Protocols for a Two-Tiered Trusted Computing Base[⋆]

José Moreira[1,2], Mark D. Ryan[1], and Flavio D. Garcia[1]

[1] School of Computer Science, University of Birmingham, Birmingham, UK
{m.d.ryan, f.garcia}@cs.bham.ac.uk
[2] Valory AG, Zug, Switzerland
jose.moreira.sanchez@valory.xyz

Abstract. A *trusted computing base* (TCB) is the minimum set of hardware and software components which are inherently trusted by a platform, and upon which more complex secure services can be built. The TCB is secure by definition, and it is typically implemented through hardened hardware components, which ensure that their secret data cannot be compromised. In this paper, we propose and investigate a two-tier TCB architecture that benefits both from a small hardened 'minimal' TCB, but also offers the possibility of integrating complex security services into an 'extended' TCB. Our design includes a collection of protocols to ensure (1) secure update of the components, (2) secure boot of the platform, (3) attestation, and (4) detection of powerful attackers that can corrupt memory regions together with a (highly probable) platform recovery mechanism after such an attack. The protocols have been formally modelled, and we provide a collection of security properties that have been verified using the automatic protocol verifier ProVerif.

Keywords: Trusted computing base · secure boot · remote attestation · formal modelling.

1 Introduction

A *trusted computing base* (TCB) is the set of software and hardware components of a system which form a trust anchor, and upon which the security of the system relies. Two considerations that influence the design of a TCB appear to oppose each other:

– On one hand, the TCB should be very secure; this means that it should be as small and as simple as possible (since complexity brings insecurity); and it should be strongly isolated from the main system so that a compromise in the main system cannot affect the TCB.

[⋆] We would like to thank the anonymous reviewers for their insightful comments and suggestions. We also gratefully acknowledge EPSRC grants EP/V000454, EP/S030867, EP/R012598, and EP/R008000/1. The majority of the research was done while J. Moreira was at the University of Birmingham.

- On the other hand, the TCB should offer trustworthy services that support the operation of the main system, such as storage and secure usage of cryptographic keys, storage of application-specific secrets, and trusted execution of application-specific code.

In this paper, we investigate how to split the TCB into two parts, a *minimal trusted computing base* (MTCB) providing limited functionalities, but the most hardened services, and an *extended trusted computing base* (ETCB) providing additional functionalities and services that cannot be protected to quite the same extent. This paper proposes a design for a secure architecture of MTCB and ETCB. Our target platform is the TCB for a network device (e.g., a router, modem, or base station). This kind of platform boots infrequently, and hence boot-time checks are insufficient to guarantee security; we also need checks done at run-time. We expect our design may be useful for other kinds of platform too.

Our contributions include:

- A novel two-tiered TCB architecture, achieving high-grade security properties for the core TCB, while also allowing a rich extended TCB to support applications;
- Security analysis of the protocols defined for the TCB architecture, including verification using ProVerif.

2 Background

Trusted Execution Environments (TEEs) such as ARM TrustZone [13], Intel SGX [5], RISC-V Keystone [6, 7], or Sancus [12] realize isolation and attestation of secure application compartments called *enclaves*. TEEs enforce a dual-world view, where even compromised or malicious operating system (OS) in the normal world cannot gain access to the memory space of enclaves running in an isolated secure world on the same processor. This property allows for a TCB reduction: only the code running in the secure world needs to be trusted for enclaved computation results. Thus, such enclave systems offer a great deal of flexibility when it comes to defining the specific code and services that can be executed. However, such flexibility usually comes at the price of an increased surface of attack which gives rise to well-known microarchitectural attacks such as cache timing in TruSpy [21], ARMageddon [8] and Cachezoom [10], or speculative execution based attacks in Foreshadow [19], ZombieLoad [16], SgxSpectre [4], CrossTalk [14], among many others. Besides microarchitectural attacks, rich TEEs often require complex interaction with the insecure world, which leads to 'Tale of two worlds' type of attacks [20].

On the other hand, fixed-API devices such as the Trusted Platform Module (TPM) [18] or the Google Titan M/C chips [15] have a significantly reduced TCB, compared to enclave systems. However, they are typically low performance devices, hard to update and cannot easily support customized applications.

Minimization of the TCB is one of the key principles for secure systems design. A two-tiered TCB has the potential to get the best of both worlds:

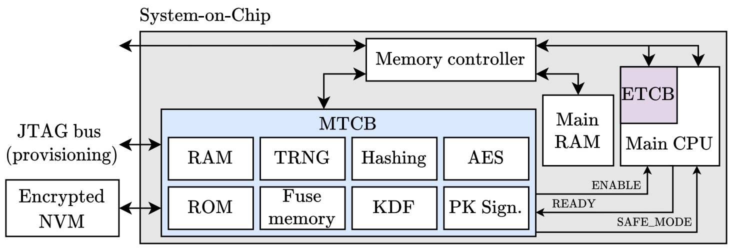

Fig. 1. Two-tier TCB architecture

a small but well protected TCB to guard high secrets which are seldom used and a larger, more feature rich and fast TCB to protect medium secrets. The extended TCB can use the minimum TCB as a trust anchor for long term key storage, integrity protection, etc.

3 Design of the TCB

We assume that the system processes assets which may be categorised as high-value (e.g., long-term keys), medium-value (e.g., ephemeral session keys), and low-value (e.g., transient data). The core functionality of the MTCB is to protect high-valued assets from strong adversaries, and provide the ultimate trust anchor for the system. In order to achieve this, we propose that the MTCB be implemented on a tamper-resistant, discrete processor with its own memory (drawing inspiration from OpenTitan). The MTCB can be integrated within the same SoC as the main processor, RAM and memory controller, in order to avoid bus probing attacks between separate components [17], but with its own, physically separated RAM, and ROM. Having its own memory, the MTCB is isolated from the main processor and thus immune to side channel leakage, e.g., through cache attacks.

Fig. 1 depicts our simple MTCB-based platform architecture. This discrete chip implements basic cryptographic primitives including hashing and public key signatures with state-of-the art countermeasures against side-channel and fault-injection attacks. We discuss the offered functionalities below.

3.1 Main Functionalities

The MTCB implements a very delimited set of services, built upon a set of main functionalities, which can be summarized as follows:

Secure boot. This functionality ensures that the device boots only if it can ascertain that the software being booted is the pre-registered one (e.g., through an enrolment process). We use secure boot to launch the main processor. At boot time, the main processor loads the ETCB in the predefined memory address and then halts. The MTCB boots and measures the ETCB by hashing the ETCB code and comparing the resulting hashes with the expected values, which are stored in an external non-volatile memory (NVM). If this check is successful, then the MTCB enables the main CPU to continue booting. The ETCB then takes control of the boot sequence. Now the ETCB will measure the relevant parts of the OS and compare these with the expected hashes. Communication between the MTCB and the ETCB occurs using their shared memory.

Attestation. This functionality allows a remote party (such as the device owner) to obtain a statement signed by the MTCB about the MTCB's own state (including its firmware version), and the currently loaded ETCB and some aspects of the system software. The MTCB stores the attestation key and its corresponding certificate (chain). Attestation requests are received by system software, and sent from there via the ETCB to the MTCB. Such attestation requests include a challenge from the verifier such as a nonce. Upon receipt of an attestation request the MTCB will produce a signed statement of the challenge, its own state, firmware version, etc., plus the hash of the ETCB code which is currently loaded in memory. Additionally, it includes the hashes provided by the ETCB of the relevant system components which need to be attested.

ETCB Recovery. This functionality aims to identify a memory corruption situation where the adversary has gained enough control over the platform so that it can change the memory contents of the ETCB and substitute it by its own, malicious version. The MTCB proactively measures the running ETCB code, and forces a reboot to a safe state if it finds an unexpected measurement.

3.2 Auxiliary Functionalities

In order to implement the above core security functionalities in practice, we require the following functionalities as well:

Measurement of the ETCB. As part of three of the security protocols, namely secure boot, attestation and ETCB recovery (discussed in Sec. 5 below), the MTCB has the ability to access the main processor's RAM in read-only mode through the memory controller, providing the best of both worlds. This design feature allows performing measurement of the ETCB which runs within the TEE and at a fixed memory address in RAM, e.g., at the very beginning. In order to prevent TOCTOU type of issues generated due to the asynchronous access to the main RAM, the MTCB is set as master of the memory controller. During measurement of the ETCB the MTCB simply disables write access to the code area of the ETCB from the main processor.

Measurement of the OS. Similarly, the ETCB is able to read and measure the memory region where the OS is loaded. We propose that the ETCB enclave system follows an architecture similar to RISC-V Keystone [6, 7] or ARM Trust-Zone [13], where the ETCB can take control of the boot sequence and instruct the CPU to execute the OS if the measurements match the expected values.

3.3 Description of the Architecture

As discussed above, the MTCB (and its RAM and ROM) are integrated within the same SoC as the main processor, and there is also RAM shared by the MTCB and the main processor. We discuss the remaining components of the proposed architecture from Fig. 1.

Provisioning bus. The MTCB must expose an interface (e.g., SPI, I2C or JTAG) for firmware provisioning at manufacture time. During the provisioning process, the MTCB will read contents from that interface and write it to its NVM. After finalizing the provisioning process, the MTCB will permanently disable the provisioning interface, preferably, at hardware level by configuring the appropriate fuse bits.

Random number generator. In order to guarantee the generation of high-entropy cryptographic material, we require that the MTCB be provided with a hardware true random number generator (TRNG), which generates random bits from a physical process.

Fuse memory. The MTCB has an internal fuse memory, which stores:

– A persistent secret key that is generated on first boot used to encrypt and authenticate the external NVM.
– CTR_CUR_VERSION: A unary counter that keeps track of the currently installed version. Each time the MTCB updates its code, it increments this value to match the installed version number.
– CTR_SAFE_MODE: A unary counter that stores the signal SAFE_MODE between reboots: an odd value of the counter signifies that the MTCB must activate the SAFE_MODE signal when the platform boots. See Sec. 5.4 below for more details.

External NVM. The MTCB requires persistent secure storage (EEPROM) to store highly-valued crypto objects and code. The semiconductor manufacturing process does not allow the integration of mixed process sizes within the same SoC, e.g., 7 nm and 14 nm processes. Hence, if current EEPROM technology has a different process size than the main processor, then they cannot be integrated within the same SoC. In order to circumvent this limitation, we implement the MTCB NVM as an external EEPROM chip connected directly to the MTCB via SPI. Because the external NVM could potentially be accessed by a physical attacker, the contents of the EEPROM are encrypted and authenticated. Upon booting of the MTCB, the MTCB BootROM verifies the integrity of the memory blob from the EEPROM, authenticates and decrypts it, using the secret key persistently stored in fuse memory. Then the MTCB BootROM verifies its version number against the unary counter CTR_CUR_VERSION held in the internal fuse memory, and then loads the firmware onto the MTCB's RAM. Conversely, with every write operation, both the NVM version number and the unary counter held in fuse memory are incremented and the NVM re-encrypted and authenticated. In order to prevent rollback attacks, it is necessary to increment the NVM version number with each state change, which has the drawback of consuming one fuse per update. Fortunately, our MTCB does not require frequent, persistent state changes.

Crypto Primitives. The MTCB contains a number of basic crypto primitives (hashing, symmetric encryption, key derivation), implemented as ASIC blocks, which are required to enable the services it offers.

Table 1. Two-tiered TCB design requirements

	Requirement	Realization
MTCB	Resistance to cryptography compromise	Firmware is updatable to enable new cryptographic algorithms
MTCB	Bricking avoidance	A/B updates
MTCB	Resistance to micro-architectural attacks	Separate processor, avoiding sharing resources such as cache
MTCB	Resistance to physical attacks (fault injection, side channel)	Separate processor, implements countermeasures, TRNG
MTCB	Resistance to physical attacks (bus probing)	SoC integration with the main CPU
MTCB	Resistance to chip decapsulation (confidentiality)	Self-destructing tamper resistance
MTCB	Resistance to chip decapsulation (integrity)	Usage of fuse memory for counters and root firmware verification key
MTCB	Usage of high entropy cryptographic material	TRNG
MTCB	Attestation	Hashing and public-key signing scheme
MTCB	Evolvability	External NVM for counters and root firmware verification key, and fuse memory
MTCB	Rollback protection	Fuse memory
ETCB	Resistance to software attacker, e.g., buffer overflow attacker	Has enclave system with memory integrity protection (authenticated)
ETCB	Keystore for protected secrets	Has enclave system
ETCB	Resistance to run-time memory corruption	Periodic run-time memory measurements by MTCB; ETCB recovery protocol; SAFE_MODE
ETCB	Attestation	Attestation of ETCB by MTCB

Table 1 mentions some architecture requirements of the TCB and the features that realize them.

4 Adversarial Model

The objective of this section is to define a realistic adversary for the architecture presented above. The adversary can send and receive messages to some platform components (see below). If it learns keys or defines new keys, it can apply cryptographic operations using those keys. To the extent that it has the appropriate keys, the adversary can intercept and spoof messages between components of the system. These aspects of the adversary model are sometimes called the Dolev-Yao model. Later in this section, we specify other aspects of the adversary model, such as the ability to corrupt memory.

Ideally, one would like to define the strongest conceivable adversarial model, since it is clear that if a security property holds for a such a model, it will automatically hold for a relaxed version of that adversary (i.e., having a subset of capabilities). However, imposing a too strong adversary will simply make it impossible that the protocol satisfies any non-trivial security property. For example, if we allow the adversary to have unrestricted control over the exchanged messages by any party, unrestricted capability to change the platform memory, and the ability to anticipate any MTCB operation, it will be impossible to prove attestation of platform state: the adversary can simply switch the memory contents to a legitimate ETCB and OS just before the MTCB is going to attest the memory contents, and switch back to the malicious version afterwards.

Therefore, in order to come up with a realistic adversary, the following reasonable considerations have been taken for our modelling of the protocols:
Communication channels. The adversary has unlimited read/write access to: 1. the Vendor-Platform channel, 2. the Verifier-Platform channel, 3. the ETCB-OS channel, 4. the MTCB-ETCB channel. However, we require that the adversary has the following restrictions:

1. no write access to the provisioning channel before initial manufacturer provisioning,
2. no read/write access to the MTCB-ETCB channel during first boot,
3. no write access to the signals (ENABLE, READY, SAFE_MODE) exchanged between the MTCB and the main CPU.

We note that making all those channels available to the adversary might be an over-pessimistic assumption (as it might be unrealistic that it has access, e.g., to the MTCB-ETCB channel, which is within the same tamper-resistant SoC), but we can still prove our desired security properties under this assumption. This means that the security properties are anchored on the secrets held by the different participants in the protocol, and in the root of trust implied by the MTCB, but not in the fact that a certain communication between parties is made unavailable to the adversary.
Integrity. We assume that the MTCB is a root of trust for the platform, its integrity is guaranteed, and its secrets are not leaked to the adversary. We also assume that the Vendor secrets are not leaked.
Initial platform state. The adversary can freely choose an initial configuration for the ETCB and OS at each boot, possibly a malicious ETCB and/or OS.
Memory corruption. The adversary can change ETCB and OS memory regions after they have been loaded by the CPU. We do not differentiate whether this can be achieved through a bug present in a faulty (but legitimate) ETCB, or through some sort of fault-injection or physical vulnerability. Our modelling, discussed below in Sec. 6, allows arbitrary change between legitimate and non-legitimate ETCBs at any time. However, in a real scenario, it is reasonable to consider that when the adversary switches the memory to a legitimate ETCB,

then it cannot longer regain control easily. We also consider that if the adversary succeeds in corrupting the memory to a rogue ETCB version, then it will be interested in running it for a non-negligible fraction of time.
Anticipation of MTCB operations. To combat memory corruption attacks, we introduce some MTCB operations that aim to detect them. We assume that appropriate protections are in place to prevent anticipation of those MTCB operations. Alternatively, we can assume that the adversary is able to anticipate MTCB operations, without guaranteeing that it will have enough time to hinder their effect. E.g., an adversary could anticipate a memory measurement, but it may not have enough time to revert the memory contents to the uncorrupted state. These assumptions are reasonable for a realistic adversary.
Denial of service attacks. As usual in the symbolic model of cryptography, it is impossible to prove "non-DoS" properties, because a Dolev-Yao adversary can drop messages indefinitely. For instance, it is impossible to prove that a Verifier eventually gets a valid attestation. However, it is possible to prove that when a Verifier is convinced that a received attestation is genuine, it is indeed the case.

Some further considerations about adversarial modelling will also be discussed below in Sec. 6.

5 Protocols

The two-tier TCB architecture comprises a set of four core security protocols designed in order to achieve secure firmware update, secure platform boot, platform attestation, and ETCB recovery of a corrupted system. Additional custom services can be build through proper customisation of the ETCB. We start by providing a description of the four core protocols investigated from an implementation point of view. That is, without taking into account the modelling approaches that will be discussed in Sec. 6 below. These protocols are designed to work in conjunction, as the security guarantees of a certain protocol might depend on the establishment of a certain parameter on a previous protocol.

There are a total of seven parties involved in the protocols, although not all of them take part in all protocols, namely:

- **MTCB:** The most protected part of the system, running on a dedicated processor mounted in the SoC, which is a small processor separate from the main CPU.
- **ETCB:** An enclave system, such as ARM TrustZone [13], Intel SGX [5] or RISC-V Keystone [6, 7], running on the main CPU on the SoC.
- **CPU:** The main processor on the SoC. It runs the ETCB in an enclave system, and it runs the OS.
- **OS:** The operating system running on the CPU.
- **Verifier:** A remote party interacting with the system. The verifier can send messages to the system (such as requests for attestation) and receives and verifies the replies.
- **Vendor:** The maker of the system, which installs the firmware and the keys.

Table 2. Glossary of symbols, representing cryptographic objects used throughout the paper.

	Object	Description
Vendor	ssk_V	Secret signing key used for MTCB A/B firmware update.
	spk_V	Public verification key corresponding to ssk_V.
	σ_V	*A signature using ssk_V.
	kfw	Symmetric encryption key, shared with the MTCB, for code confidentiality.
MTCB	id_M	Unique identifier for a particular MTCB instance.
	$code_M$	Manufacturer-supplied MTCB code.
	ver_M	MTCB code version.
	ptr	Firmware pointer (either NVM region A or B).
	lts_{ME}	Long-term secret, shared between MTCB and ETCB, established on first boot.
	bs_{ME}	*Boot secret, shared between the MTCB and the ETCB, established on each boot through the AKEP2 subprotocol.
	k_{MAC}	*MAC key, shared between the MTCB and the ETCB, used in Protocol 3 (attest.).
	ssk_M	Secret signing key used for attestation.
	spk_M	Public verification key corresponding to ssk_M.
	σ_M	Signature using ssk_M.
ETCB	id_E	Unique identifier. See remark on Sec. 5.2 below.
	$code_E$	Adversary-supplied ETCB code.
	ver_E	Version.
	h_E	*Currently loaded ETCB measurement, i.e., $h_E = h(code_E)$.
	h_E^{ref}	Reference ETCB measurement.
OS	$code_O$	Adversary-supplied OS code.
	h_O	*Currently loaded OS measurement, i.e., $h_O = h(code_O)$.
	h_O^{ref}	Reference OS measurement.

All objects (except those marked with *) are stored persistently, e.g., NVM, fuse memory, or enclave secure store. Objects marked with * are kept in RAM while needed.

– **Adversary:** An agent that tries to circumvent the secure operation of the system. The adversary's capabilities are defined in Sec. 4.

There are many objects held or exchanged by the different parties. For convenience, in Table 5, we provide a glossary of the symbols used to represent those objects throughout the paper.

5.1 Protocol 1: MTCB A/B Update

Protocol 1 implements over-the-air (OTA) MTCB code updates. In order to ensure that a workable booting system remains on the MTCB NVM space during the update process, we implement "A/B updates," in which there are two slots that can contain the MTCB code, called A and B. This approach reduces the likelihood of an inactive or "bricked" device, should the update process be interrupted for any reason. If this occurs, the MTCB would boot on the non-updated version again. In the execution of this protocol, there are three elements that are updated in the MTCB NVM: the version number ver_M, the code itself $code_M$, and the Vendor public signing key spk_V. For clarity, we parametrize these three elements using a bracketed index indicating a specific update, e.g., $code_M[i]$ and $code_M[i+1]$ denote the code contents of two consecutive updates.

Let us consider that the current status of the MTCB corresponds to the ith update. When the platform boots, the MTCB boot pointer ptr indicates what region of the NVM contains $code_M[i]$ (either region A or B), and the MTCB BootROM loads it. Also, we assume that the system currently has a legitimate MTCB, and as such, it implements the A/B update protocol. More concretely, all legitimate and signed $code_M$ implements the A/B update protocol. The sequence of steps to achieve the next, $i+1$, A/B update is as follows:

1. The Vendor updates its signing keypair $ssk_V[i+1], spk_V[i+1]$ (or copies the previous one), generates the updated $code_M[i+1]$, and increments the version number $ver_M[i+1]$, for example, $ver_M[i+1] = i+1$. Then, it produces the tuple
$$(ver_M[i+1], \mathrm{enc}_{kfw}(code_M[i+1]), spk_V[i+1]), \tag{1}$$
and signs it with the current signing key $ssk_V[i]$, obtaining the signature $\sigma_V[i+1]$. It outputs publicly the tuple (1) and $\sigma_V[i+1]$.
2. The MTCB receives the secure update command together with the tuple (1) and the signature $\sigma_V[i+1]$, and it proceeds as follows:
3. Verify $\sigma_V[i+1]$ with currently installed $spk_V[i]$.
4. Check that $ver_M[i+1] > ver_M[i]$.
5. Decrypt $code_M[i+1]$ using kfw.
6. Copy the decrypted contents $code_M[i+1]$ and updated signing key $spk_V[i+1]$ at the complementary NVM location (which is B if the MTCB is currently executing from A, and is otherwise A).
7. Check the hash of written contents.
8. Update the stored version number.
9. Change the boot pointer to the complementary NVM location and reboot.

Fig. 2 briefly depicts the NVM state in the different stages of the A/B update. Note that the proposed design does not allow non-consecutive updates if there is a change of the Vendor signing keypair in between. That is because in order to verify the signature $\sigma_V[i+1]$, it has to match the corresponding signing key $spk_V[i]$. It could be argued that there are two types of MTCB updates: minor, where the signing key does not change, and major, where the signing key changes. For simplicity, and due to the fact that the MTCB is expected to execute a limited number of updates during its lifetime, we only consider major updates in our analysis. This can also be enforced at Step 4 by checking whether $ver_M[i+1] = ver_M[i] + 1$.

Also, note that the current version number ver_M is stored in unary notation in fuse memory as the counter CTR_CUR_VERSION. This removes the need to

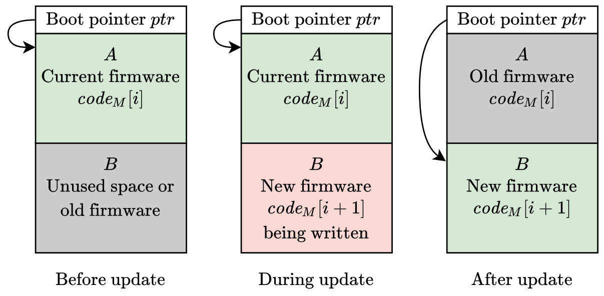

Fig. 2. Protocol 1: NVM stages in the A/B update

store the boot pointer: the MTCB BootROM can simply execute code stored in, e.g., NVM region A if ver_M is even, and NVM region B if ver_M is odd.

Observe that we do not allow for the code encryption key kfw to be updated. This is due to the fact that there is not much gain in doing so: if this key is compromised at any point in time, all future versions of this key will be compromised as well.

Finally, we require that the reference ETCB measurement h_E^{ref}, used in the secure boot protocol (see Sec. 5.2 below) has the same level of protection in the NVM as $code_M$ has. That is, integrity, confidentiality and rollback protection. The reason for this is that otherwise an attacker can do a rollback attack if it can corrupt the NVM memory. As a result, an update in the ETCB requires an update in the MTCB.

5.2 Protocol 2: Secure Boot

The goal of secure boot is for the MTCB to validate the integrity and trustworthiness of the ETCB and OS code before the platform executes them, so that it can ensure it starts with an expected, legitimate combination of ETCB and OS. Therefore, the MTCB has to be regarded as a root of trust of the whole system, and the security of this protocol (and indirectly that of the remaining ones) relies on this assumption. Hence, in our analysis, we consider that both $code_E$ and $code_O$ are freely chosen by the adversary, but the MTCB firmware $code_M$ is unconditionally trusted.

This protocol uses the AKEP2 protocol [1] in order to derive a shared boot secret bs_{ME}. A long-term secret lts_{ME} shared between the ETCB and the MTCB is required, which was established on the first boot at manufacture time. We abstract away from the specific enclave system used by the ETCB, and we assume that it provides long-term secure storage for its internal secrets. Note that the ETCB does not have a persistent identity; the platform's persistent identity is given by the MTCB identifier id_M. Nevertheless, for the AKEP2 protocol, the ETCB is required to have an identity, which we define as $id_E = h(lts_{ME})$, for some secure hash function h.

Fig. 3 depicts this protocol. The sequence of actions is as follows:

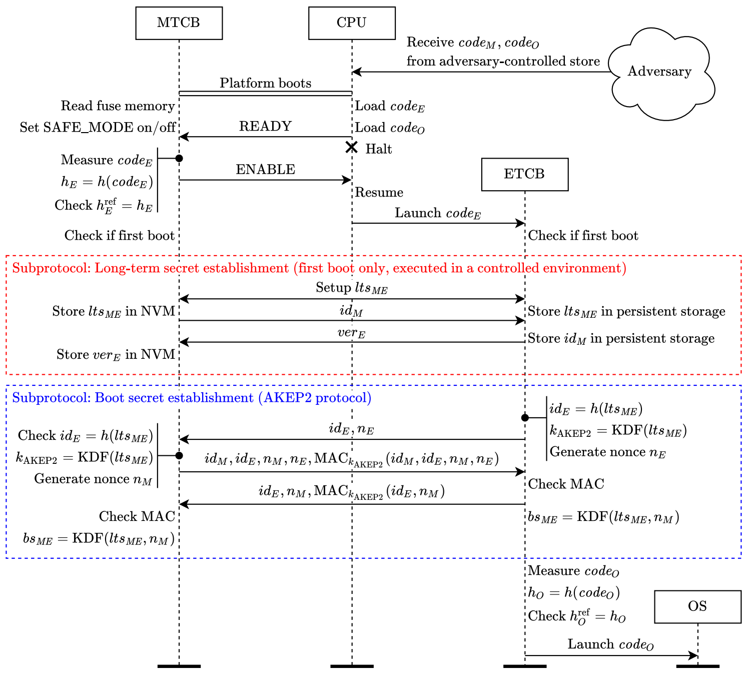
MTCB
CPU
Adversary
Receive $code_M, code_O$ from adversary-controlled store
Platform boots
Read fuse memory
Set SAFE_MODE on/off
Load $code_E$
Load $code_O$
READY
Halt
Measure $code_E$
$h_E = h(code_E)$
Check $h_E^{\text{ref}} = h_E$
ENABLE
Resume
ETCB
Launch $code_E$
Check if first boot
Check if first boot
Subprotocol: Long-term secret establishment (first boot only, executed in a controlled environment)
Setup lts_{ME}
Store lts_{ME} in NVM
id_M
Store lts_{ME} in persistent storage
ver_E
Store id_M in persistent storage
Store ver_E in NVM
Subprotocol: Boot secret establishment (AKEP2 protocol)
$id_E = h(lts_{ME})$
$k_{\text{AKEP2}} = \text{KDF}(lts_{ME})$
Generate nonce n_E
id_E, n_E
Check $id_E = h(lts_{ME})$
$k_{\text{AKEP2}} = \text{KDF}(lts_{ME})$
Generate nonce n_M
$id_M, id_E, n_M, n_E, \text{MAC}_{k_{\text{AKEP2}}}(id_M, id_E, n_M, n_E)$
Check MAC
$id_E, n_M, \text{MAC}_{k_{\text{AKEP2}}}(id_E, n_M)$
$bs_{ME} = \text{KDF}(lts_{ME}, n_M)$
Check MAC
$bs_{ME} = \text{KDF}(lts_{ME}, n_M)$
Measure $code_O$
$h_O = h(code_O)$
Check $h_O^{\text{ref}} = h_O$
OS
Launch $code_O$

Fig. 3. Protocol 2: Secure boot

1. The adversary freely chooses ETCB and OS code to be loaded at boot time, namely $code_E$ and $code_O$, and forwards it to the platform.
2. The platform boots. The MTCB checks the counter CTR_SAFE_MODE in fuse memory. If the counter has an odd value, it sets the signal SAFE_MODE to true.[3]
3. The main CPU has pre-boot ROM called BootROM Secure Boot Code (BSBC). This code loads $code_E$ and $code_O$ into memory, signals the MTCB using the signal READY, and then halts the CPU.
4. The MTCB disables the CPU by setting the signal ENABLE to false. Then, it reads $code_E$, which is located at a fixed, predefined memory location known to BSBC and MTCB, and obtains the measurement hash $h_E = h(code_E)$.
5. The MTCB compares the measurement h_E with an expected, reference value h_E^{ref}. If these values match, it notifies the CPU to continue by setting the signal ENABLE to true.
6. Upon receiving the signal from the MTCB, the CPU launches the ETCB code, $code_E$.
7. Only on first boot, the MTCB and the ETCB establish a long-term secret lts_{ME}. The first boot is assumed to happen in a controlled environment, outside the reach of any adversary. The MTCB stores lts_{ME} and ver_E in

[3] The SAFE_MODE signal does not play a role in Protocol 2, but it is used by Protocol 4 (ETCB recovery) *after* secure boot has finished. See Sec. 5.4 for more details.

its encrypted NVM. The ETCB stores lts_{ME} and id_M in encrypted form in untrusted storage.

8. The MTCB and the ETCB establish a boot secret bs_{ME} by executing the AKEP2 protocol. This will be required by Protocol 3 below.
9. The ETCB reads $code_O$, which is located in a predefined memory location known to ETCB and BSBC, and obtains the measurement hash $h_O = h(code_O)$.
10. The ETCB compares the measurement h_O with an expected, reference value h_O^{ref}. If these values match, it launches the OS code, following an approach similar to RISC-V Keystone [6, 7] or ARM TrustZone [13].

We remark that the ETCB and OS measurements carried out at Steps 4 and 9, respectively, must take into account all the data that is expected to remain immutable (e.g., keys and version numbers).

5.3 Protocol 3: Remote Attestation

Remote attestation concerns the reporting of the current platform state (e.g., hardware and software configuration) to an external entity (Verifier). The goal of the protocol is to enable the Verifier to determine the level of trust in the integrity of the platform, that is, that the platform runs a legitimate combination of ETCB and OS. The security guarantees of remote attestation, in general, are limited to state that "at some point in time between the attestation request and its reception, the platform was running with the attested configuration."

In order to prevent the adversary from executing an attack by reusing messages from an earlier boot instance, the communication between the MTCB and the ETCB is authenticated (integrity-protected) through a MAC. Therefore, the protocol uses the shared boot secret bs_{ME} established in the secure boot protocol (Sec. 5.2 above) to derive the MAC key.

Protocol 3 is depicted in Fig. 4, and the sequence of actions is as follows:

1. The Verifier generates a challenge *chal* and forwards it to the OS.
2. The OS forwards the challenge to the ETCB.
3. The ETCB and MTCB derive a MAC key k_{MAC} using the boot secret bs_{ME}.
4. The ETCB reads $code_O$, which is located in a predefined memory location, and obtains the measurement hash $h_O = h(code_O)$.
5. The ETCB forwards *chal*, h_O, and $\text{MAC}_{k_{\text{MAC}}}(chal, h_O)$ to the MTCB.
6. The MTCB checks $\text{MAC}_{k_{\text{MAC}}}(chal, h_O)$.
7. The MTCB reads $code_E$, which is located in a predefined memory location, and obtains the measurement hash $h_E = h(code_E)$.
8. The MTCB signs the tuple
$$(id_M, chal, h_O, h_E) \tag{2}$$
with the attestation signing key ssk_M, obtaining the signature σ_M.
9. The tuple (2) together with its signature σ_M is forwarded back to the ETCB, OS and Verifier.

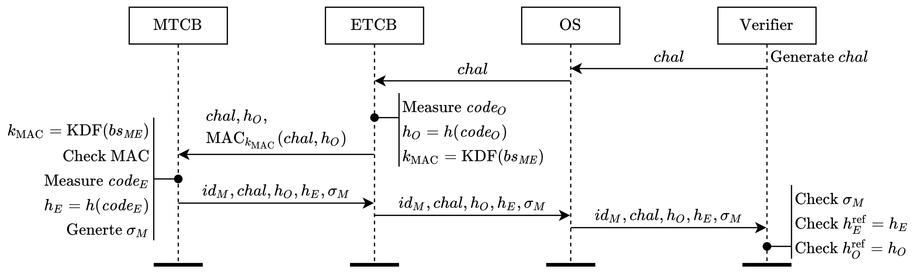
MTCB
ETCB
OS
Verifier
$chal$
$chal$
Generate $chal$
Measure $code_O$
$h_O = h(code_O)$
$k_{\mathrm{MAC}} = \mathrm{KDF}(bs_{ME})$
$chal, h_O,$
$\mathrm{MAC}_{k_{\mathrm{MAC}}}(chal, h_O)$
$k_{\mathrm{MAC}} = \mathrm{KDF}(bs_{ME})$
Check MAC
Measure $code_E$
$h_E = h(code_E)$
Generte σ_M
$id_M, chal, h_O, h_E, \sigma_M$
$id_M, chal, h_O, h_E, \sigma_M$
$id_M, chal, h_O, h_E, \sigma_M$
Check σ_M
Check $h_E^{\mathrm{ref}} = h_E$
Check $h_O^{\mathrm{ref}} = h_O$

Fig. 4. Protocol 3: Remote Attestation

10. Using the attestation public signing key spk_M, the Verifier checks that σ_M is a valid signature for the received tuple (2).
11. The Verifier compares the measurements h_E and h_O with the expected, reference values h_E^{ref} and h_O^{ref}, respectively. If these values match, the Verifier declares a successful attestation.

Observe that, for simplicity, we assume that the MTCB attestation keypair ssk_M, spk_M is persistent. This can be an issue from the point of view of privacy, as this keypair uniquely identifies the platform. A possible alternative to overcome this problem is to use an approach similar to that used by TPM remote attestation. The TPM uses a master key (*endorsement key* in TPM's terminology), which is used to decrypt arbitrarily many attestation keys as desired, which have been certified through a Privacy Certification Authority, or through the Direct Anonymous Attestation protocol. Obviously, this would require to remove id_M from (2). As a result, from the Verifier's perspective, they will know that "a platform has successfully been attested" without knowing the precise identity of that platform. This arrangement is important for TPMs because they are deployed in personal laptops, where privacy is an important issue. However, it may not be needed in network infrastructure devices or similar scenarios.

5.4 Protocol 4: ETCB Recovery

This protocol aims to identify a memory corruption situation where the adversary has gained enough control over the platform so that it can change the memory contents of the ETCB and substitute it by its own, malicious version. The MTCB proactively measures the running ETCB code, and forces a reboot to a safe state if it finds an unexpected measurement.

This protocol can be divided into two processes: corruption detection and recovery. We assume that the platform is already booted, and that the adversary had freely chosen the ETCB and OS code to be loaded at boot time, namely $code_E$ and $code_O$. Also, at any point in time, the adversary might be able to corrupt the RAM location where $code_E$ is stored. See the considerations about the adversary in Sec. 4. The sequence of actions for this protocol is as follows:

(a) Memory corruption detection:

1. The MTCB periodically reads $code_E$, which is located in a predefined memory location, and obtains the measurement hash $h_E = h(code_E)$. We can set h_E to an undefined value if the MTCB is unable to conduct this measurement, e.g., if the attacker is blocking the channel.
2. The MTCB compares the measurement h_E with an expected, reference value h_E^{ref}. If these values do not match, it increments the counter CTR_SAFE_MODE in fuse memory to an odd value, indicating that the signal SAFE_MODE is set, and reboots the platform.

(b) Recovery:

3. The platform reboots. The MTCB checks the counter CTR_SAFE_MODE in fuse memory. If the counter has an odd value, it sets the signal SAFE_MODE to true.
4. If the main CPU is booted with SAFE_MODE set to true, the firewall is configured to allow outgoing connections only. This aims to prevent the ETCB/OS from becoming immediately compromised again. The ETCB/OS attempts to report the security violation to the cloud service.
5. The OS downloads a new signed version of the ETCB code. That is, a tuple $(ver'_E, code'_E)$ with a Vendor signature σ_V.
6. The OS forwards the tuple and signature to the MTCB (via the ETCB).
7. The MTCB checks the Vendor signature σ_V, and checks that the received ETCB version is strictly larger than the current one, i.e., $ver'_E > ver_E$.[4]
8. The MTCB updates the reference measurement for the MTCB as $h_E^{\text{ref}} = h(code'_E)$, and stores ver'_E in its encrypted NVM.
9. If the MTCB has succeeded in downloading and installing $code'_E$, it increments the counter in fuse memory to an even value, indicating that the signal SAFE_MODE is clear, and reboots the platform. Otherwise, it remains indefinitely in safe mode.

This protocol is not guaranteed to succeed, for a number of reasons: First, a powerful adversary could anticipate the moment that the measurements will occur, and take action to avoid detection. Second, even if detected, the replacement of the ETCB might not prevent further attacks. And third, the adversary might block fetching of $code'_E$.

However, we show that the protocol will succeed with probability arbitrarily close to 1 under some reasonable assumptions; see Sec. 4 above. The MTCB implements protections so that the adversary cannot always anticipate the MTCB memory measurements, and it can also identify whether its measurements are being prevented or delayed. Also, the adversary is interested in keeping a rogue ETCB version in memory for a non-zero fraction of time. This excludes improbable corner cases, for example, an adversary running a malicious ETCB for a

[4] From the point of view of security, the most conservative approach is to require that $ver'_E > ver_E$. However, this has the downside effect that if there is no new ETCB version available, the platform would remain in safe mode (inoperative) indefinitely. To avoid this situation, we could relax this check and only require that $ver'_E \geq ver_E$. This is justified if there is a significant cost of time and resources to the adversary to mount the attack again.

very small fraction of time between secure boot and the first MTCB memory measurement, and then switching to a legitimate ETCB the rest of the time. We note that the adversary is free to arbitrarily change the memory at any point, which is probably an overestimation of its capabilities. Nevertheless, our probabilistic argument works as detailed below, even with this overestimation.

Although the adversary is interested in running a rogue version of the ETCB, it is also forced to switch the memory contents (to a legitimate ETCB) so that the MTCB produces an expected measurement and does not trigger the recovery procedure. Hence, there is a trade-off between the adversary spending enough time running the rogue version and the possibility of being detected: too much time spent by the rogue ETCB will increase the chances of the MTCB in identifying the attack, whereas too much time spent by the legitimate ETCB in memory will restrict its malicious abilities. Consider the following given parameters:

- T: time interval during which the platform is active, $T > 0$,
- p: minimum proportion of active time that the adversary has the rogue ETCB in memory, $0 < p \leq 1$,
- ϵ: target error probability, i.e., maximum admissible error of not identifying the attack occurring while the platform is active. We can take this parameter as small as desired.

We are interested in finding the frequency f, i.e., the number of memory measurements by the MTCB per unit of time, so that the actual error probability does not exceed the target ϵ. That is, we want that the probability of failing to identify an attack occurring in T, which is $(1-p)^{fT}$, does not exceed ϵ. It is straightforward to see that for any choice of T, p the target error is satisfied for any $f \geq f_0$, with

$$f_0 = \frac{\log \epsilon}{T \log(1-p)}.$$

As expected intuitively, the required frequency of measurements increases for $\epsilon \to 0$, and $p \to 0$. Also, note that $f_0 \to 0$ for $p \to 1$, i.e., a single measurement is sufficient if the adversary always keeps its malicious version in memory.

6 Modelling and Verification of Security Properties

We have verified the core security properties of our TCB design using ProVerif [3, 2], which is a tool for automated analysis of security properties in cryptographic protocols. ProVerif analyses our pseudocode for Protocols 1–4, and determines whether the security properties we specify hold or not. This is an excellent tool for uncovering design errors, because ProVerif explores all permitted actions of the adversary, and reports potential attacks if such attacks exist. It is a valuable tool for the development of protocols. Nevertheless, it should be remembered that even if ProVerif shows that all the properties are satisfied, this does not mean that the system is secure.

Further details, including the formal description of the security properties can be found in Appendix A. ProVerif is successful in proving the set of security properties for the protocols discussed. The source code with the models, the security properties and a collection of sanity check queries can be found in [11].

7 Conclusion

We have motivated and described our design for a two-tiered TCB, which is targeted at network infrastructure devices such as routers and modems. It aims to provide a small and hardened "minimal" TCB that is assumed secure, but is nevertheless updatable if it turns out insecure. This MTCB is rather inflexible, however, because of its small size and minimal size and strong isolation from the rest of the system. The second tier is a bigger "extended" TCB that offers application-specific services, and is more flexible, while not offering quite such rigorous security because it runs on the same processor as potentially untrusted code. The ETCB is also updatable.

Designing such a two-tiered TCB led us to many design decisions and intricate protocols in order to get the two parts to work together securely. In arriving at the designs, we studied attacks that are common for this kind of device, as well as good practice recommendations that have arisen, both in the academic literature and in industry (e.g., The MITRE Corporation CWE). We detailed our design decisions, and specified the protocols both informally and in the formal language of ProVerif. We have used ProVerif to verify a number of relevant security properties about them.

References

1. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: International Cryptology Conference (CRYPTO). LNCS, vol. 773, pp. 232–249. Springer, Berlin, Heidelberg, Santa Barbara, CA (Aug 1993)
2. Blanchet, B., Smyth, B., Cheval, V., Sylvestre, M.: ProVerif 2.04: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial (Dec 2021)
3. Blanchet, B., Cheval, V., Sylvestre, M.: ProVerif (v. 2.04) (Sep 2021), http://prosecco.gforge.inria.fr/personal/bblanche/proverif/
4. Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.H.: SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution. In: IEEE European Symposium on Security and Privacy (EuroS&P). pp. 142–157. IEEE, Stockholm, Sweden (Jun 2019)
5. Costan, V., Devadas, S.: Intel SGX explained. Cryptology ePrint Archive **2016**(086), 1–118 (2016)
6. Lee, D., Kohlbrenner, D., Shinde, S., Song, D., Asanović, K.: Keystone: A framework for architecting TEEs. arXiv preprint arXiv:1907.10119 (2019)
7. Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., Song, D.: Keystone: An open framework for architecting trusted execution environments. In: Fifteenth European Conference on Computer Systems (EuroSys). pp. 1–16. ACM, Heraklion, Greece (Apr 2020)
8. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: Armageddon: Cache attacks on mobile devices. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 549–564 (2016)

9. Lowe, G.: A hierarchy of authentication specifications. In: IEEE Computer Security Foundations Workshop (CSFW). pp. 31–43. IEEE, Rockport, MA (Jun 1997)
10. Moghimi, A., Irazoqui, G., Eisenbarth, T.: Cachezoom: How sgx amplifies the power of cache attacks. In: International Conference on Cryptographic Hardware and Embedded Systems. pp. 69–90. Springer (2017)
11. Moreira, J., Ryan, M.D., Garcia, F.D.: Repository for ProVerif models (Aug 2022), available at: https://github.com/jmor7690/esorics2022-two-tiered-tcb
12. Noorman, J., Van Bulck, J., Mühlberg, J.T., Piessens, F., Maene, P., Preneel, B., Verbauwhede, I., Götzfried, J., Müller, T., Freiling, F.: Sancus 2.0: A low-cost security architecture for IoT devices. ACM Transactions on Privacy and Security (TOPS) **20**(3), 7:1–7:33 (2017)
13. Pinto, S., Santos, N.: Demystifying arm trustzone: A comprehensive survey. ACM Computing Surveys (CSUR) **51**(6), 130 (2019)
14. Ragab, H., Milburn, A., Razavi, K., Bos, H., Giuffrida, C.: CrossTalk: Speculative data leaks across cores are real. In: IEEE Symposium on Security and Privacy (S&P). pp. 1852–1867. IEEE, San Francisco, CA (May 2021)
15. Savagaonkar, U., Porter, N., Taha, N., Serebrin, B., Mueller, N.: Titan in depth: Security in plaintext (Aug 2017), https://cloud.google.com/blog/products/identity-security/titan-in-depth-security-in-plaintext
16. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-privilege-boundary data sampling. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 753–768. ACM, London, UK (Nov 2019)
17. Skorobogatov, S.P.: Semi-invasive attacks: a new approach to hardware security analysis. Tech. Rep. 630, University of Cambridge (2005)
18. Trusted Computing Group (TCG): Trusted Platform Module Library Specification, Part 1: Architecture (Family "2.0", Level 00, Revision 01.59) (Nov 2019)
19. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: USENIX Security Symposium (USENIX Security). pp. 991–1008. USENIX Association, Baltimore, MD (Aug 2018)
20. Van Bulck, J., Oswald, D., Marin, E., Aldoseri, A., Garcia, F.D., Piessens, F.: A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1741–1758 (2019)
21. Zhang, N., Sun, K., Shands, D., Lou, W., Hou, Y.T.: Truspy: Cache side-channel information leakage from the secure world on arm devices. IACR Cryptol. ePrint Arch. **2016**, 980 (2016)

A Security Properties

In this appendix, we present the formal description of the security properties for Protocols 1–3. Due to space constraints, we include the properties for Protocol 4 in the extended version of the paper. We have verified these properties using ProVerif [3, 2], which is a tool for automated analysis of security properties in cryptographic protocols.

Security properties are expressed through guarded first-order logic formulas. These properties can be classified as reachability properties (i.e., a certain event in the execution trace is reachable) or as correspondence assertions (i.e., a certain event always occurs prior to the execution of a later event). For correspondence assertions it is customary to check reachability of the event that occurs later, since if this event never occurs, then the assertion will be trivially verified. We omit those reachability sanity checks here. Events are used to define security properties, and they do not modify the semantics of the protocols. Events represent local computations or mark relevant points in the execution of the protocols. The notation "EventName$(x_1, \ldots, x_n)$@t" indicates that an event with name "EventName" and parameters $x_1, \ldots, x_n$ occurs at time t. The naming of the events has been chosen to be self-documenting.

Also, note that the security properties for Protocol 2 and 3 can only hold if no memory corruption occurs in a given platform boot. For this reason, correspondence properties in Protocols 2 and 3 are conditioned to the event "MemoryCorrupted( )" in the description of the relevant security properties below.

Security properties for Protocol 1: MTCB A/B Update.

[P1.1] Every MTCB only executes firmware installed by the Vendor (initial install) or a previous legitimate firmware (subsequent installs) that has been previously created and signed by the Vendor:

$$\begin{aligned}
&\forall id_M, ptr, ver_M, code_M, spk_V, t_3.\\
&\quad \text{MtcbStarts}(id_M, ptr, ver_M, code_M, spk_V)@t_3 \Rightarrow\\
&\qquad \big(\exists t_2.\ \text{MtcbInstalls}(id_M, ptr, ver_M, code_M, spk_V)@t_2 \wedge\\
&\qquad\ \ \exists t_1.\ \text{VendorCreates}(ver_M, code_M)@t_1 \wedge (t_1 < t_2 < t_3)\big).
\end{aligned}$$

[P1.2] Once a given MTCB executes firmware of a certain version number ver_M it will never execute firmware with version number $ver'_M < ver_M$

$$\begin{aligned}
&\forall id_M, ptr, ver_M, code_M, spk_V, t_1, \quad id'_M, ptr', ver'_M, code'_M, spk'_V, t_2.\\
&\quad \text{MtcbStarts}(id_M, ptr, ver_M, code_M, spk_V)@t_1 \wedge\\
&\quad \text{MtcbStarts}(id'_M, ptr', ver'_M, code'_M, spk'_V)@t_2 \Rightarrow\\
&\qquad \big((t_1 \le t_2) \wedge (ver_M \le ver'_M)\big) \vee \big((t_2 \le t_1) \wedge (ver'_M \le ver_M)\big).
\end{aligned}$$

Security properties for Protocol 2: Secure Boot.

[P2.1] Only a legitimate ETCB is allowed to start on the platform.

$$\begin{aligned}
&\forall inst, code_E, code'_O, code'_E, t_2. \quad \text{EtcbStarts}(inst, code_E)@t_2 \Rightarrow\\
&\quad \text{IsLegitimateEtcb}(code_E) \vee\\
&\quad \big(\exists t_1.\ \text{MemoryCorrupted}(inst, code'_O, code'_E)@t_1 \wedge (t_1 < t_2)\big)
\end{aligned}$$

[P2.2] Only a legitimate OS is allowed to start on the platform.

$$\begin{aligned}
&\forall inst, code_O, code'_O, code'_E, t_2. \quad \text{OsStarts}(inst, code_O)@t_2 \Rightarrow\\
&\quad \text{IsLegitimateOs}(code_O) \vee\\
&\quad \big(\exists t_1.\ \text{MemoryCorrupted}(inst, code'_O, code'_E)@t_2 \wedge (t_1 < t_2)\big)
\end{aligned}$$

[P2.3] The AKEP2 protocol between the MTCB and the ETCB guarantees mutual, injective agreement [9] for the MTCB nonce n_M (which is used later to obtain the boot secret bs_{ME}). For convenience, we assume that the array of agreed parameters $pars$ contains n_M:

$$\begin{aligned}&\forall id_M, id_E, pars, t_2.\ \text{MtcbAkep2Commit}(id_M, id_E, pars)@t_2 \Rightarrow\\ &\quad \big(\exists t_1.\ \text{EtcbAkep2Running}(id_E, id_M, pars)@t_1 \wedge (t_1 < t_2)\big)\wedge\\ &\quad \neg\big(\exists t_2'.\ \text{MtcbAkep2Commit}(id_M, id_E, pars)@t_2' \wedge \neg(t_2 = t_2')\big),\end{aligned}$$

$$\begin{aligned}&\forall id_M, id_E, pars, t_2.\ \text{EtcbAkep2Commit}(id_E, id_M, pars)@t_2 \Rightarrow\\ &\quad \big(\exists t_1.\ \text{MtcbAkep2Running}(id_M, id_E, pars)@t_1 \wedge (t_1 < t_2)\big)\wedge\\ &\quad \neg\big(\exists t_2'.\ \text{EtcbAkep2Commit}(id_E, id_M, pars)@t_2' \wedge \neg(t_2 = t_2')\big).\end{aligned}$$

These formulas represent injective correspondence assertions as predicate logic formulas. We remark that the last line of the two formulas above ensure the injectivity of the correspondence assertions, since no two events can occur at the same time point.

Security properties for Protocol 3: Remote Attestation.

[P3.1] For a given boot instance, It cannot happen that the MTCB generates an attestation signature, the Verifier validates the attestation, and there is an attack event.

$$\begin{aligned}\neg\big(\exists \mathit{inst}, \mathit{chal}, \sigma_M, t_1, t_2, t_3.\ &\text{MtcbGeneratesSignature}(\mathit{inst}, \mathit{chal}, \sigma_M)@t_1\wedge\\ &\text{VerifierValidatesAttestation}(\mathit{chal}, \sigma_M)@t_2\wedge\\ &\text{AttackEvent}(\mathit{inst})@t_3\big).\end{aligned}$$

[P3.2] For every boot instance, if the Verifier validates an attestation, then an attestation signature must have been generated by the MTCB before, and the following events must have occurred before that: 1. the Verifier generates the challenge, 2. the legitimate OS has been loaded, 3. the legitimate ETCB has been loaded, 4. the legitimate ETCB has been started.

$$\begin{aligned}&\forall \mathit{chal}, \sigma_M, t_7.\ \text{VerifierValidatesAttestation}(\mathit{chal}, \sigma_M)@t_7 \Rightarrow\\ &\quad\big(\exists \mathit{inst}, t_6.\ \text{MtcbGeneratesSignature}(\mathit{inst}, \mathit{chal}, \sigma_M)@t_6\wedge\\ &\quad\ \exists t_5.\ \text{VerifierGeneratesChallenge}(\mathit{chal})@t_5\wedge\\ &\quad\ \exists t_4.\ \text{EtcbStarts}(\mathit{inst}, \text{ETCB\_LEGITIMATE})@t_4\wedge\\ &\quad\ \exists t_3.\ \text{EtcbLoaded}(\mathit{inst}, \text{ETCB\_LEGITIMATE})@t_3\wedge\\ &\quad\ \exists t_2.\ \text{OsLoaded}(\mathit{inst}, \text{OS\_LEGITIMATE})@t_2\wedge\\ &\quad\ (t_2, t_3, t_4, t_5 < t_6 < t_7)\\ &\quad\big) \vee \exists \mathit{code}_O', \mathit{code}_E', t_1.\ \text{MemoryCorrupted}(\mathit{inst}, \mathit{code}_O', \mathit{code}_E'))@t_1 \wedge (t_1 < t_7).\end{aligned}$$

The source code with the formal models, the security properties and a collection of sanity check queries can be found in [11].