement is stated as: *The design should leverage a hardware or firmware TPM to provide remote attestation to a trusted remote verifier, through the use of a CAmkES module.*

Besides not using a CAmkES module, the example system demonstrates this requirement exactly, with an attestation being verified remotely over a serial connection.

6.7 Platform

The platform requirement is stated as: *The implementation should work on any ARM platform which supports some level of secure boot on the firmware and fTPM via. TrustZone.*

Most of the drivers used for the implementation would be platform independent. The exception is the MMC driver, however this follows a standard interface (the RPMB commands). Thus, to port this implementation to another system, one would only need to check for support for fTPM and provide an MMC driver capable of issuing MMC commands. This requirement is met.

6.8 Summary

Requirement	Met?
Security	Mostly
Performance	Somewhat
Policy Freedom	Yes
Verification	No
Secure Boot	No
Attestation	Mostly
Platform	Yes

Table 6.5: Summary of requirements met.

Table 6.5 summarises the degree to which each requirement is met. Four out of seven requirements are at least mostly met, and with some future work it should be possible to meet all requirements by building off this design. Only verification would not be possible to achieve with this design, due to fundamental problems with the `smc` instruction.

Future Work

We will now look at some work for the future: what work would be needed to consider this thesis truly complete, and with the benefit of hindsight some interesting alternative designs.

7.1 Near Future

For the near future, there are some small usability changes to be made to the source code. This includes packaging the libraries as CAmkES modules, and de-coupling the MMC driver, memory allocation, etc. However, the main aspect missing from the solution is secure boot support.

7.1.1 Secure Boot

Although attestation provides similar functionality to secure boot, an ideal system would support some combination of both. Although each stage of the boot process has some support for secure boot, this was never fully looked into. The only difference secure boot should have for seL4 is the quirk that the seL4 image has to be verified in two separate pieces: the kernel and ELF loader, and the root server image. This is because the root server image would likely be signed with a different key (or not at all) to the kernel and ELF loader. Since the way the image is generated depends on the platform, this could require some modifications to U-Boot such as a custom command.

To truly get secure boot working on ARM in the spirit of seL4 and verified software, it would be ideal to port a verified bootloader such as SABLE, however this would require complex changes to the way that SABLE works.

7.2 Far Future

For the far future, during this thesis we have identified some fundamental problems with the approach. These are some different approaches tailored to seL4 which can help address these problems, although they would require relatively more work.

7.2.1 seL4 as a Trusted OS

An ideal secure world operating system should be small enough to fit into secure memory, have strong security guarantees, and a small trusted computing base. This makes seL4 an ideal choice for running as a trusted operating system, and doing so has been proposed before in the seL4 community.

This means that, with a number of other secure world services, seL4 could be configured to replace OP-TEE as the underlying secure world operating system for fTPM as shown in Figure 7.26. This could even evolve into a multikernel-style solution, in which the normal world and secure world kernel are aware of each other's protections, and there are some guarantees that secure world applications cannot interfere with the normal world in unwanted ways.

Such a system is the only real way to make the `smc` instruction compatible with verification, assuming that the secure monitor is trusted that seL4 is in fact being used in both the normal and secure world.

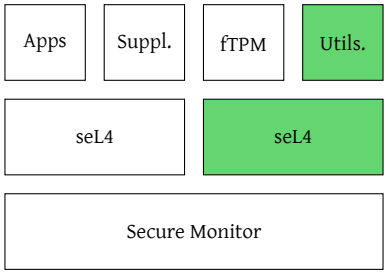


Figure 7.26: How seL4 can work as both a normal-world OS and a secure-world OS, in this case running fTPM in the secure world.

7.2.2 seL4 as a Bootloader

The main motivation behind using TrustZone at all for attestation is that, in a normal system, TrustZone boots before seL4 and can provide isolation before seL4. However, if seL4 was used as a bootloader, seL4 would be able to provide it's own isolation early in the platform initialisation process. This would significantly reduce the reliance on TrustZone. Such a system could also help reduce the security issues faced by bootloaders, by offloading some parts of the boot process into isolated userspace processes.

This would require some modification of seL4, to support a 'boot capability' granted to the root server, as shown in Figure 7.27. This boot capability, when invoked by a thread, would:

1. Unmap all pages except for the pages owned by this thread. Boot images, etc. would need to be copied into this 'booting thread'.
2. Jump to a given address in kernel mode, this address must be mapped in this thread.

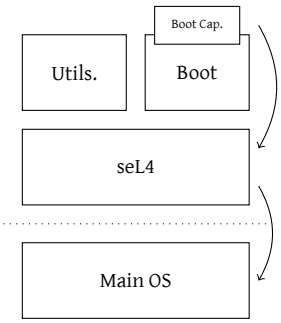


Figure 7.27: How seL4 can be reconfigured to support a bootloader.

Conclusion

To conclude, we have designed and implemented a system which supports attestation on ARM systems using fTPM. While not implementing it in the design, we also speculate that secure boot is compatible with this design. There are however some fundamental issues with the design, and some other areas that could be improved.

When it comes to seL4, while secure boot and TPM attestation are viable means of providing platform security, it could be the case that a fundamentally different approach is better suited. This is particularly evident with fTPM and TrustZone, where it is difficult to justify having an isolated 'secure' environment running on an operating system with *less* security guarantees than seL4. Additionally, we are left wondering why we cannot bring seL4 into the boot process earlier, and rely on software-based isolation earlier in the boot process as opposed to hardware or firmware isolation.

Bibliography

[Ars Technica 2011] Ars Technica. 2011. 4 million strong Alureon P2P botnet "practically indestructible". (July 2011). Retrieved from https://arstechnica.com/information-technology/2011/07/4-million-strong-alureon-botnet-practically-indestructable/.

[Goodin 2010] Dan Goodin. The Register. 2010. World's most advanced rootkit penetrates 64-bit Windows. (November 2010). Retrieved from https://www.theregister.com/2010/11/16/tdl_rootkit_does_64_bit_windows/.

[Rutkowska 2009] Joanna Rutkowska. 2009. Why do I miss Microsoft BitLocker?. (January 2009). Retrieved from http://theinvisiblethings.blogspot.com/2009/01/why-do-i-miss-microsoft-bitlocker.html.

[UEFI 2020] UEFI Forum Inc. 2020. Unified Extensible Firmware Interface (UEFI) Specification Version 2.8 (Errata B).

[TCG 2019] Trusted Computing Group. 2019. Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.59.

[Klein et. al. 2009] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Endelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal verification of an OS kernel. *ACM SOSP '09*, (January 2009), 207–220.

[Heiser 2020] Gernot Heiser. 2020. seL4 Design Principles. (March 2020). Retrieved from https://microkerneldude.wordpress.com/2020/03/11/sel4-design-principles/.

[Shkatov and Michael 2020] Mickey Shkatov and Jesse Michael. 2020. There's a Hole in the Boot. (July 2020). Retrieved from https://eclypsium.com/2020/07/29/theres-a-hole-in-the-boot/.

[Constable et. al. 2018] Scott Constable, Rob Sutton, Arash Sahebolamri and Steve Chapin. 2018. *Formal Verification of a Modern Boot Loader*. Electrical Engineering and Computer Science Technical Reports. Syracuse University, Syracuse, NY.

[Raj et. al. 2016] 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. 2016. fTPM: A Software-Only Implementation of a TPM Chip. *USENIX Security 16*, (August 2016), 841–856.

[Eldefrawy et. al. 2017] Karim Eldefrawy, Norrathep Rattanavipanon and Gene Tsudik. 2017. HYDRA: Hybrid Design for Remote Attestation (Using a Formally Verified Microkernel). *ACM WiSec '17*, (July 2017), 99–110.

[Heiser et. al. 2011] Gernot Heiser, Leonid Ryzhykm Michael von Tessin, and Aleksander Budzynowski. 2011. What If You Could Actually Trust Your Kernel? *USENIX HotOS 13*, (May 2011).

[Tremblay 2021] Kevin Tremblay. 2021. UEFI Signing Requirements. (January 2021). Retrieved from `https://techcommunity.microsoft.com/t5/hardware-dev-center/updated-uefi-signing-requirements/ba-p/1062916`.

[Debian 2021] Debian Wiki. 2021. Secure Boot. (February 2021). Retrieved from `https://wiki.debian.org/SecureBoot#Shim`.

[Garrett 2019] Matthew Garrett. 2019. What does remote attestation buy you?. *Linux Plumbers Conference 2019*, (September 2019).

[Garret 2017] Matthew Garrett. 2017. Avoiding TPM PCR Fragility Using Secure Boot. (July 2017). Retrieved from `https://mjg59.dreamwidth.org/48897.html`.

[Arthur et. al. 2015] Will Arthur, David Challener and Kenneth Goldman. 2015. *A Practical Guide to TPM 2.0*. Apress, Berkeley, CA.

[Chiang 2021] Eric Chiang. 2021. The Trusted Platform Module Key Hierarchy. (January 2021). Retrieved from `https://ericchiang.github.io/post/tpm-keys/`.

[Trustworthy Systems 2020] Trustworthy Systems Team. 2020. *seL4 Reference Manual*. Version 12.0.0.

[VanVossen 2021] Robbie VanVossen. 2021. seL4 RFC-9: Add new capability for seL4 SMC Forwarding. (November 2021). Retrieved from `https://sel4.atlassian.net/browse/RFC-9`.

[OP-TEE 2021] OP-TEE Core Maintainers. 2021. OP-TEE build.git. Retrieved from `https://github.com/OP-TEE/build`.

[Denk 2021] Wolfgang Denk, DENX Software Engineering and U-Boot Maintainers. 2021. Das U-Boot Source Tree. Retrieved from `https://github.com/U-Boot/u-boot`.