Remote attestation in an isolated server setting through the usage of a tamper-resistant hardware token

Master's Thesis of

André Pascal Sperrle

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr. Jörn Müller-Quade
Second examiner:	Prof. Dr. Thorsten Strufe
First advisor:	Dr. Jeremias Mechler

12. Mai 2025 – 12. November 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Remote attestation in an isolated server setting through the usage of a tamper-resistant hardware token (Master's Thesis)

Karlsruhe, 12. November 2025

. . . .

(André Pascal Sperrle)

Abstract

This work implements a TPM-based remote attestation protocol in the context of a data diode. The remote attestation protocol allows a Verifier to gain trust in the software state of a server, while the data diode prevents the leakage of data even after the server is compromised. Seemingly the aforementioned primitives cannot be combined, since remote attestation inherently requires interaction. This work leverages a tamper resistant hardware to overcome the challenge. The Verifier delegates trust into the token, which audits the Prover on behalf of the verifier. After the Verifier transported the token to the enclave, the token verifies the state behind the data diode, before it reports back to the Verifier over a visual channel like the server rooms camera feedback.

1. The implementation of an Auditor application in the memory safe programming language Go. The work leverages the TamaGo framework to run the Auditor application as a bare metal program on the hardware token 'USB Armory Mk II'. The focus lies on a minimalistic and secure implementation.
2. A literature research which covers the TPM based remote attestation, as well as the security features and limitations of the used hardware token.
3. An architectural overview and a security analysis which covers the different stakeholders, the security guarantees this work provides and the limitations of this work.

Table of Contents

Chapter 1. Introduction

In the era of cloud computing most services are delivered on rented computing power. With the rise of large language models there is an even greater demand for rentable computing power. For an institution or a business which respects the privacy of their customers this creates a challenge, since it should be reluctant to entrust its confidential data to a cloud provider. This is for example the case for a medical institution, since a single patient record can be worth hundreds of dollars in the hands of a skilled black mailer [44]. Therefore, it is of utmost importance for a medical institution to protect the confidentiality of their patient records with all possible means.

To protect the confidentiality of data which resides on a remote-server, a so-called enclave can be used. Intel SGX for example is a CPU provided enclave. The server's CPU creates an isolated execution environment inside the untrusted server, which enforces that no data leaves the CPU unencrypted. A remote attestation protocol in place provides the institution with evidence that only institution provided data entered the enclave. A secure implementation of the enclave is crucial, since the confidentiality and integrity of the institution is at stake. The implementation of such an enclave is difficult to achieve in practice, since the underlying attack model assumes that an adversary has full control over the system. This includes the OS, which has fine-grained control over the system's resource allocation or in other words access to a large side-channels. A malicious OS can for example monitor the enclaves access patterns to shared caches, monitor and alter the behavior of the CPU's branch-prediction, and can even use a software induced fault-injection. As a result, CPU-based enclaves are vulnerable to strong, and novel side-channel attacks which compromise the confidentiality and integrity guarantees. As demonstrated in a survey about attacks against Intel SGX [85].

In this work the server is assumed to reside in a so-called macro-enclave which is an isolated execution environment in the physical world. The macro-enclave can be seen as an enclosed room which cannot be entered without triggering some tamper-intervention mechanism. Once the enclave detects an intrusion, it destroys all data residing on the server by for example erasing memory-encryption keys. If the enclave launched trustworthy code, the macro enclave offers significantly stronger privacy guarantees compared to a CPU provided enclave. Since the program which has been launched on the enclave does not have to share its resources with a potentially malicious OS, most of the attack scenarios against Intel-SGX do not exist in the macro-enclave model. Even better, the macro-enclave also discloses most side-channels and can protect the confidentiality of data residing on the enclave in case physical intrusion occurs. The question remains, how a remote party can ensure that the enclave launched trusted from a set of trusted configurations. This is where a TPM comes into play. A TPM or Trusted Platform Module is a cryptographic processor which offers platform independent remote-attestation capabilities. The TPM has access to a set of memory banks, so-called Platform Configuration Registers (PCR), which store a set of digests. The TPM based remote-attestation creates essentially a signature over the TPM's PCR values, which provide evidence to the Verifier which software has been launched on the enclave.

In the best case, the remote-attestation protocol provides evidence that only code trusted by the remote party has been launched. The evidence created by the TPM can not proof however, that the software executed on the enclave is trustworthy, since an adversary who exploits the enclave's software mid-execution, will be able to compromise the enclave without being detected. So even in the context of a perfectly secure macro-enclave which offers TPM-based remote attestation, one cannot guarantee that an adversary is unable to steal confidential data from the enclave. In a scenario where interaction is not required, the enclave can be connected via a so-called data-diode - a unidirectional network connection. A set of health care institutions for example, can a train a neuronal network by sending their joint data-set to the server. Due to the delicate nature of the data, an adversary must not be able to exfiltrate data from the enclave. After the server trained the model, the remote party requests access to the trained model from the provider. In practice a physical switch could signal the enclave's program to encrypt all data based on an established secret, before the enclave reverts the data-diode's orientation. To ensure that the server's software does not behave dishonest by for example leaking all data to the outside once the data-diodes orientation is fliped, the remote parties demand proof that only trusted software has been launched on the machine. In the context of a data-diode however, a remote-attestation protocol is impossible to realize, since it requires interaction.

This work addresses the question of how a remote attestation protocol can be realized, if a data diode is in place. The remote party - the Verifier - configures a secure hardware token acting as a hardware firewall between the attesting server - the Prover - and the remote Verifier. The token performs the attestation on behalf of the Verifier, before providing the Verifier with a transcript of the entire interaction, before establishing a shared secret between both the Prover and the remote Verifier. This work furthermore provides an implementation which minimizes the trust assumptions necessary into the token. In fact the result of this work suggest that the Verifier does have to make any trust assumptions about the token, without sacrificing security with respect to the remote-attestation and Key-Exchange protocol.

Previous work focused their attention on local attestation to fend off evil-maid attacks [52, 53]. The eponymous example describes an evil cleaning lady who finds an unattended laptop. Later, the owner - finding the unattended device - must verify that the device is authentic and trustworthy. However, in practice pure software solutions fall short. Hard disk encryption for example protect the data against the preying eyes of the evil maid, but the maid can trick the user to reveal the password by imitating an inconspicuously looking log-in screen. Stateful (H)OTP-based protocols established themselves as a common countermeasure. During system startup, the device authenticates itself to the user through a one-time-password (OTP). For this to work, the owner stores a symmetric key within the TPM, which binds the secret to a policy. The policy dictates that the device obtains access to the key only if the system is in a known, and trusted state. The owner trusts the machine if and only if the device produces an OTP-code equivalent to the created code by a second-factor. In a remote scenario, a verifier with access to the camera feedback of the server room can capture the code created and verify that the device in question is trustworthy. The problem with a stateful (H)OTP solution is, that it implicitly assumes that the user, who owns the device, can trust the device initially. In a remote setting however, a Verifier cannot trust a foreign and remote device to begin with. In fact the (H)OTP approach does not verify the device's trustworthiness at all. Instead the TPM uses a so-called policy which compares the system's measured state with the measurement created when the key was sealed away. In other words, the solution compares the current state of the system with the previous one. Therefore, such a solution protects insufficiently in a remote scenario and a proper remote-attestation protocol should be used. The idea is, that the Verifier sends in a secure hardware module which performs the actual attestation in the name of the verifier. Once the token verified the server's state, it informs the client over a visual channel that the server can be trusted.

In a setting where a data diode is in place the Verifier must also have means to establish a shared secret with the server. Without an additional assumption this however is impossible, and the remote verifier cannot send confidential data to the server. Therefore the hardware token may provide the server with a secret key, once it verified the server's state.

1.1. Challenges and Goals

The work operates under the assumption that a remote server resides in an isolated environment, a so-called macro-enclave. To uphold strong confidentiality guarantees, the remote enclave cannot leak data to the outside since it is connected via a data diode to a network. The direction of the data diode is reversed only, when the enclave is ready to report the result of the enclave's computation to a remote party. To protect the data's confidentiality, the enclave must encrypt the data with a secret shared only with the remote client. Furthermore, the client must make sure that the software running inside the enclave does not leak any data to the outside. A remote attestation protocol provides the client, the so-called Verifier, with assurance over the enclave's software state.

An architecture which provides an attestation service must address the following challenges:

1. How can the Prover assure that the enclave is physically uncompromised?
2. How can the Verifier verify the entire software and firmware state of the enclave?
3. How can the Verifier perform remote attestation with the Prover who resides behind a data diode?

It is out-of-scope for this work to fully address the first question. This work assumes, that the macro-enclave is tamper-evident and that it destroys an asymmetric key once an intruder attempts to forcefully enter the macro-enclave where the server resides. If a Reviewer provides the Verifier with the macro-enclave's public key, the Verifier can request fresh evidence from the enclave that it is in a trustworthy state. In addition, the enclave must have access to a tamper-intervention mechanism, which protects data residing on the macro-enclave. For example the deletion of RAM encryption keys may be enough to implement the said intervention mechanism. Therefore, the first question boils down to the question on how can the Provider prove that it behaved honestly during the enclave's setup phase? Not only is willful tampering with computer hardware illegal in most countries, but an external Reviewer can, to some extent, assure itself that the enclave is trustworthy at the end of the enclave's setup phase. The rest of the work focuses on addressing the second and third question, whereas answering the first question remains for future work.

This work makes the following contributions.

1. A novel approach for verifying the software state of a remote server - which is connected to a network via a data diode. The work leverages a secure hardware token who plays the role of the Auditor who resides physically close to the remote server. Upon a successful remote attestation, the token reports back to the Verifier over a unidirectional communication channel like the server room's camera-feedback. The implemen-

tation [89] of the Auditor application is kept in the memory safe programming language Go on the hardware token 'USB Armory Mk II'. To minimize code dependencies - especially to memory-unsafe ones - the TamaGo framework is used to provide bare-metal execution. Which means that the token executed the application without an underlying OS.

2. A literature research which covers TPM based remote attestation, as well as the security features and limitations of the hardware token in use. The research uncovers two vulnerabilities which allow an adversary - with unmonitored access to the token - to take control over the token's execution flow.
3. The work provides an architectural overview and a subsequent security analysis. The architecture explains the role of the Auditor, which in essence plays the role of a hardware firewall. The Auditor provides the Verifier with a verifiable transcript of the remote attestation protocol.
4. The work discusses how a secure key-exchange can be realized and discusses potential weaknesses of different approaches discussed.
5. The work discusses different variations in the architecture and motivates how a scenario with multiple Verifiers and a single token can be realized.

1.2. Outline

The following outline's the rest of this work's structure.

The remaining work starts with the “Preliminaries” in Chapter 2 ny introducing “Common Terms and Definitions” in Section 2.2, “Cryptographic Terms” in Section 2.3 and TPM related terms in Section 2.5 “TPM”. The Section 2.4 introduces the token used throughout this work - the “USB Armory Mk II” - as well as the framework used to program the token. To maintain the reading flow, the Glossary references to most of thr terms and definitions used throughout this work. The Related Work ends the chapter with the “Related Work”.

The following Chapter 3 defines the “Architecture” in which the remote-attestation takes place. A good understanding of the architecture is necessary to understand the following Chapter 4 which cover's a “Security Analysis”. The Analysis covers the strength and weaknesses of the components used throughout this work and explains against which types of attacks the overall architecture protects against and which not. One of the findings is, that the MK II is vulnerable against an adversary which is physical present. The Chapter 6 marks the end of this work and discusses “Summary and Future Work”.

Chapter 2. Preliminaries

The work first explains in Section 2.1 the Syntax used throughout this work, before introducing "Common Terms and Definitions" in Section 2.2 and "Cryptographic Terms" in Section 2.3. The rest of the preliminaries provides background information on the token which interacts with the enclave in the role of an Auditor in Section 2.4. A TPM device inside the enclave implements the attestation on the Prover's behalf, therefore the Section 2.4 cover's TPM based attestation and all datastructures related to TPM based remote-attestation. For quick reference the reader is referred to the Glossary at the end of this work. Some readers may find that the digital version [89] of this paper is more accessible.

2.1. Syntax

The chapter explains the syntax used throughout this work.

Pseudo-Code

The work uses pseudo code leans on the syntax of the Go programming language, but deviates in favor of readability. The data structures should provide an understanding how the data is represented in memory. If not specified otherwise primitive datatypes are represented in the big-endian format and datastructures are stored in compact form.

```
var (
    byteType    byte    ❶
    uintType    uint8   ❷
    intTypt     int8    ❸
    array1Type  []byte  ❹
    array2Type  [length]byte  ❺
    constant    [2]byte{1, 2} ❻
)
type structure struct{ ❼
    entry1 byte
    entry2 []byte
}
```

Example syntax

❶ A variable of type `byte`.

❷ An unsigned integer with 8 bits in size. Valid integers of larger size are `uint16`, `uint32`, `uint64`.

❸ A signed integer with 8 bits in size. Valid integers of larger size are `int16`, `int32`, `int64`.

❹ A (pointer) to an array which does not have a predefined size. The binary representation of such an array must start with the length of the array followed by the arrays content.

❺ An array of predefined length, which according to the Go syntax must be constant. This work however uses variables as well, to indicate how marshalled data is represented in memory.

❻ A constant variable of predefined length which is not supported by the Go syntax

❼ A structure which is composed of other data types.

For the representation of mappings or functions, the following syntax is used:

Function

$$[2]\text{byte}\{\text{data.length}\} \tag{1}$$

$$\mathbf{H}_{\text{hashAlg}}(\,\cdot\,) \tag{2}$$

$$f\left(x\left\{\text{middle} \parallel y_{\text{optional}}\right\}\right) \tag{3}$$

Example Functions

1. The length of the data encoded as binary. The memory layout is typically in big-endian.
2. A hash function of type hashAlg. By default it is of type SHA-256.
3. A function called with the values x and an optional value y which can be ommited.

2.2. Common Terms and Definitions

The chapter introduces general terms and definitions used throughout this work.

ACPI (Advanced Configuration and Power Interface)

“*An open industry specification that describes OS-directed configuration, thermal and power management. ACPI Machine Language code is usually provided as part of the BIOS and interpreted by an operating system-provided interpreter*” [40 p. 11].

APDU (Application Protocol Data Unit)

“*In the context of smart cards, an application protocol data unit (APDU) is the communication unit between a smart card reader and a smart card. The structure of the APDU is defined by ISO/IEC 7816-4 Organization, security and commands for interchange*” [1].

CBOR-Format (Concise Binary Object Representation) [75]

“*The Concise Binary Object Representation (CBOR) is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation*” [75 pp. 3.10]. A smaller code base is easier to maintain, and harder to exploit. In addition the format specifies a variety of security options. A pure, and apparently well-supported, Go implementation exists [54], which does not rely on Go’s 'unsafe' primitive.

BIOS (Basic Input Output System)

“*Performs basic system configuration and setup and starts the operating software*” [40 p. 11].

Bus

"*In computer architecture, a bus (historically also-called a data highway or databus) is a communication system that transfers data between components inside a computer or between computers*" [2].

Challenge-Response Protocol

In computer security, a challenge-response protocol is a protocol in which one party presents a question - the “challenge” - and another party must provide a valid answer - the “response” [3]. A challenge-response authentication protocol can for example allow the challenger to authenticate itself to another party without revealing the shared secret.

Data-diode

A data-diode is “*a network appliance or device that allows data to travel in only one direction*” [4].

DRT (D-RTM Resources Table)

An [ACPI] table. It provides, among others, the location of the Platform Manufacturer code which starts the D-RTM process [40 p. 12].

Endianess

"*In computing, endianness is the order in which bytes within a word data type are transmitted over a data communication medium or addressed in computer memory, counting only byte significance compared to earliness*" [5]. A **Big-Endian** system stores the most significant byte of an integer in the smallest address, whereas a **Little-Endian** system stores the least-significant byte in the smallest address.

Evidence

"*is a set of Claims about the Target Environment that reveal operational status, health, configuration, or construction that have security relevance. Evidence is appraised by a Verifier to establish its relevance, compliance, and timeliness*", as defined in [74 p. 24].

Ephemeral key-pair

is an asymmetric key-pair, that is intended for a very short period of use. Ordinarily for a single transaction.

I2C

"*I2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices*" [43 p. 1229]. An I2C-bus specification for NXP processors are provided at [48].

Remote Attestation

is according to [88] "*a process allowing to challenge, using time and behavior measurements, the assumption that a black box runs a given software. The concept is based on coding programs in a way that modifications in them will inevitably cause observable behavioral changes, variations in resource consumption, or response time differences*".

Nonce

or **N**umber used **once** that is a value which has at most an acceptably small chance of repeating itself, like a random number with high entropy.

Protocol

"*In the context of cryptography, 'protocol' is a shorthand for 'cryptographic protocol.' A cryptographic protocol is a distributed algorithm describing precisely the interactions between two or more entities to achieve certain security objectives (Menezes et al., 1997). The entities interact with each other by exchanging messages over private and/or public communication channels*" [55 p. 1987].

PUF (Physical Unclonable Function)

"*A physical unclonable function, or PUF, is a physical object whose operation cannot be reproduced ('cloned') in physical way (by making another system using the same technology)*" [6].

SMM (System Management Mode)

The SMM "*is an operating mode of x86 central processor units (CPUs) in which all normal execution, including the operating system, is suspended. An alternate software system which usually resides in the computer's firmware, or a hardware-assisted debugger, is then executed with high privileges*" [7].

SoC (System on a chip)

"*A system on a chip (SoC) is an integrated circuit that combines most or all key components of a computer or electronic system onto a single microchip*" [8].

TLV-Format (Type-Length-Value Format)

is a simplistic encoding used to encode a stream of variable-length data of varying types. The encoding maps data into it's corresponding Type-Length-Value representation. The type must be of fixed length, and the length's encoding must be fixed for a specific type. Entries which are implied by the context are can be omitted. In the TPM specification the encoding is usually as follows:

$\text{data} \rightarrow (\text{byte}[2]\{\text{data.type}\} \parallel \text{byte}[2]\{\text{data.len}\} \parallel \text{byte}[\text{data.len}]\{\text{data.value}\})$

TPM (Trusted Platform Module)

"*A Trusted Platform Module (TPM) is a secure cryptoprocessor that implements the ISO/IEC 11889 standard. Common uses are verifying that the boot process starts from a trusted combination of hardware and software and storing disk encryption keys*" [9].

TEE (Trusted execution environment)

is according to [10] a "*secure area of a main processor. It helps the code and data loaded inside it be protected with respect to confidentiality and integrity. Data confidentiality prevents unauthorized entities from outside the TEE from reading data, while code integrity prevents code in the TEE from being replaced or modified by unauthorized entities, which may also be the computer owner itself as in certain DRM schemes described in Intel SGX. [...]. This allows user-level code to allocate private regions of memory, called* ***enclaves****, which are designed to be protected from processes running at higher privilege levels*". For example, CPU's with Intel SGX provide hardware support for an allegedly secure enclave. The enclave can attest for its state to an external party, and allegedly protects the integrity, and confidentiality of its content from a compromised OS.

Macro-Enclave

The Macro-Enclave extends the idea of an enclave to the real-world, but is not a term found in the literature. Instead of the CPU providing shielded execution to a single application against another malicious application, the enclave shields the computer system as a whole against physical tampering from an outsider. To uphold the enclave's confidentiality guarantees, it erases its memory in response to physical tampering. In addition, the enclave has access to a TPM, which allows an external party to establish trust into the enclave's software state. A honest setup phase is essential to the enclave's security. A certifying Reviewer provides the Verifier with the assurance that the provider behaved honestly during the setup phase.

The advantage of a physical enclave, over a CPU provided enclave is, that the strong physical isolation greatly reduces the risk of information leakage in the context of a strong adversarial model. In addition, a physical enclave does not impose a computational burden, as a Verifier can run a native application on the platform's hardware. The literature provides numerous examples for - purely software driven - classes of side-channel attacks against Intel-SGX [85].

2.3. Cryptographic Terms

The chapter introduces common cryptographic primitives and definitions. It furthermore defines a variety of security notions used throughout this work.

Asymmetric Cryptography

“*Cryptography that uses two separate keys to exchange data, one to encrypt or digitally sign the data and one for decrypting the data or verifying the digital signature. Also known as public key cryptography*” [11 p. 160].

Block Cipher

“*An invertible symmetric-key cryptographic algorithm that operates on fixed-length blocks of input using a secret key and an unvarying transformation algorithm. The resulting output block is the same length as the input block*” [12 p. 6].

Block cipher mode of operation

“*In cryptography, a block cipher mode of operation is an algorithm that uses a block cipher to provide information security such as confidentiality or authenticity. A block cipher by itself is only suitable for the secure cryptographic transformation (encryption or decryption) of one fixed-length group of bits called a block. A mode of operation describes how to repeatedly apply a cipher’s single-block operation to securely transform amounts of data larger than a block*” [13]. The following provides an overview of common block-cipher modes.

Security	Mode	Literature
[IND-CPA]	CBC (Cipher Block Chaining) mode	[78 p. 17]
	CFB (Cipher Feedback) mode	[78 p. 18]
	CTR (Counter) mode	[78 p. 22]
[IND-CCA]	GCM (Galois/Counter Mode)	[79]

Hash-Function

is in this work an abbreviation for the term “cryptographic hash function”. A hash function is according to [14 p. 12] “*A function that maps a bit string of arbitrary length to a fixed-length bit string*”. A cryptographic hash function must satisfy the following properties [14 pp. 12-13]:

1. “***One-way** – It is computationally infeasible to find any input that maps to any prespecified output*”. Formaly defined at [82 pp. 242-245].
2. “***Collision resistant** – It is computationally infeasible to find any two distinct inputs that map to the same output*”. Formaly defined at [82 pp. 154-155].

MAC-Scheme (Message Authentication Code)

“*In cryptography, a message authentication code (*MAC), sometimes known as an authentication tag, is a short piece of information used for authenticating and integrity-checking a message*” [15]. A MAC-Scheme creates a MAC with the message and a secret key as an input. The verification of a MAC requires knowledge of the secret key.

RO (Random Oracle)

“*In cryptography, a random oracle is an oracle (a theoretical black box) that responds to every unique query with a (truly) random response chosen uniformly from its output domain. If a query is repeated, it responds the same way every time that query is submitted*” [16].

Security Bits

The security bits of a cryptographic algorithm is a numerical estimate of the algorithm’s strength in security. For an algorithm with 120 bits of security - which in this work is the accepted minimum - an adversary must perform at least 2^{128} computational steps to break the algorithm’s security. The estimated strength of an algorithm is an upper bound, since it does not account for potentially undiscovered attacks.

Signature-Scheme

A signature scheme can be viewed as the asymmetric equivalent to a MAC scheme. “*Signature schemes allow a signer S who has established a public key pk to ‘sign’ a message using the associated private key sk in such a way that anyone who knows pk (and knows that this public key was established by S) can verify that the message originated from S and was not modified in transit*” [82 p. 439]. The scheme is formaly defined at [82 p. 442].

Symmetric Cryptography

“*A cryptographic algorithm that uses the same secret key for its operation and, if applicable, for reversing the effects of the operation (e.g., an AES key for encryption and decryption)*” [11 p. 161].

For encryption schemes, which protect the confidentiality of a message, the following terms apply.

IND (Indistinguishability)

The term captures an encryption scheme’s goal to provide confidentiality for messages encrypted via the encryption scheme. “*The basic idea is to consider an adversary (not in possession of the secret key) who chooses two messages of the same length. Then one of the two messages is encrypted, and the ciphertext is given to the adversary. The scheme is considered secure if the adversary has a hard time telling which of the two messages was the one encrypted*” [73 p. 102]. The chosen message is also called the "challenge", and the adversary wins if it can answer the challenge significantly better than guessing.

IND-CPA (Indistinguishability under Chosen-Plaintext Attack)

The Chosen-Plaintext Attack (CPA) “*capture the ability of an adversary to exercise (partial) control over what the honest parties encrypt [...] by giving the adversary A access to an encryption oracle*” [82 pp. 73-74]. Informaly, an encryption scheme secure in the **IND-CPA** model protects the confidentiality of ciphertexts. The reader must consult the literature for a formal definition.

IND-CCA (Indistinguishability under Chosen-Ciphertext Attacks)

An encryption scheme which is secure in the IND-CCA model protects the confidentiality of encrypted messages also against active tampering, since the adversary gains “*access to a decryption oracle in addition to an encryption oracle*” [82 p. 96]. The adversary cannot use the decryption-oracle to decrypt the challenge itself, however. An adversary in the **IND-CCA2** model can access the decryption oracle at arbitrary times, whereas the **IND-**

CCA1 model restricts access to the decryption oracle before the adversary requests the challenge. Compared to **IND-CCA1**, the adversary in the IND-CCA2 model is strictly stronger, as the encryption scheme must also protect against so-called "Padding-Oracle Attacks" as described in the literature [82 p. 98].

RSAES-OAEP [17]

"*RSAES-OAEP is a public-key encryption scheme combining the RSA algorithm with the Optimal Asymmetric Encryption Padding (OAEP) method*" [17 p. 3], which is IND-CCA1 secure [17 p. 21] in the Random Oracle Model.

AES (Advanced Encryption Standard)

AES is a symmetric block-cipher with 128-bit block length and supported key-lengths 128, 192, and 256, which replaced DES by winning in the year 2000 the NIST competition [82 p. 223]. Despite extensive public scrutiny, at the time of this writing no significant security vulnerability is known. The best known key-recovery attack [91] reduces AES security strength to K-2, where K is the length of the key. A quantum computer can however leverage so-called Grover's algorithm to reduce the security strength to K/2.

Grover's algorithm

Grover's algorithm, also known as the quantum search algorithm, is a quantum algorithm for unstructured search that finds with high probability the unique input to a black box function that produces a particular output value, using just $O(\sqrt{N})$ evaluations of the function, where N is the size of the function's domain.

For MAC and Signature schemes, which protect the integrity of a message, the following terms apply.

EUF-CMA (Existentially Unforgeable Under an adaptive Chosen-Message Attack)

"*The Security of a signature scheme [or MAC scheme] means that an adversary should be unable to output a forgery even if it obtains signatures on many other messages of its choice*" [82 p. 443]. The literature formalizes the term [82 p. 443].

Hash-and-MAC

The Hash-and-MAC Paradigm constructs a MAC algorithm for messages of arbitrary size from a collision-resistant hash function and a MAC algorithm for messages of fixed size as followed: "*First, an arbitrarily long message m is hashed down to a fixed-length string H s (m) using a collision-resistant hash function. Then, a (fixed-length) MAC is applied to the result*" [82 p. 159].

Hash-and-Sign

The Hash-and-Sign Paradigm constructs a Signature algorithm for messages of arbitrary size. The Paradigm is in principle the same as the Hash-and-MAC Paradigm.

Encrypt-then-Mac

The Encrypt-then-Mac Paradigm first encrypts and then macs a message. This provides authenticated encryption.

Encrypt-then-Sign

The Encrypt-then-Sign Paradigm first encrypts and then signs a message. This provides authenticated encryption.

HMAC (keyed-Hash Message Authentication Code)

The HMAC algorithm is a MAC algorithm constructed from a secure hash function [38]. The resulting MAC algorithm accepts messages of arbitrary length, if the underlying hash function allows it.

CMAC (keyed-Hash Message Authentication Code)

The CMAC algorithm is a MAC algorithm constructed from a secure block-cipher [80], which accepts messages of arbitrary length.

Key-Exchange Protocol

A key-exchange protocol established a shared secret between two parties so-called Alice and Bob. It is secure, if "*an adversary who has eavesdropped on an execution of the protocol should be unable to distinguish the key k generated by that execution (and now shared by Alice and Bob) from a uniform key*" [82 p. 365].

Key-Derivation Function

"*A key-derivation function provides a way to obtain a uniformly distributed string from any distribution with high (computational) min-entropy*" [82 p. 187]. A high computational min-entropy means that the function draws any specific key with no more than a very small probability [82 p. 187].

SHA (Secure Hash Functions)

"*The Secure Hash Algorithm (SHA) refers to a series of cryptographic hash functions standardized by NIST*" [82 p. 234]. See also [SHA2] and [SHA3].

SHA2

SHA2 is a family of hash functions, which has no known collisions and no known attack better than brute-force. The family is suspectible to a theoretical attack, a so-called length-extension attack, which requires a known collision. SHA-256, SHA-384, and SHA-512 are, among others, standartized [86], where the respective number represents the length of the hash-functions computed digest.

SHA3

A standartized alternative to the SHA-2 hash family, which has no known weakness. SHA3-256, SHA3-384, and SHA3-512 are standartized among others [87].

2.4. USB Armory Mk II

The chapter is dedicated to the 'USB Armory Mk II' - a secure hardware token - in the shape and size of a USB-Stick. Since the token comes in different flavors, this work uses the term 'USB Armory Mk II' interchangeably with the token's part number 'UA-MKII-LAN-UL-512M' revision β [56]. The token's SoC is the i.MX 6UltraLite Processor processor, which contains an integrated single-core Cortex™-A7 528 MHz processor, 512Mb of DDR3 RAM, and a secure-boot feature provided by the so-called HAB (High Assurance Boot). In addition, the SoC implements a variety of security features, like a cryptographic processor called the CAAM, tamper-intervention mechanisms, and memory encryption for the RAM. An application cannot protect itself against physical tampering by software means alone. For the Auditor(-application) - which is introduced in Chapter 3 - the token's security features are vital, since the Auditor performs cryptographic operations in a hostile environment.

The Section 2.4.1 introduces the TamaGo framework, which provides driver support for the token's features. Next, the work provides an overview of the SoC's features in Section 2.4.2, followed by an introduction to the token's cryptographic processors in Section 2.4.3.

2.4.1. Programming the Token

The USB Armory can be programmed via the TamaGo framework. The framework provides native driver support for the token, and allows for the compilation, and bare-metal execution of Go applications. Its target audience are developers which aim to minimize code dependencies to memory unsafe languages like C by running the application directly on the hardware, without an underlying OS. Hence the term 'bare-metal' execution. The framework provides full compatibility to the Go's standard library, by implementing four system calls in assembly - namely 'sys_write', 'sys_exit', 'sys_clock_gettime', and 'sys_getrandom'. In addition, the framework implements a CPU-architecture specific driver for the USB Armory Mk II to provide, among others, an interface to the CPU's security features. Assuming the USB Armory Mk II as the target, TamaGo uses a total of 695 lines of assembly code in the Go-version 1.25.1 to allow bare-metal execution on the token. This count excludes external drivers like `armory-boot` [18] which may add another few lines of code.

2.4.2. SoC provided features

The chapter defines common terms related to the token, describes which features the token has to offer, and provides links to the related TamaGo drivers.

Black-Key

A black-key is an encrypted blob which contains a key. On the imx6ul only the CAAM has access to the decrypted key. A black-key protects the confidentiality and can, if desired, protect the integrity of the key in question, even if an adversary executes code on the processor.

CAAM

The **Crypto accelerator and assurance module** (CAAM) is the imx6ul's cryptographic processor [43 p. 366]. The CAAM supports for example AES encryption, ECDSA signing and hashing via the SHA-256 function [42]. An application can interact with the CAAM via the Job Descriptor language, as described in detail in the imx6ul's security reference manual. The CAAM has access to an **OTPMK** (One Time Password Master Key) which is stored in a PUF.

imx6ul

The i.MX 6UltraLite Processor (imx6ul) is a [SoC] of type Arm Cortex® - A7 Core and described in great detail in the SoC's manual [43]. The SoC' security reference manual describes security related features like tamper-monitors, tamper-intervention mechanisms, or the CAAM, but it cannot be accessed publicly. Therefore this work uses the security reference manual of other processors. More specifically the manual of the imx6ul's predecessor [39] and a processor from the same generation [41].

Job Descriptor

"*A job descriptor is a small program of CAAM commands. The commands may include conditional branches, loops, subroutine calls, or jumps to other job descriptors, as well as mathematical, cryptographic and data movement operations*" [39 p. 157].

RTIC (RunTime Integrity Checker)

The RTIC is a CAAM provided feature which performs periodic integrity checks on up to 4 selected memory areas in the RAM. Once the memory contents change, the SNVS enforces a reset of the platform. The RTIC protects the boot-process by monitoring the image which the HAB loads into the RAM during the early boot stages. After the boot finished, an application can put the RTIC in the so-called run-time mode to reactivate the RTIC. In run-time mode, the application can configure the RTIC at most once until the platform resets. From then on the RTIC periodically computes hardware-backed SHA-256 hashes of the selected memory areas and compares thee resulting digests with expected ones. Since the expected digest cannot be changed, the RTIC can only verify memory areas which do not change. The RTIC cannot protect the Stack or Heap of an application. In addition, the RTIC does not continuously monitor the memory areas contents, instead the „*RTIC periodically reads a small section of memory, waits for a specified period of time,and then reads another small section of memory*" [41 p. 652]. After the RTIC read out the entire memory section it performs a single check. The delay between periodic reads is defined by the 'RTICThrottle's throttle parameter [41 p. 651] and implemented in the TamaGo driver [57]. Interestingly, the application must trigger the verification of the memorys integrity. A coarse-grained timer enforces that the application verifies the integrity at a regular basis.

SE050 (Secure Element)

The SE050(-C1HQ1) [49] is a cryptographic processor which supports a variety of cryptographic operations [51 p. 5]. The SE050 is an external security chip on the token. An application can communicate with the token by exchanging APDU encoded data [50]. The application must first establish a connection to the SE050, where the imx6ul's I2C controller [43 p. 1229] acts as a proxy between itself and the SE050. The application must uses "*a peer to peer, half duplex transmission protocol*" [45 p. 3] to communicate with the SE050. To protect the communication against attacks on the Bus, the application must establish an authenticated channel with the SE050 [50 p. 19]. This however requires exchanged keys between the application and the SE050 [46].

SNVS (Secure Non-Volatile Storage)

The SNVS stores secrets in a supposedly secure fashion as long as the SNVS has access to a power-source. Based on a predefined security policy, the token can destroy managed secrets upon a detected security violation. In TamaGo [58], a privileged application which runs on the token can configure the token's security policy.

1. Access to the SoC's clock-, temperature-, voltage-, and power-tamper monitors, as well as the state of the RTIC. Upon detected tampering, the SNVS can enforce a reset of the platform.
2. A High Assurance Counter which is monotonic over the devices lifetime, as described in [43 p. 2911]. The counter is a non-rollover counter retained partially in volatile memory and partially on physical fuses. A security violation or a power outage puts the counter into an invalid state. During the next boot the device resets the volatile memory and blows a physical fuse to increment the counter after a reset occurred.
3. A Nonvolatile General-Purpose 32-bit register which clears itself upon detected tampering [43 p. 2943]. In addition, the SNVS can destroy the encryption keys which the CAAM uses to decrypt the black-keys.
4. The manual [43] recommends that the SNVS has access to a reliable power source to retain the counter's state and the secrets which manage the black-keys A reliable source also powers the tamper detection monitor. The SoC has a dedicated power supply pin called 'SNVS_LP' which is reserved for a low-power, always-ON power supply - like a coin-cell battery. The token however, does not support such a power-supply without modification of the token's circuit board layout.

HAB (High Assurance Boot)

The HAB is a component of the processors on-chip boot ROM code, which loads and boots from signed images. The HAB provides secure-boot features which *„protects against the potential threat of attackers modifying the areas of code or data in the programmable memory to make it behave in an incorrect manner“* [43 p. 231].

BEE (Bus Encryption Engine)

The imx6ul's BEE provides an on-the-fly AES-128 encryption and decryption engine with support for the CTR mode [43 p. 369]. The BEE implements the token's RAM encryption and provides confidentiality guarantees to the memory's content. It does not however protect the integrity of the RAMs content.

2.4.3. Cryptographic-Processors

The following chapter will discuss the specific security features available to the token.

The imx6ul has access to two cryptographic processors, the internal CAAM module and the external SE050. The TamaGo framework provides a driver for the CAAM which implements a few cryptographic algorithms, but lacks proper support for the CAAM's 'Job Descriptor language'. Without access to the Job-Descriptor language, most features like the usage of black-keys are not available to an application. In addition, an application cannot utilize the SE050 without writing ones own driver, since TamaGo lacks the corresponding driver.

If a driver for the SE050 would exist, two questions remain. First, how can the token detect and respond to physical tampering on the external chip, since an adversary could for example solder off the chip and launch an offline attack. The imx6ul has access to additional tamper-detection pins which may be able to monitor the secure element, but an adversary may be able to tap into the communication between both chips to provide the SoC with false assurance. Second, an application and the SE050 must communicate over authenticated channels. The provisioner must create and protect the shared communication secret in the long-term, without being hindered at provisioning the token multiple times. The CPU vendor discusses how the key can potentially be protected in [46], and the usage of black-keys may solve the question satisfactorily.

The driver for the CAAM [59] provides limited hardware support for the following cryptographic functions, among others:

- SHA-256 based hashing
- ECDSA-P256 signing
- AES based MAC scheme
- AES-CBC-256 based encryption
- KDF for AES-256 keys based on the internal OTPMK

According to [42], the job descriptor language [39 p. 154] provides full access to the CAAM's capabilities which includes:

- Hashing via the SHA-2 hash function. Supported are digests of size 256, 384, and 512 bits.
- Elliptic-curve cryptography for groups of up to 1024 bits. The CAAM has hardware support for timing-equalized versions of ECDSA signing operations and the generation of ECDH and ECDSA keys.
- Support for AES-CMAC and HMAC operations on all supported hash functions.
- Support for the AES block-cipher with 128-256 key-bits. The supported block cipher modes are CBC, CFB, CTR, CCM, and GCM.
- Access to two random number generators (RNG), both a Pseudo PRNG and True RNG.

However, without access to the SoC's security manual, it is difficult to assess and implement the CAAM's feature set on the token. Luckily, the vendor provides public access to a security reference manual for an older chip-set family [39]. A comparison between the manual for the older chip-set and the CAAM drivers provided by the Yoto project [42] with the driver provided by TamaGo [59] indicates, that the CAAM's design of both chips should be similar. Albeit, the older chip-generation lacks some features like mathematical operations on elliptic curves, lacking an alternative, this work uses the security manual of the imx6UL's predecessor [39] as a point of reference. For newer features the security reference manual of a chip from the same generation can be used [41].

2.4.4. Setting Up the token

By default, the token is in the so-called open configuration which is useful for testing. In this mode, secure boot is deactivated and the token has no access to the OTPMK. Before the token provisioner can flash the Auditor onto the token, it must program the SoC's One-Time-Programmable (OTP) fuses first. This process is irreversible as the Provisioner has to burn physical fuses on the token by using the manufacturer provided tool called crucible [60]. The available fuses are described in the SoC's reference manual in chapter 5.3: „*Fusemap Descriptions Table*" [43 pp. 205-213]. The armory-witness project implemented this process already at [19].

To activate the secure boot feature, one has to first create 4 RSA signing keys which form a Public Key Infrastructure (PKI). Three out of four can be used to sign the image andsss can be revoked later on by burning an associated fuse. The u-boot project explains the HAB's entire secure-boot architecture in detail [61] and provides tools to create such keys. The manufacturer also provides a tutorial on how to set-up secure-boot [62] and provides a tool to generate the PKI [20]. Alternatively, one can customize a manufacturer provided Makefile [21]. It should also be possible to create the entire PKI on a hardware token like a NitroKey [22] to protect the keys from exposure.

The provisioner computes a hash over the public keys, before fusing the digest into OTP. Crucially though, the provisioner does not require access to the corresponding private keys.

At last, the token's temperature monitor must be configured, if one desires to use it. For the calibration, two measurements taken in a temperature chamber suffice. The silicon provider explains how these measurements must be taken [23] and the provisioner should follow the recommended best practices.

2.5. TPM

The following introduces the Section 2.5 specific Terminology, describes attestation related commands, and their associated command and response structures.

2.5.1. Terminology

PCR (Platform Control Register)

A PCR is a volatile memory bank of the TPM which stores the digest of a hash function. A TPM2.0 device has guaranteed access to at least 24 PCRs, where each one can store a single SHA256 hashes. The platform can reset the value of a PCR, or extend the PCR's value as follows:

$$\text{PCR-Digest}[\text{id}][\text{hashAlg}] \leftarrow \mathbf{H}(\text{data} \parallel \text{digest}_{\text{old}})$$

Primary Seed

"*A Primary Seed is a large, random value that is persistently stored in a TPM; it is never stored off the TPM in any form. Primary Seeds are used in the generation of symmetric keys, asymmetric keys, other seeds, and proof values*" [63 p. 78].

NV (Nonvolatile Storage)

The NV stores persistent data. The manufacturer can for example leverage the NV to ship the TPM with self-signed certificates.

Hierarchy

All objects managed by the TPM are in a hierarchical parent-child relationship, where every object has a parent [63 p. 160]. An object's parent is either another object or one of the primary seeds, which form a group or hierarchy: "*A hierarchy is a collection of entities that are related and managed as a group. [...] Each hierarchy is targeted at specific use cases: for the platform manufacturer, for the user, for privacy-sensitive applications, and for ephemeral requirements*" [70 p. 105].

The **Endorsement Hierarchy** is a persistent hierarchy intended for privacy-sensitive operations. In contrast, the **Storage Hierarchy** is also persistent but reserved for non-privacy-sensitive operations. A user can disable both hierarchies.

(Authorization) Policy

"*A policy defines the conditions for use of an entity*" [63 p. 127]. The policy can be seen as a set of tests which must be passed, before the TPM uses the entity. A policy is a means of authorization and can for example dictate that the TPM's PCRs must have a specific value, before the TPM uses an AK for signing operations. The policy can be represented as a circuit composed of 'AND' and 'OR' gates [63 p. 128]. The authorization policy is the policies digest and is included in an object's public area.

EK (Endorsement Key)

The EK refers to a key from the Endorsement Hierarchy. The key uniquely identifies the TPM, and due to privacy concerns the TPM restricts the usage of the EK. The purpose of the EK is to endorse other keys.

AK(Attestation Key)

The AK refers to a signing key created by the TPM-device. The TPM can use the AK to for example create a signed certificate over selected PCR values.

Object Structure

The TPM stores data or a key as an object with a public and a private area. The public area contains an object's attributes, and a public identity, whereas the private area is sensitive and will never be exposed to the outside without being encrypted. For details, see [63 p. 185].

Object Attributes

Every object managed by the TPM has a set of attributes, which are stored as booleans represented by an uint32 value. The attributes are clustered into 5 classes [63 p. 185]: usage, authorization, duplication, creation, and persistence. Unused bits are set to zero and reserved for future use.

```
type ObjectAttributes struct{
    usage         [3]bool{ ❶
        sign,
        encrypt,
        restricted,
    }
    authorization [3]bool{ ❷
        userWithAuth,
        adminWithPolicy,
        noDA,
    }
    duplication   [3]bool{ ❸
        fixedParent,
        fixedTPM,
        encryptedDuplication,
    }
    creation      [1]bool{ ❹
        sensitiveDataOrigin,
    }
    persistence   [1]bool{ ❺
        stClear,
    }
    RESERVED      [21]bool ❻
}
```

❶ The usage specifies if the key can be used for signing or encryption. The `restricted` bit enforces that the key can only be used on certain commands and on objects which abide to certain format rules. For keys used in remote attestation purposes this bit must be set. For details see [63 p. 181].

❷ `userWithAuth`, and `adminWithPolicy` defines the authorization required to use the object. For details see [63 p. 184].

❸ The duplication class specifies if the object can be duplicated to another parent or hierarchy via the `fixedParent` attribute, and if the object can be exported to another TPM via the `fixedTPM` attribute [63 p. 172]. The `encryptedDuplication` bit enforces, that the objects sensitive portions are protected [63 p. 162], when the object is exposed to the platform.

❹ For details see [63 p. 183].

❺ The `stClear` bit enforces that the TPM destroys an object after it is reset with no way of recovering the object [63 p. 183]. The bit is useful to protect against attacks which require a reset of a TPM like a down-grade attack.

❻ The reserved bits provide are reserved for future use.

Public Area of a Key

The public area of a TPM key contains the public key, as well as associated parameters and metadata. The public area of a key is structured as follows:

```
type  PublicArea struct {
    objectType       [2]byte ❶
    hashAlg          [2]byte ❷

    objectAttributes ATTRIBUTES ❸
    authPolicy       []byte     ❹

    parameters       PARAMETERS ❺
    unique           UNIQUE     ❻
}
```

❶ An enumeration of the type of the key, which can be RSA or ECC.

❷ The enumeration of the hash algorithm used to compute the name of the object.

❸ The object's attributes.

❹ The authorization policy under which the object can be used. The content of the authorization policy defines how the Prover handles the key internally. Although the content is irrelevant to the Auditor, it requires knowledge to compute the key's public name.

❺ The keys parameters structure depends on the key type, and contains information about the key's parameter, but excludes the key specific data. For an ECC it contains for instance the elliptic curve's type.

❻ The public key. For an RSA key it contains the values of N and e.

Object Name

The object name is a unique identifier of the object O, which is typically the digest over the object's public area. According to [63 p. 86] for a key object it is defined as follows:

$$\text{Name}_O = \text{hashAlg} \parallel \mathbf{H}_{\text{hashAlg}}(\text{handle.nvPublicArea})$$

Qualified Name

The Qualified Object name is in essence a hash chain composed of all the object's ancestor keys, where the handle of the primary seed is at the root of the resuling hierarchy. The qualified name is defined as follows, where `P` is the parent, `PS` is the primary seed and `O` is the object:

$$\begin{aligned} \text{QN}_{\text{PS}} &= \text{PS.Name} && (1) \\ \text{QN}_{\text{O}} &= \mathbf{H}_O(\text{P.QN} \parallel \text{O.Name}) && (2) \end{aligned}$$

1. The qualified name of the primary seed at the root of the hierarchy is the digest composed of the primary seeds name, and the qualified name of the primary seed's handle. The name of the primary seed is its handle as specified in the TPM specification.
2. The name of an object is recursively defined. The object's qualified name is the concatenated hash of its parent's qualified name, and its own name.

ECDSA (Elliptic Curve Digital Signature Algorithm)

ECDSA is a signature algorithm and its security is based on the discrete-logarithm problem.

One-Pass Diffie-Hellman, C(1, 1,ECC CDH)

The One-Pass Diffie-Hellman key-exchange protocol of the ECDH-Key-Exchange protocol, which does not require bidirectional communication. The protocol assumes the Sender knows the Receiver's public key Q_{EK}. The Verifier creates an ephemeral key pair $(sk, pk) = (d, Q)$ and sends the freshly created key to the Receiver. At the end of the protocol, both participants established a shared secret S. Since the Receiver does not contribute randomness, it can not ensure freshness of the established key. In addition, an active adversary can tamper with the share sent to the Receiver. The specifics are found in the literature [72 p. 110].

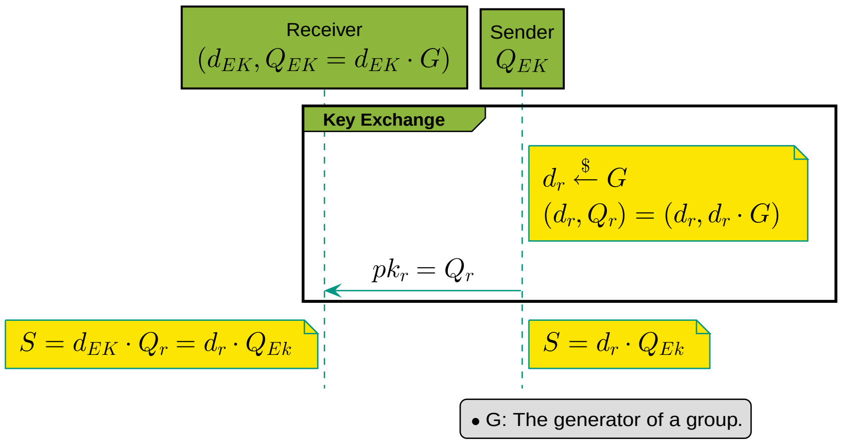

Receiver
$(d_{EK}, Q_{EK} = d_{EK} \cdot G)$
Sender
Q_{EK}
Key Exchange
$d_r \xleftarrow{\$} G$
$(d_r, Q_r) = (d_r, d_r \cdot G)$
$pk_r = Q_r$
$S = d_{EK} \cdot Q_r = d_r \cdot Q_{Ek}$
$S = d_r \cdot Q_{Ek}$
• G: The generator of a group.

Figure 1. Sketch of a One-Pass Diffie-Hellman Key-Exchange

KDFa (Key Derivation Function a)

The KDFa is an HMAC based key-derivation function. Except for ECDH keys, the TPM uses the function for all cases where a KDF is required. See [63 pp. 50-51, 76 p. 14] for specifics.

KDFe (Key Derivation Function e)

The TPM uses this function to derive ECDH keys as specified at [63 pp. 51-53].

2.5.2. TPM’s Attestation Commands

The chapter explains in detail, how the TPM implements attestation. In addition, the work points out which parameters the Auditor must check, and provides an explanation how each parameter available influences the security.

The following tables provide a synopsis of the attestation-commands which are useful to a remote Auditor. The Auditor uses a key attestation command as shown in Table 1 to establish trust in an AK which the TPM allegedly created. Meanwhile the TPM certifies the content of its PCR and NV memory via a data attestation command shown in Table 2.

Command	Input	Output	Purpose
Certify Creation [64 p. 146]	keyHandle objectHandle Nonce creationHash creationTicket [...]	certifyInfo Signature	The TPM creates proof that it created the specified object.
Activate Credential [64 p. 76]	decKey object challenge	response	The TPM responds to a challenge-response protocol with proof if the specified object resides on the TPM. The challenge is composed of an encapsulated ephemeral secret and a credentialBlob which is encrypted via the secret. The public portions of the handleDecKey is used to encapsulate the secret, whereas the credential blob contains the name of the object and the expected response. The TPM returns the response, if the TPM is in possession of the specified object.

Table 1. Key Attestation Commands

Command	Input	Output	Purpose
Quote [64 p. 148]	signingKey Nonce PCRs [...]	Quote Signature	The command attests for the digests of up to 8 PCR-digests selected. The command cannot provide evidence for the TPM's NV-memory.
Get Session Audit Digest [64 p. 150]	keys session	auditInfo signature	"*An audit session allows for the auditing of a selected sequence of commands so that evidence can be provided that the commands were executed.*" [63 p. 148]. The created evidence includes the inputs and outputs of every command executed. In other words the audit generalizes the attestation of the Quote command, since the TPM can provide evidence for all data which can be requested via a command. This includes the content's of the NV memory.

Table 2. Data Attestation Commands

Internally, the TPM handles binary-encoded data structures in little-endian format. Each entry of a TPM data structure can be viewed as TLV encoded data, where the encoding of the type and length is omitted, where possible.

A successfully executed attestation command responds with a signed certificate shown in Generic Certificate. The certificate's head contains metadata like the TPM's firmware version, and the type of attestation, whereas its tail contains data specific to the attestation type.

```
type certificate struct {
    magic [4]byte{0xff, 0x54, 0x43, 0x47} ❶

    attestationType   uint16
    certificateLength uint16

    qualifiedSigner QualifiedName      ❷
    extraDataLen    uint16
    extraData       [extraDataLen]byte ❸
    clockInfo       clockInfo ❹
    firmwareVersion uint64    ❺
    Attested        TAIL      ❻
}

type clockInfo struct {
    clock           uint64
    resetCounter    uint32
    restartCounter  uint32
    save            byte
}
```

Generic Certificate

❶ The certificate always starts with the magic value, which allows the TPM to differentiate between externally and internally created data. A user can for example use the TPM to create and manage a so-called unrestricted signing key. The unrestricted key can sign arbitrary data which does not start with the magic value. In other words, a user can use the TPM as a cryptographic co-processor, while the user cannot use the key to forge certificates in the name of the TPM.

❷ The qualified name of the Signer, which is essentially a hash of the signers name, and the qualified name of its parent.

❸ Length encoded data provided by an external Auditor to ensure freshness.

❹ The clock info contains information on how long the TPM has been runnung, the number of resets, the number of restarts since the last reset, and if the clock value monotonically increased since the last reset occurred.

❺ Contains data related to the specific attestation request, as specified in the Section 2.5.2.

The attestation of an AK must precede the attestation of actual data, since a key from the endorsement hierarchy, like the TPM’s EK, cannot attest for actual data. This chapter explains how the TPM proofs to an external entity, that it has knowledge of an AK.

Certify Creation

The TPM creates signed proof over the objects name, and the objects creation hash as shown in certifyInfo.

```
type certifyInfo struct {
    objectNameLen   uint16
    objectName      objectName ❶

    creationHashLen uint16
    creationHash    [creationHashLen]byte ❷
}
```

certifyInfo

❶ The length encoded objects name for whose creation the TPM attests. The objects name is an unique identifier for the keys public portions stored in the TPM’s NV RAM. The Verifier must be able to reproduce the keys name, otherwise the certificate is meaningless.

❷ (Type, Digest) encoded creation hash is a digest over the object’s creation data. The creation data encompasses metadata which was used to create the requested key, as defined in the specification [65 p. 206]. The verification of parts of the creationHash is optional, but highly recommended to ensure freshness. Among others the creation data contains the externally provided nonce, which was used to create the key.

ActivateCredential

Due to privacy concerns, by default a TPM uses an EK which is restricted to decryption operations. Such an EK cannot leverage the CertifyCreation command to prove that the TPM created the key in question. The TPM must therefore prove that it has knowledge of said key via the 'Activate Credential' command. In essence, the challenger uses authenticated encryption to create a credential blob, or credential challenge, and sends it to the TPM. The credential blob contains not only a random challenge, but also the name of the object to be certified. Only the TPM which is possession of the EK can decrypt the challenge and an honest TPM will only respond to the challenge, if it is in possession of the object in question. Therefore the ActivateCredential command provides the challenger with assurance that the TPM has knowledge of the object in question.

For the creation of the challenge, the TCG defines a helper function called 'MakeCredential', which works as follows. The Challenger derives an ephemeral and secret seed, which is used as an input to a key-derivation function to derive an AES encryption key, as well as an HMAC-key. The Challenger uses the encrypt-and-mac scheme to create the credential challenge. Finally, the Challenger uses authenticated encryption to protect the seed with the EK as input and waits for the TPM to respond with the decrypted challenge.

```
// Inputs the TPM's pk, the objects name and a random challenge
func MakeCredentialRSA(ekPub rsa.Pk, nameAndChallenge []byte) {
    ❶
    seed := random.bits(seedLength)
    encapsulatedSeed := rsaes-oaep.Encrypt(ekPub, seed)
    ❷
    encKey := KDFa(seed, credentialParameterss...)
    credentialBlob := encryptAndMac(encKey, nameAndChallenge)
    ❸
    return encapsulatedSeed, credentialBlob
}
```

Make Credential for RSA keys

❶ The Challenger draws a random binary string - the seed - from the uniform distribution. The Challenger encapsulates the seed via authenticated RSAES-OAEP encryption.

❷ The Challenger derives an encryption key via the TPM's KDFa Key-Derivation function from the freshly created seed, before encrypting the credential blob. The credential blob contains the keys name and the challenge in question.

❸ The TPM decrypts the received encapsulated secret first, before the TPM decrypts and verifies the credential blobs content.

```
// Inputs the TPM's EK, the objects name and a random challenge
func MakeCredentialECC(ekPub ecc.Pk, nameAndChallenge []byte) {
    ❶
    ephKey := random.NewECDH(ekPub.Params)
    seed := ephKey.d * ek.Q
    encapsulatedSeed := ephKey.Q
    ❷
    encKey := KDFe(seed, credentialParameterss...)
    credentialBlob := encryptAndMac(encKey, objNameAndChallenge)
    ❸
    return encapsulatedSeed, credentialBlob
}
```

Make Credential for ECC keys

❶ The Challenger encapsulated the seed via the a One-Pass Key-Establishment Protocol. The Auditor creates a fresh ephemeral ECDH key, denoted as $(d, Q) = (d, d \cdot G)$, which the Auditor uses to encapsulate the seed ($d_{ephKey} \cdot Q_{EK}$).

❷ The Challenger encrypts the credential blob via the encryption key derived from the TPM's KDFe Key-Derivation function.

❸ The TPM decapsulates the seed ($d_{Ek} \cdot Q = d_{Ek} \cdot d \cdot G = d \cdot Q_{EK}$), decrypts the credential blob and verifies its content.

Quote

A Quote contains a list of the selected PCR's, as well as a digest computed over the selected PCR values. An external Auditor can compute the quote's expected value, if it has knowledge about the unverifiable data residing in the certificate's head.

```
type Quote struct {
    pcrSelectionLen uint32
    responseCode    byte{0} ❶
    pcrSelections   [pcrSelectionLen]pcrSelection     ❷
    digestLen       uint16
    pcrDigest       [digestLen]byte ❹
}

type pcrSelection struct {  ❸
    hashType        byte
    selectionLen    byte
    selection       [selectionLen]byte
}
```

Quote related data

❶ The Quotes response code, which is zero. The code differs if the PCR selection count is greater than the number of PCR banks available.

❷ A length encoded list, which contains all PCR digests for which the quote attests for. In total, the quote can attest for up to 8 PCR digests.

❸ A pcrSelection structure contains a list of encoded PCR-indices for the hash-type provided. For the indice i the i'th bit is set in the array. For a PCR with 24 entries, the length of the array is 3.

❹ The pcrDigest is a hash over all PCR digests reported. The quote command computes the digest from the hash function specified in the public portion of the key which signs the Quote. The hash functions is the same as the one used to compute the key's name.

Audit-Session

An audit-session based attestation allows the TPM to create proof for arbitrary data which the TPM can report for via commands. To be more specific an audit-session based attestation provides proof that the TPM executed a list of commands in a given order. The proof furthemore extends to each commands associated inputs and outputs.

The auditInfo describes the data specific to an audit which is contained in the certificate's tail.

```
type auditInfo struct {
    exclusive     bool ❶
    sessionDigest []byte ❷
}
```

auditInfo

❶ A session is exclusive, if the TPM did not execute any command outside the session, and therefore provides atomicity of the executed command chain.

❷ The length encoded digest of the audit's session. The digest is the end of a hash-chain, where each entry represents a command executed during the session in the order provided.

Akin to the Quote based attestation, the Auditor can compute the audit's digest beforehand by emulating the Provers behavior, which is the following:

1. The Prover starts an HMAC based authorization session on the TPM. The TPM initializes the audit's session digest, a record of all commands executed so far, with a digest of value zero.
2. As part of the session, the Prover executes the commands in the order provided. Every called command has an associated cpHash, a hash of the command and it's associated argument values [63 p. 103], as well as a rpHash, a fingerprint of the TPM's response [63 p. 104]. For every command the TPM executes, the TPM keeps record, by extending the command's rpHash, followed by it's cpHash into the current session digest. If the Prover executes any TPM command, which is not recorded by the audit, the TPM sets the `auditExclusive` bit to false.
3. The Prover finishes the session, and the TPM creates signed prove via the Ak.

Computing the audit's session digest *[63 p. 148]:*

$$\begin{aligned} \text{auditDigest}_{\text{new}} &:= \mathbf{H}_{\text{auditAlg}} \left(\text{auditDigest}_{\text{old}} \parallel \text{cpHash} \parallel \text{rpHash}\right) && (1)\\ \text{cpHash} &:= \mathbf{H}_{\text{sessionAlg}} \left(\text{cmdCode} \parallel \{\text{objHandles}\} \parallel \{\text{params}\}\right) && (2)\\ \text{rpHash} &:= \mathbf{H}_{\text{sessionAlg}} \left(\text{responseCode} \parallel \text{cmdCode}\{\parallel \text{returnData}\}\right) && (3) \end{aligned}$$

cmdCode
: Specifies the name of the executed command.

objHandles
: A list of up to three associated unique object-identities. For instance the signing operation takes the handle of a signing-key as input.

params
: The parameters or inputs of the executed command.

responseCode
: specifies if the command executed successfully. The rest of the response can be omitted.

returnData
: contains the data returned by the command executed.

2.5.3. Chain-Of-Trust Measurements

A TPM attestation creates evidence for the TPM's PCR values - a list of digests. This begs the question what the TPM actually attests for and how a Prover can use these PCR values to provide a Verifier assurance over its software state.

Beginning with powering on a platform the TPM protocols the entire boot process and hashes each entry into a set of PCRs. Once the boot process is completed, the TPM locks down the access to these PCR values until the next reset occurs. Ideally, a remote Verifier can use the PCR values to verify the entire boot process. The protocol also contains so-called measurements, which are hashes of the firmware and software components loaded during the boot process. Therefore, the Verifier also learns which software has been loaded and executed on the freshly booted system. If malicious software has been loaded, the PCR values reflect the deviation and the attestation fails.

The preceding sneak peek introduced the term measured boot which is closely linked to Chain-Of-Trust measurements. The chapter introduces the term chain-of-trust measurements, different roots for trusts, and provides a brief overview over the different preexisting implementations.

In the context of TPMs, Chain-of-Trust measurements refers to the following. Starting with an piece of intitally trusted code - the so-called Root-of-Trust **RT** - performs an initial measurement of its own code. Every time the currently executed code loads fresh data from the memory, it extends the data's measurement into one of the PCRs. Once finished, the code delegates trust to the next piece of measured software. The measurements form a chain, where each link is associated with a piece of software executed in the provided order. Commonly referred to as Chain-of-Trust-Measurements (**CTM**). Intuitively, the CTM provides evidence that no malicious code has been loaded and executed, if the root-of-trust can be trusted. According to [83] a trustworthy chain-of-trust must fulfill the following conditions:

1. The root-of-trust is trustworthy and an adversary cannot tamper with it.
2. The system software cannot reset PCRs, without delegating control to trusted code.
3. The chain is contiguous. Every piece of code loaded will be extended into the PCR entries.

The TPM specification differentiates between three types of Root-Of-Trust Measurements, or **RTM**. They differentiate on where the core-root-of-trust assumptions lies, and when during the boot-process the RTM takes place.

S-RTM (Static Root-Of-Trust-Measurement)

S-RTM, commonly referred to as "*measured boot*", starts with the boot-process and ends once the boot-process is finished. The BIOS takes the initial measurement and acts as the core-root-of-trust. For the SRTM to be trustworthy, the platform must protect data exchanged between BIOS, CPU, and TPM from being altered. A Verifier can allegedly verify the entire boot process by comparing the PCR entries with the system provided event-log. The TPM locks write-access to the associated PCRs until the platform signals a reboot of the platform.

2.5.4. D-RTM (Dynamic-RTM)

D-RTM allows the system to reestablish trust into its software state after the boot process took place. A program can execute a CPU provided command, from where the CPU takes over. The CPU sets itself into trusted mode, where it cannot be interrupted by the rest of the system. The CPU starts the DRTM by setting all DRTM related PCR values to zero. As described in detail [40], the already booted system emulates the measured boot process on a provided image. At the end of the execution, the program which issued the DRTM process does not take back control. Instead, the booted image proceeds the execution. The TPM does not lock down access to the PCRs, since the image may continue with the measured boot process. Since an adversary may reset the PCRs to emulate the DRTM proccess, the PCRs associated value with the reset DRTM is -1. Only the CPU - acting as the core-root-of-trust - can initialize the DRTM, and the CPU-vendor must provide an instruction to launch the DRTM. On AMD CPUs the `SKINIT` command is available for CPU's which have the SVM extension [66], TXT-capable Intel CPU's provide the `GETSEC[SENTER]` command, and on ARM the `DRTM_DYNAMIC_LAUNCH` command can be used [67]. The ARM specification is a good point of reference and includes a security analysis.

Once the DRTM process is triggered, the CPU performs the following steps:

1. Set the CPU(-core) into a trustworthy state. The state must protects itself from interrupts and outside interference.
2. Load the SMM-code and ACPI tables from the system memory and verify their integrity.
3. Set the DRTM related PCRs to zero, followed by the initial measurement.
4. Transfer control to a measured image, which the caller passed as an argument to the CPU. Typically, the image contains a minimalistic kernel which performs a second stage measured-boot.

To be secure, the execution of the entire DRTM must be atomic from the perspective of the system. In addition, the SMM-code and the DRTM Resources Table - contained in the ACPI tables - must be trustworthy. Since the CPU typically entrusts the BIOS to provide this data, the BIOS and the platform's early boot stages must also be trusted.

Related Work

This section presents research which relates to this work.

Inspiration

The thesis originally took inspiration from Heads, which is a Coreboot based firmware solution. Coreboot [24] replaces the platform's proprietary BIOS with open source linux firmware and can supposedly fend of evil maid attacks. An evil maid attack model is seemingly similar, but strictly weaker than the one considered in this work, since the device in question is either only temporarily in the hands of the attacker or replaced by an inconspicuously looking forgery. Heads uses a TPM to seal away a symmetric secret, which can only be accessed once the platform's PCR values contain a known and trusted value. After the boot process finished, the TPM device creates a fresh One-Time-Password (OTP) to prove its own authenticity and trustworthiness to the user. By default, heads creates a user verifiable OTP code via a fork of the tpmtotp library. Follow-up work leverages a tamper resistant hardware token, which verifies the devices authenticity automatically [52]. For linux based operating systems the tpm2-totp library [68] can be installed, which creates an OTP before the user is supposed to decrypt their hard-drive. Although it is tempting to adapt an OTP solution to a remote-attestation scenario, the underlying adversarial model of the evil-maid scenario is in fact orthogonal to the remote one. In the evil maid scenario the device is initially trusted, while in the remote scenario the Verifier wants to establish trust into a previously untrusted and foreign system. The solutions discussed do not even verify the systems PCR values against a predefined state, and While the remote-attestation based solution can be used to verify the integrity and authenticity of a local device, the OTP-solutions found in the literature are not suitable in a remote-scenario.

Remote-Attestation frameworks

A variety of concurring TPM related remote-attestation frameworks do exist. Keylime provides a remote-attestation service to provide evidence which software has been booted on the Prover's platform In addition, it also performs runtime integrity measurements [25] using a so-called runtime integrity monitor by keeping track of the contents of some predefined files. It is implausible to assume that the monitor can detect a sophisticated attack, however. The TrenchBoot project offers attestation services based on CRTM and DRTM. On platforms with AMD or Intel CPUs, the project allows one to a Debian based linux system with DRTM The project furthermore has an ongoing project [26] to crowd-source a large and divers set of PRC and firmware measurements, in the attempt of „becoming a public authority available to verify BIOS/Firmware PCR values“ [27].

TPM Libraries

A variety of libraries exist which implement TPM based attestation primitives. The tpm2-tools library [28] is an open-source implementation in C which provides an interface to a computers TPM. The community provided introduction [47] helped in creating the first prototype of this work. The go-attestation is package written purely in go in the „attempts to provide high level primitives for both client and server logic“ [29] and can be used to implement remote-attestation. It is however under active development and may change its API at any time. This work decided not to use the package and instead works with underlying low-level primitives directly by using the 'go-tpm/tpm2' library [30]. The library allegedly plans to provide a 1:1 implementation of the entire TPM specification in the future.

USB Armory Mk II and TamaGo

TinyGo [31] targets a large variety microcontrollers and compiles go programs into very small binaries with a small memory footprint. With tiny go it may even be possible to reimplement this work to fit into the token’s internal RAM. It is however not fully compatible with the Go runtime [32] and compared to TamaGo very invasive, since TinyGo reimplements the entire Go compiler. In contrast only few projects are realized in the TamaGo framework. A good point of reference is the tamago-example [33] and shows how for example the network can be set up. Many code snippets including large portions of the Makefile are taken from the example. The Armory-Witness project [34] implements a verifiable and transparent logger on the Mk II. The project uses many of the token’s features and this work took clear inspiration from it.

Chapter 3. Architecture

The first Section 3.1 provides a “High-Level Introduction” to the architecture proposed by this work and introduces the parties which participate in the remote-attestation in parties. Followed by the brief Section 3.2 which explains on how the token is meant to be provisioned. The subsequent Section 3.3 takes a split view for the “Remote Attestation” where it explains the attestation once from the perspective of the token and once from the perspective of the remote Verifier, before providing some final remarks. The rest of the chapter dedicates itself to a discussion on how an Architecture may accommodate Multiple Verifier is Section 3.4, before making suggestions in Section 3.6 on a variety of changes in the architecture which may enhance the security in follow-up work considerably.

A good understanding of the architecture lays the foundation for the subsequent Security Analysis in Chapter 4.

3.1. High-Level Introduction

The chapter introduces the parties which take part in the remote attestation, followed by an overview over the architecture.

Involved Parties

Prover

The Prover is a synonym for the server which processes confidential data inside a shielded environment - the macro-enclave. The Prover has access to two network interfaces. A unidirectional network connection to the outside - referred to as data diode. It allows the Prover to receive data from the outside and - after the orientation of the data diode is reversed - to report back the final results of its computations. In addition, the Prover offers a remote-attestation service over a bidirectional network interface. The service allows an external entity to verify the Provers software state.

Provider

The Provider is the owner of the server which resides in the macro-enclave and provides access to the enclave to an external party - the Verifier. To minimize the attack surface, the provider restrict access to the attestation service through institutional measures. The network interface is exposed to a selected and monitored room, where only trusted personnel has access to.

Auditor

The Auditor is a synonym for an application executed on the hardware token Section 2.4. The function of the Auditor is to verify the enclave’s software state in the name of a remote party and to establish a shared secret between a remote party - the Verifier - and the Prover. The Provider allegedly places the Auditor inside the monitored environment, before the Auditor and the Prover participate in a remote attestation protocol. After the Auditor is convinced of the Provers state, it reports back to an external entity via a unidirectional communication interface. This interface can be the room’s camera feedback or an additional data diode connected to the outside.

Verifier

The Verifier is a synonym for a remote client which uses the enclaves computing power to perform computations on confidential data. The Verifier demands assurance that the enclave is in a known and trusted state. In addition, they require the establishment of an authenticated channel between themself and the Prover. Since the enclave cannot send information to the Verifier, the Verifier delegates the trust to the Auditor(-token).

Provisioner

The Provisioner is a synonym for the party which provides the Auditor-token - this is typically done by one of the verifier's. The term is useful in a scenario where multiple, mutually distrusting verifier's delegate trust into a single auditor. Since one party must physically interact with the token during its provisioning, the tokens provisioning cannot be performed by a multi-party computation protocol.

Reviewer

The reviewer assures itself that the enclave is physically uncompromised, before issuing a certificate. A verifier assuming that the reviewer does not conspire with the provider requests the certificate from the provider. The certificate provides the verifier with the assurance, that the enclave was physically uncompromised at the end of the setup phase. Additionally, the certificate contains the enclaves public EK and the public identifier of the enclave. The party should offer a revocation mechanism in case the enclave's trustworthiness can be compromised due to the disclosure of a vulnerability in the enclave's firmware.

The figure [architectureOverview] describes how the provider sets up the enclave and how the verifier establishes trust in the enclave's software state.

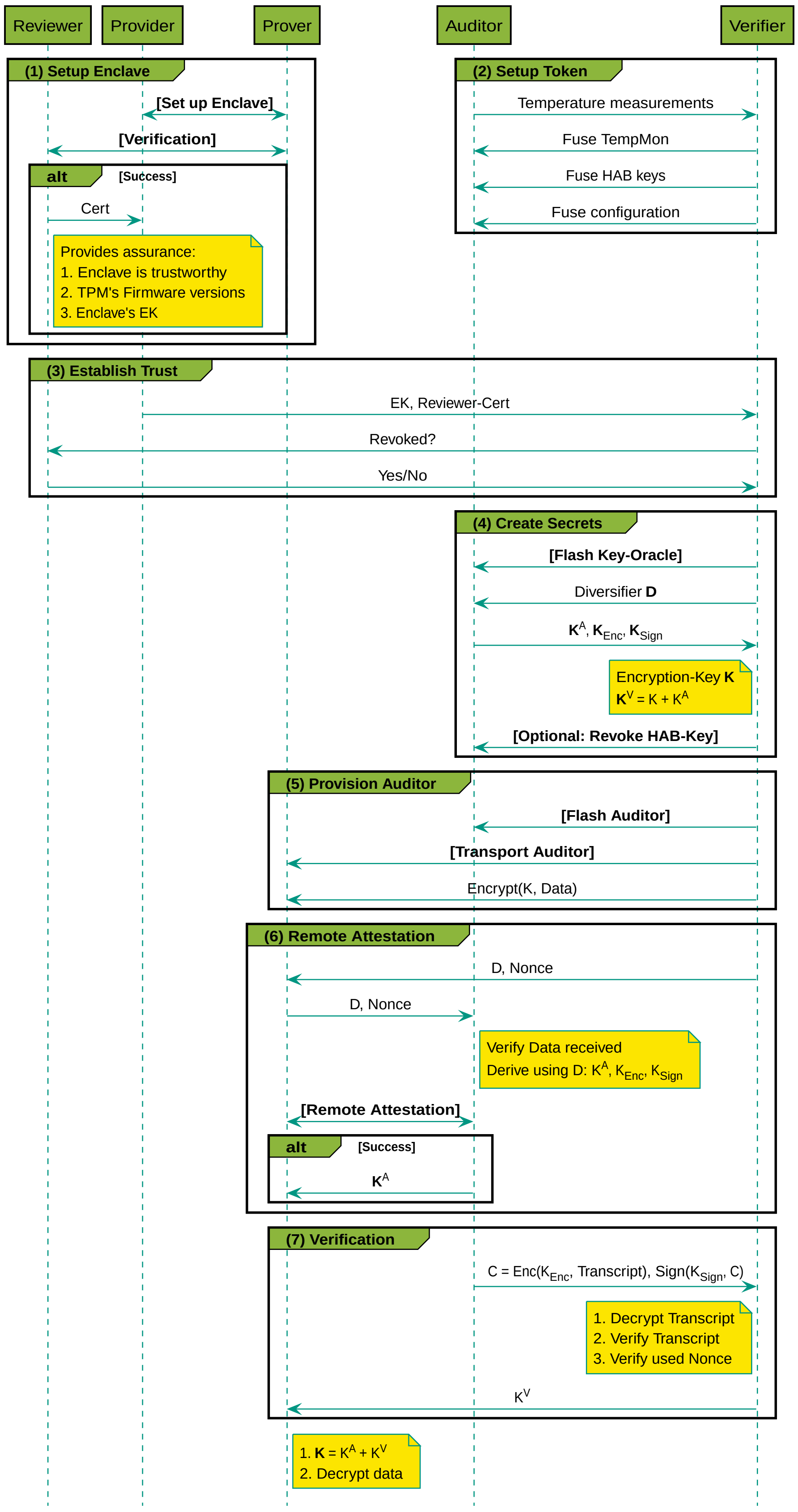


Figure 2. Architecture - an Overview

1. The provider sets up the enclave and the Reviewer assesses the enclave's trustworthiness and creates an associated Certificate. Under supervision of the Reviewer the provider initializes and seals of the enclave.
2. The Verifier sets up the token so that the Auditor application is ready to be deployed on the token.
3. The Verifier receives the allegedly trusted EK and the Reviewer's certificate. The certificate does not only provide evidence that the Reviewer assessed the enclave as trustworthy, but it also provides evidence that the EK originates from the enclave. In addition, the Reviewer should provide a revocation mechanism.
4. The Verifier flashes an application onto the token which allows the Verifier to create some authentication keys, as well as a pair of key-shares. The tokens revocation mechanisms can make sure that the Token will not boot from the image again.
5. The Verifier provides the Auditor and transports the Auditor as well as some encrypted data to the Prover.
6. The Auditor performs the remote attestation. If the attestation was successful, the Auditor sends its key-share to the Prover, which is created from the provided diversifier **D**.
7. The Verifier receives the encrypted and signed transcript from the Auditor. After the verifier established trust into the attestation, the Prover gains access to the encrypted data through the Verifier provided key-share.

After the Verifier verified the auditor's proof, the Verifier has assurance that the enclave is in a known and trustworthy state.

3.2. Provisioning and Transportation of the Token

This chapter discusses details on how the token is set up, provisioned and supposedly transported to the enclave.

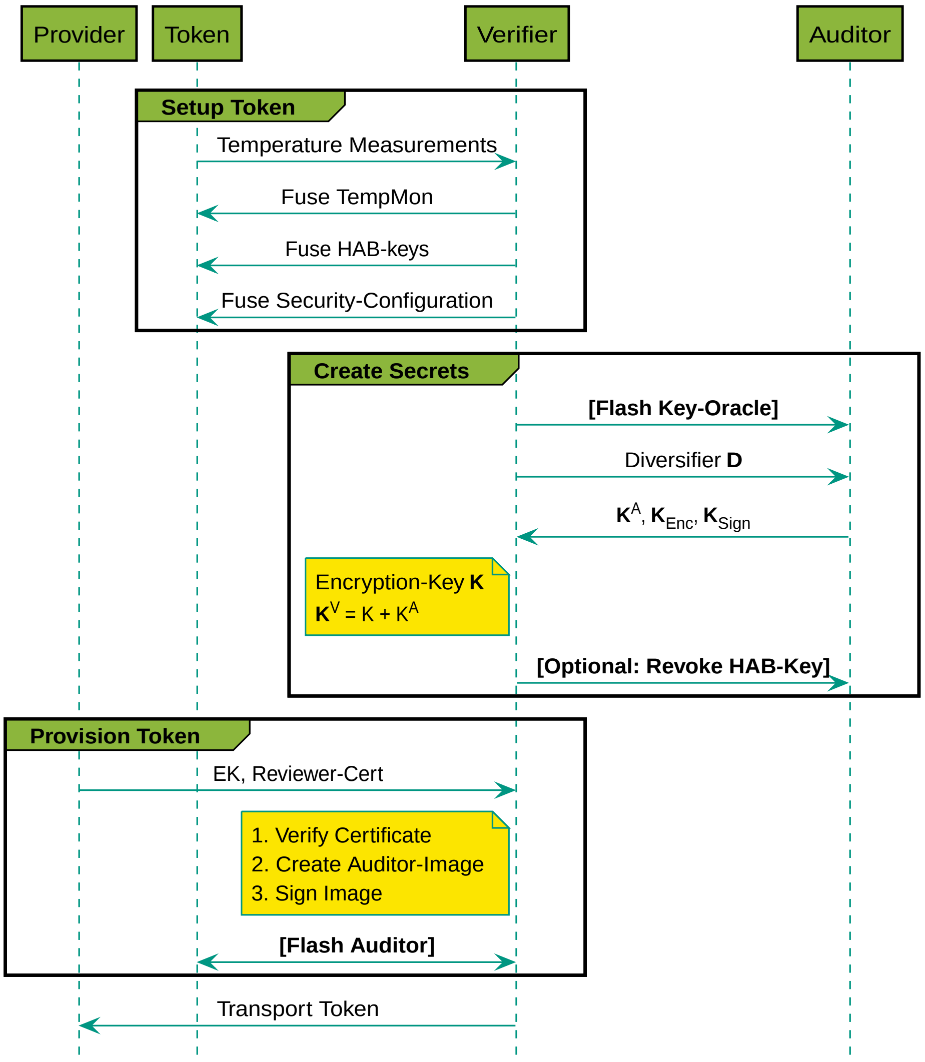


Figure 3. Provisioning of the Token

1. The Verifier configures and sets up the token as described in the section Section 2.4.4.

1. The Verifier flashes the Key-Oracle image onto the token. The Verifier uses the Oracle to establish shared secrets between the Auditor and the Verifier.
2. To establish trust into the enclave and the reported EK, the Verifier first verifies the Reviewer's certificate. In addition, the Verifier should check if the Reviewer revoked the certificate. Finally, the Verifier creates and signs an Auditor-Image. The Auditor will later on perform the remote attestation.
3. After the Verifier provisioned the token, the Verifier transports the token securely to the Prover. The Verifier can either transport the token in person or ship the token in a tamper-evident envelope.

Key-Oracle Application

The key-oracle is an application which the token executes during the provisioning phase. The oracle allows the Verifier to establish a set of shared secret between Verifier and Auditor, before the Auditor performs the attestation. It creates a set of keys - an encryption and a signing key as well as the Auditors key share, based on a Verifier provided diversifier. The key-derivation function used is deterministic and its outputs depends only on the tokens secret OTPMK and the diversifier. After the verifier received the keys, the oracle immediately erases the keys. During the attestation, the Auditor will recreate the keys for attestation purposes based on the Verifier provided diversifier. To prohibit further access to the key-oracle, the Verifier may revoke the signing key which signed the Key-Oracle image.

Auditor Application

The Auditor is an application which performs the attestation in interaction with the Prover. During the attestation phase, the Auditor receives a diversifier and Nonce allegedly send by the verifier. Before the Auditor recreates the secrets shared with the Verifier, the Auditor verifies that the diversifier has not been altered. The Verifier must provision the Auditor with some form of evidence, which protects the diversifier's confidentiality. After the Auditor verifies the Prover' software state, it sends a key-share to the Prover. Combined with the key-share of the Verifier, the Prover can reconstruct a key which allows it to decrypt the received Verifier data. Finally, the Auditor creates a transcript of the attestation performed, before sending the encrypted and signed transcript to the Verifier over a unidirectional communication channel.

3.3. Remote Attestation

The chapter describes how the token performs the remote attestation.

During the provisioning phase, the Verifier requests a fresh set secrets by querying the Key-Oracle. Combined with the Verifier provided diversifier, the token's key-derivation function uses a burnt in secret - the OTPMK - to create this key. Only a party who has knowledge of the Diversifier and who can query the token's Key-Derivation function can recreate the Verifier's secrets.

The Auditor sent to the enclave does not have knowledge of the Verifier's diversifier, it only knows evidence for the verifier's diversifier. Even if an adversary learns the evidence and queries the token's key-derivation function during the transportation phase, it cannot learn the Verifier's secrets. After the token enters the enclave, the Verifier queries a reset of the Auditor, before the attestation begins.

As visualized in the state-diagram Figure 4 the Auditor performs the attestation as follows:

1. The Auditor requests the Verifier data from the Prover which acts as an untrusted proxy, since the Auditor cannot interact with the Verifier directly. The data entails a public identifier - like an id - of the Verifier, their associated diversifier, and some Nonce.
2. Using the evidence provisioned with, the Auditor verifies the data received, before recreating the secrets shared with the Verifier.
3. The token requests a fresh attestation key from the Prover.
4. The token requests evidence from the Prover that the received AK originates from the trusted TPM. Dependent on the EK's type, the token either requests a certificate for the AK - see Certify Creation - or uses a challenge-response protocol - see Activate Credential.
5. The token requests a fresh and signed certificate over the TPM's PCR values via an Audit.
6. If the attestation succeeded, the token uses the diversifier as the input to a KDF to create a fresh authentication key, as well as a key-share. It then creates a signed and encrypted transcript of the attestation performed.
7. The Auditor sends the authenticated transcript to the Verifier, and provides the Prover with the key-share created.

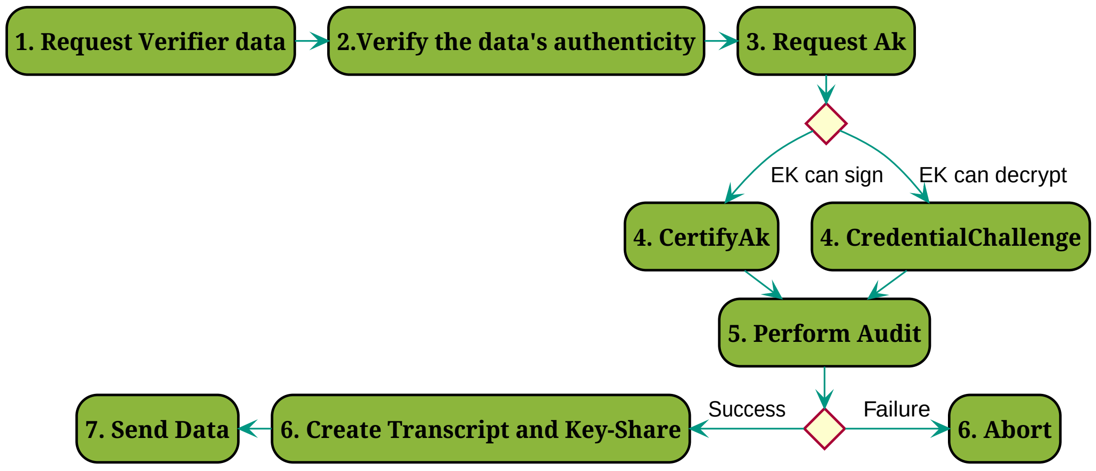


Figure 4. Attestation Flow

From the perspective of the Verifier the entire interaction with the Auditor can be viewed as querying an Audit oracle once. Given the diversifier’s public data, the Oracle creates a signed transcript of a freshly executed remote-attestation protocol. Taking everything together which has been previously discussed, the figure Figure 5 explains how the remote-attestation is performed from the Verifier’s perspective.

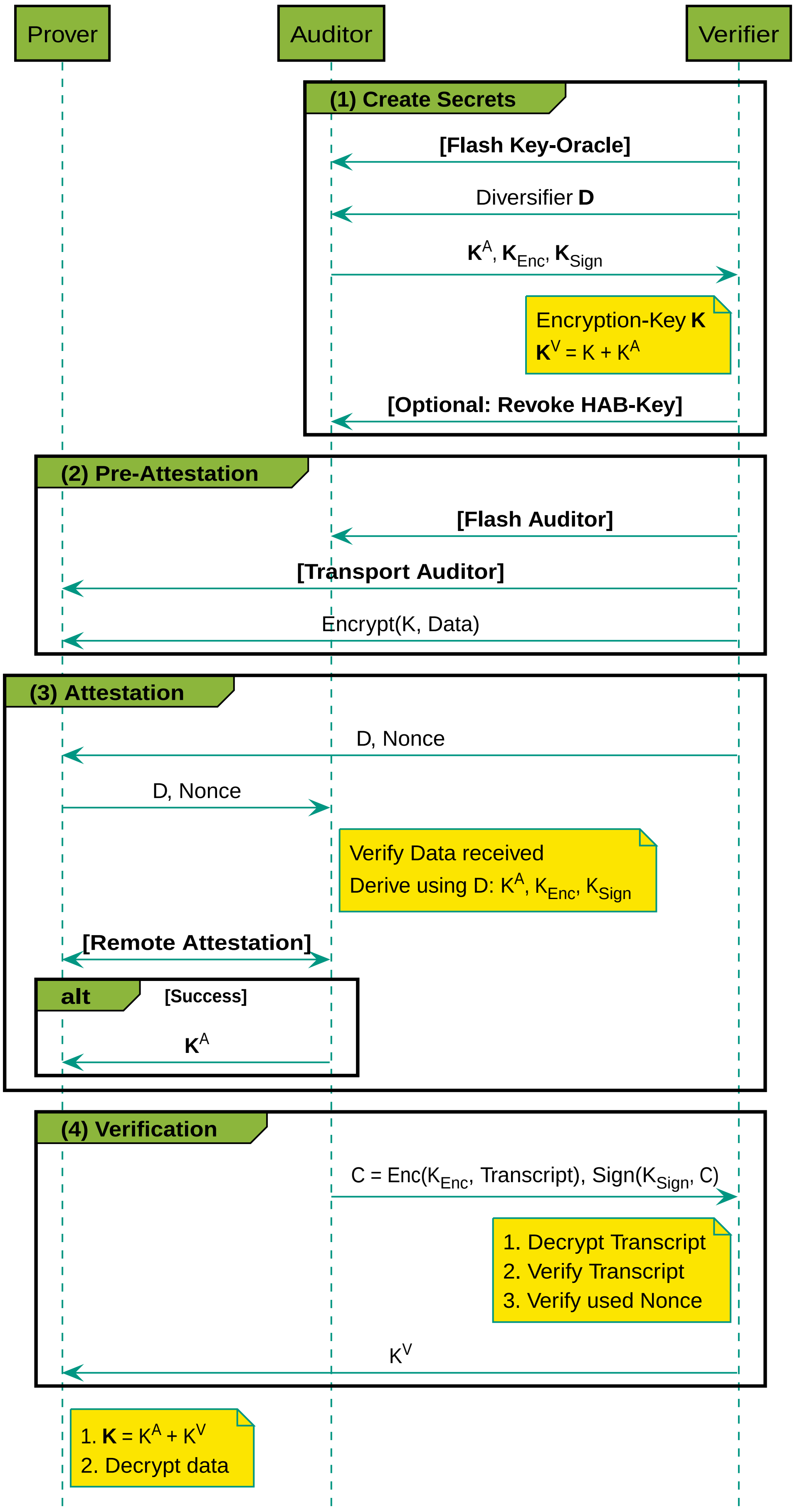


Figure 5. Attestation from the Verifier's perspective

1. The Verifier provisions the encryption oracle onto the Token. The Oracle responds with a set of secrets - including an encryption key K_{Enc}, a signing key K_{Sig}, and a Key-Share S_A. The Verifier derives a second Key-Share S_V from the Auditors Key-Share S_A, based on the decryption key used to decrypt confidential data as follows: $S_V = S_A \oplus K$.
2. The Verifier flashes the Auditor application onto the token, before transporting the Auditor to the Prover. While the Auditor is in transit, the Verifier sends encrypted data to the Prover.
3. Once the Auditor arrived at the Prover, the Verifier sends the Diversifier and some Nonce to the Verifier. Since the Verifier cannot send data directly to the Auditor it uses the Prover as an untrusted Proxy. After the Auditor verified the data received, it recreates the secrets using the token's KDF. Once the remote-attestation succeeded, the Auditor sends its own Key-Share to the Prover.
4. The Auditor creates an encrypted Transcript of the remote-attestation over a data diode to the Verifier. The Verifier checks that the Transcript is trustworthy and makes sure, that the Verifier provided nonce where included during the remote attestation. The idea is that both the Verifier and Auditor contributed some freshness. Once finished, the Vierifier provides the enclave with its own key-share.

Remarks

- The Verifier reveals its own key-share to the enclave only, after the verification of the transcript succeeded. The reason the Auditor does not hold the decryption key, but a key-share instead, is to ensure that the Verifier can abort the protocol at any time, without being at risk of leaking the decryption key for the confidential data.
- The evidence can be computed from a cryptographically secure hash-function. The diversifier's minimum length should be chosen with respect to the digest's length, since the hash-function may leak some bits of the input. A diversifier thrice the size does suffice.
- The provider may fear attacks from a malicious Verifier and therefore restrict access to trusted personnel only. Consequently, the Verifier may not be able to transport the token in person to the location where the attestation takes place.
- The Verifier may use a tamper evident envelope and a hard to forge pattern printed onto the token, to establish trust into the token's authenticity. During the envelope's opening, the Verifier can monitor the envelope's opening in the camera feedback where it sees the pattern printed on the token. The motivation is that a skilled magician may be able to replace the token with a forgery temporarily. The nail polisher however, prevents the magician from creating an inconspicuously looking forgery and the Verifier discovers the attack, before the magician can launch a code-injection attack.
- The communication channel between Prover and Auditor should be confidential. The Auditor can for example request a fresh and ephemeral key from the Prover's TPM, to send an encrypted key-share to the Prover.
- The Verifier and Prover should establish an authenticated channel, before the Verifier sends its own key to the Prover.
- Since the diversifier is public after the attestation took place, the adversary may attack the Auditor after the attestation succeeded. If the adversary knows the Verifier's key-share, it can exploit the token and decrypt all encrypted messages which the Verifier

send to the Prover. To minimize the risk of such an attack, the Auditor can revoke the HAB's signing keys. Since the HAB refuses to boot from there on from any image, the adversary cannot use on any of the attacks found in this work.

- In case the Auditor gets compromised during its transportation, the token should provide a mechanism to reset the token into a trustworthy state. The Verifier should have means to initiate and monitor the Auditor's reset. For example the Verifier may instruct personnel which connects the token with the Prover to initiate the reset.
- The Verifier provided nonce must influence the attestation to provide a fresh transcript for the Verifier. In case the Auditor is compromised, the transcript still provides evidence that the Prover is trustworthy.
- The Prover should also attest for data which it allegedly received from the Verifier. If the Prover is trustworthy, the transcript provides evidence, that no malicious party attempted to send tampered data to the Auditor. Such a trusted Prover acts is some sense as a hardware firewall between the outside world and the Auditor, since a compromised Auditor cannot receive data from the outside without the Verifier noticing it.
- The reason for why the Auditor signs the transcript is, that it provides fresh prove that the trusted token created this signature. If however the Auditor has been replaced with a forgery, the forgery cannot have created this signature. Albeit an adversary with full access to the trusted token can steal its secrets, it must have means to inform the forgery about the freshly created keys. Since the Prover acts as a firewall it cannot do so however. The only chance the adversary has in convincing the Verifier is to sign the transcript while it is send to the Verifier.
- If the diversifier is used to create an ephemeral secret which the Prover extends into one of the TPM's PCRs, the forgery will inadvertently create a transcript which will be rejected by the Verifier, even if and adversary can forge the Auditor's signature during transit.

3.4. Architecture with Multiple Verifiers

This section briefly discusses how the architecture may accommodate multiple Verifiers who delegate trust into the same Auditor-Token. The scenario is of interest where different entities plan to perform computations on their joint inputs. In the context of medical data for instance, private and public institutions from different regions counseled cooperate to train a single machine learning model on their joint dataset. The institutions mutually distrust each other and each one of them is a distinct Verifier from a set of Verifiers. To reduce the cost and to maintain scalability, the involved parties - mutually distrusting each other - delegate their trust onto the same Token - the Auditor.

Intuitively speaking, the Auditor can serve an arbitrary amount of parties, as long as the Auditor has been set up honestly and the parties have access to a trusted source of entropy. Each one of the Verifiers interacts with the Oracle in private to create its own Diversifier, while the nonce drawn at random is shared between all Verifiers. From the perspective of the Auditor there is little change, except that it receives now a set of Diversifiers instead of a single one. Once the attestation succeeded, the Auditor sends a set of key-shares to the Prover and reports the signed transcript back to the Verifiers. For each one of the Verifier, the Prover creates a fresh signature with the associated signing key. After a Verifier established trust into the Prover's state, it sends their own key-share to the Prover. Once the Prover reconstructed all decryption keys, it decrypts the Verifier provided data.

During the setup phase all Verifiers delegate their trust into a single entity - the Provisioner. The Provisioner fuses the HABs public keys onto the token, calibrates the Temperature Monitor, and flashes the signed images onto the token. At first glance the Verifiers have to trust the Provisioner completely, but this is not entirely true. Assuming that the Provider of the enclave does not conspire with the Provisioner or the Verifiers, far weaker assumptions are necessary.

During the setup phase, each Verifier must make sure that it communicates with an authentic Token with no physical backdoors who launches a trustworthy Key-Oracle image. Crucially though a Verifier does not have to confirm that the token in the setup phase is the same as the one who Audits the enclave. First off, a forged token who acts in the role of a malicious Auditor, does not have access to the OTPMK of the token used during the setup phase. In other words the forgery cannot recreate the setup tokens secrets on its own and must interact with a compromised setup token to retrieve the secret shared between verifier and setup token. The enclave provider - who does not cooperates with the adversary - should have means to prevent this type of interaction, thus the attack fails. Secondly, even if the compromised Auditor has access to the shared secrets, it still must provide the Verifier with a fresh and verifiable transcript attestation. To summarize, the architecture can protect against a compromised Provisioner who replaces the Auditor-Token with a forgery. The Verifier must only make sure that it communicates with an authentic and uncompromised token during the setup phase.

A Verifier can establish trust in the setup token with the help of the token's manufacturer. The manufacturer can print a uniquely identifiable and hard to forge pattern onto the token, and ship the token in a tamper evident envelope. When the Provisioner receives the envelope, the Verifiers can physically meet up and monitor the Provisioner.

The Verifiers provide the Provisioner with public HAB keys and a signed Key-Oracle image, which the Provisioner flashes onto the token. Each verifier inspects the token individually. During the inspection, the verifier reads out the state of the tokens fuses [69] and confirms that they abide to expected values. The Key-Oracle image may provide the verifier with an appropriate interface to read out the relevant fuses. After the verifier rebooted the token, it requests fresh secrets from the Key-Oracle via a trusted network connection. The setup phase is completed, when the Provisioner allegedly flashed the provisioned Auditor Image onto the token. To enhance security, the Verifiers may revoke the key used to sign the Key-Oracle.

The question remains, how the Verifiers create and sign an image without leaking the private key to one of the Verifiers or the provisioner. The Verifiers can leverage a multi-party computation protocol to create a set of fresh signing-key. Each one receives a private key-share of the created key, as well as an associated public key. During the provisioning phase the Verifiers participate in another protocol to sign an image created via a reproducible build system With the Verifiers private key-shares and the public image as input, the protocol creates a fresh signature.

3.5. Putting more trust in the Token

The current architecture puts very little trust into the Auditor, as it assumes that the token cannot protect its own secrets during transportation. A polar opposite approach is to put more trust into the token by provisioning the Auditor with erasable secrets stored in the token's SNVS. Once a security violation is detected or the Auditor fulfilled its purpose, the token's hardware destroys the secret. The approach has two advantages. First, if the token's hardware can reliably destroy the secret before an adversary can learn the secret or compromise, the token is tamper-evident. Second, the approach protects against any recovery attempts, since the Auditor erases the secret after the attestation took place. To extend the idea even further, the Auditor may not provide the Verifier with a transcript of the attestation. Instead, the Auditor creates a small tag over some Verifier provided nonce. In an environment where the outgoing communication channel is heavily constraint in size, the approach is preferable - an AES based CMAC tag for instance requires just 16 bytes.

The aforementioned approach was in fact the original design of this work, but resulted in a very unstable implementation. Once the token loses access to a power supply or detects voltage fluctuations due to sporadic movements on the power supply cable, the Auditor is inoperable. If the token would have access to a reliable always-on power-supply provided by a coin-cell, the token could be transported to the Prover without being powered over a USB connection. To minimize the risk of an accidental erasure of secrets during the Transportation, the token's designated power supply port should also compensate for sporadic voltage fluctuations. Additionally, it is unclear how the approach can be securely scaled to a setting with multiple Verifier's without a loss in security.

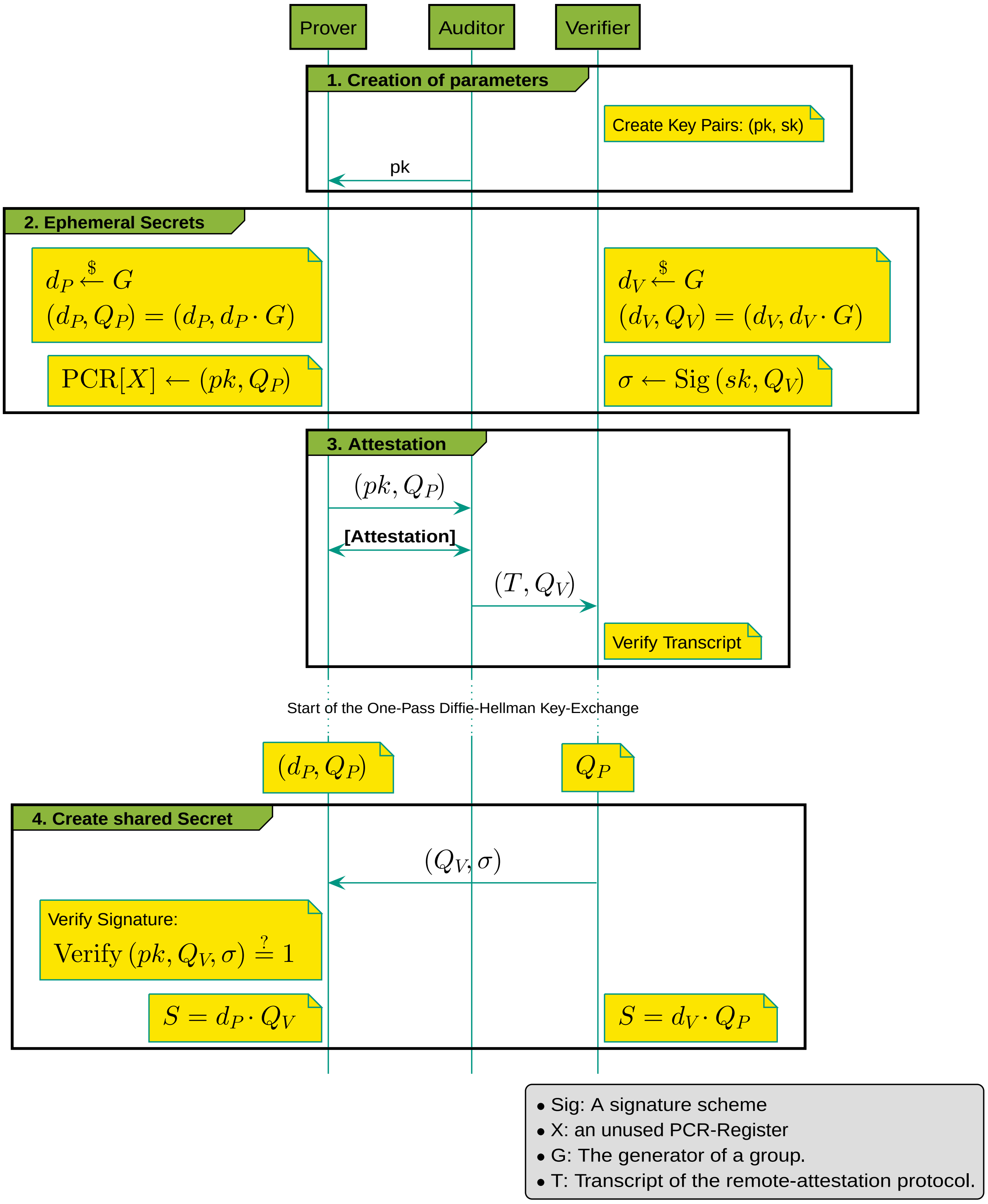
It is worth mentioning, that both approaches - the current architecture and the aforementioned proposed - can be combined to enhance security further. By using an additional erasable key-share which the Auditor is provisioned with, the Auditor can send both the erasable key-share, as well as the one created from the Verifier provided Diversifier to protect against a key-recovery attack. The idea is, that an adversary cannot retroactively reconstruct the established decryption key, since the Auditor erased the third key-share after the attestation took place. It is left for future work to implement such design on a more stable platform.

3.6. Improving the Key-Exchange

This section discusses how a key-exchange may be realized in this architecture and discusses potential trade-offs between practicability and security.

TPM-based key-exchange: The current architecture does not provide any protections against MITM attacks. In the current architecture, an adversary can steal the Prover's key-share by intercepting traffic between the Auditor and the Prover. Even worse, if adversary acts covert then the unsuspecting Verifier will reveal its own key-share to the adversary as well, since the Verifier must send its key-share over an unprotected channel to the enclave. Without a key-exchange protocol in place, the key cannot be protected. Since the Auditor has already knowledge of the Prover's EK, it can request a fresh asymmetric encryption key from the enclave, before verifying the key's trustworthiness. Afterwards, the Auditor sends the encrypted key-share to the Prover. Since the Verifier receives the entire transcript, the Verifier can use the same key to encrypt its own key. The approach is however vulnerable to key-recovery attacks, since an adversary may compromise the enclave in the future. Once the adversary corrupted the enclave, it can use the transcript to recreate the encryption key. The adversary can then instruct the TPM to decrypt the encrypted key-shares. The Auditor must therefore request the encryption key from the TPM's ephemeral hierarchy, which destroys all associated keys between reboots in a way in which they cannot be recovered. The current TPM specification does not support asymmetric cryptography which is Post-Quantum Cryptography, and therefore a purely TPM based key-exchange is unfavorable.

Key-Exchange protocol Alternatively, a key-exchange protocol between Prover and Verifier can be utilized. The Auditor acts as an untrusted proxy between both parties. A Key-Exchange protocol - like the Diffie-Hellman protocol - can be used in combination with a one-time signature scheme, to establish a shared secret between Prover and Auditor. For post post-quantum security, an appropriate Key-Exchange protocol must be used. The figure Figure 6 outlines a conceptual key-exchange which is based on the Diffie-Hellman key-exchange and works as follows.



Prover
Auditor
Verifier
1. Creation of parameters
Create Key Pairs: (pk, sk)
pk
2. Ephemeral Secrets
$d_P \xleftarrow{\$} G$
$(d_P, Q_P) = (d_P, d_P \cdot G)$
$\mathrm{PCR}[X] \leftarrow (pk, Q_P)$
$d_V \xleftarrow{\$} G$
$(d_V, Q_V) = (d_V, d_V \cdot G)$
$\sigma \leftarrow \mathrm{Sig}\,(sk, Q_V)$
3. Attestation
(pk, Q_P)
[Attestation]
(T, Q_V)
Verify Transcript
Start of the One-Pass Diffie-Hellman Key-Exchange
(d_P, Q_P)
Q_P
4. Create shared Secret
(Q_V, σ)
Verify Signature:
$\mathrm{Verify}\,(pk, Q_V, \sigma) \stackrel{?}{=} 1$
$S = d_P \cdot Q_V$
$S = d_V \cdot Q_P$
• Sig: A signature scheme
• X: an unused PCR-Register
• G: The generator of a group.
• T: Transcript of the remote-attestation protocol.

Figure 6. Remote-Attestation based Diffie-Hellman Key Exchange

1. The Verifier generates a fresh signing keys and sends the associated public key to the Prover over an insecure channel.
2. Both the Verifier and the Prover create a set of ephemeral secrets. The Prover extends the Verifier provided public key, as well as the ephemeral public key into an unused PCR register.
3. The Prover sends both public keys to the Auditor, which requires knowledge of said keys to verify the PCRs values. After the attestation finished, the Auditor sends the transcript to the Verifier together with the Prover's ephemeral key. After the Verifier verified the transcript, the Verifier knows that the Prover is in an expected state which the Verifier trusts. In addition, the Verifier has evidence, that the Prover received the Verifier's public key and that the ephemeral key received has been created by the Prover.
4. The Verifier performs a modified version of the Diffie-Hellman Key-Exchange protocol. The Verifier or Sender - has knowledge of the Receiver's key, which in this case is the Prover. After the Prover received the Verifier's ephemeral key, both established a shared secret. The signature prevents against tampering of the ephemeral key.

A combined approach: To be cryptographically secure, a Key-Exchange protocol requires additional cryptographic assumptions and may be broken in the future. With the rise of quantum computer's there is a significant risk that a asymmetric primitive deemed to be secure, will be broken in the future. For signing schemes used in remote-attestation the long-term security is secondary, since an adversary cannot use a quantum computer in the future to forge a signature right now. But for encryption this does not hold true, since an adversary can keep a record of all encrypted messages until the encryption scheme is broken. The architecture proposed in this architecture does however provide long-term security, assuming that the Auditor is tamper-evident and that an adversary cannot launch an MITM attack between Auditor and enclave. To maximize security a combined is therefore favorable. Both the Verifier and the Auditor establish an authenticated channel with the Prover, before sending their respective key-share to the Prover. An adversary in the known must therefore invade the enclave or compromise the token, in the hope that the key-exchange protocol used will be broken in the future. For a rational adversary, the potential return might not be worth the effort. If the proposed combined approach can discourage an adversary to even attempt the effort, the architecture will protect against an adversary even if the scheme will be broken in the future.

Chapter 4. Security Analysis

The chapter starts with a security analysis of the TPM's security guarantees which it allegedly provides, followed by a discussing of the mkII's security and limitations.

The rest of the work analyses the security of the overall architecture, which starts with defining the Security Model of this work. The model introduces security goals and stakeholders, as well as the associated adversarial model, before discussing threats Threats and Mitigations.

4.1. Security of TPM based Remote-Attestation

One of the trust assumptions is that the TPM can be trusted and this work assumes that the TPM provides **Immutability**, **Exclusive Access**, has **No leaks** and that the mechanism is **Trustworthy**. In addition, the TPM and the CPU manufacturer must make sure, that the DRTM can only be **Invoked** by the CPU once it entered a trusted state. The chapter explains what security primitives a TPM based Remote-Attestation offers and discusses weaknesses of the TPM implementation. The chapter is structured in accordance to the security primitives a secure Remote-Attestation protocol must realize.

4.1.1. Attested Information

This chapter dicussess which information attestation mechanisms provides to the Auditor. A secure attestation mechanism must provide the Auditor with comprehensive, atomic, and fresh information.

Comprehensive: This property depends on the software which implements the DRTM or the SRTM. The DRTM which loads and executes from a provided image is implemented by the CPU's manufacturer and lies outside of the Prover's control. Since the image loaded by the CPU is restricted in size, it may act as a second state bootloader. Therefore, both the CPU's manufacturer and the image provider must implement the DRT measurements properly. Crucially, the image must also measure all software components which may be loaded after the attestation takes place. Regardless how the DRTM is implemented, the measurements can only attest for code and configurations which have been loaded through known channels. However, malicious code will not be measured into the PCRs, if an adversary can exploit a running program which has already been loaded and measured. In other words, once an adversary can exploit the program during runtime, they can evade detection completely. The existence of such an exploit is quite realistic for projects with a large code basis, especially when the programs code - like the linux kernel - is written in a memory unsafe language. The Auditor must make sure that the attesting program is trustworthy and free of any exploitable vulnerabilities. Therefore, the Auditor's code and the underlying drivers should be written in a memory safe code. In addition, the network interface between all three participants should be as small as possible.

Atomic: An Audit is atomic in the sense that it cannot be interrupted without leaving some evidence behind. Albeit the entire attestation protocol requires multiple steps where the Auditor first establishes a trusted public key, this can be viewed as a setup phase. As long as the TPM can protect the requested AK in the short-term, the interaction before the Audit takes place does not affect the security. Therefore, the TPM provides atomic proof over the TPM's PCR values. The manufacturer must ensure that the CPU cannot be interrupted while the DRTM boots from a provided image. Once the image has been booted, the image must continue the DRTM until it measured every piece of software and configuration which may used during its execution. If the image continues the DRTM, it must protect itself against invocation.

Freshness: As discussed in the preliminaries, every TPM certificate includes some nonce from an external entity to ensure freshness. From the perspective of the Auditor, every certificate which the Prover creates is therefore fresh. This does not mean however, that the data attested for is fresh. In the context of the Static-Root-of-Trust-Measurements this poses a problem, since the chain-of-trust ends once the boot process is finished. To speak in extremes, a platform which ran for an entire year could have entered some unattested state during its execution. In the context of DRTM however, a Verifier can ensure that the DRTM occurred recently, if the mechanism which implements the DRTM accepts some nonce provided by the Auditor. More pressing is the question of how the Auditor can verify the freshness of the Prover's AK. An old signing key may be compromised due to some preexisting but small side-channel. Once the Auditor trust such a key the Prover can cheat in the remote-attestation. But not all TPM's do protect their secrets properly, which the subsequent section Section 4.1.2 shows. The side-channel leakage may be acceptable however, if the Auditor has assurance that the attestation key is fresh.

To recap, in the TPM's architecture the nonce used to create a key is not stored in a key's public area, but instead in its creation data. The Prover must therefore provide evidence which links the requested key to its creation hash. The TPM's Certify Creation command creates the evidence required, but - presumably due to privacy concerns - the Activate Credential does not. For a secure remote-attestation protocol this can be seen as a design flaw at the architectural level.

The Auditor may therefore prefer the usage of the Certify-Creation command. The logic is flawed however, since the Certify-Creation command requires that the TPM's EK is also a signing key and therefore also affected by the side-channel leakage. The Auditor can resolve the issue by creating a fresh authorization policy. A key's authorization policy is a digest contained in its public area. The idea is, that the Auditor can choose a policy at will, as long as the policy can be fulfilled. A random tautology like $\text{Nonce} \overset{?}{=} \text{Nonce}$ should suffice. The implementation of said workaround is left for future work.

4.1.2. Protection of secrets

The enclave's TPM is the source of trust for the Auditor. Once an adversary steals the TPM's EK or an established AK, the adversary can forge arbitrary attestations. It is of utmost importance, that the TPM protects its secrets at all costs. To increase the trust in the TPM and to motivate the manufacturer to prioritize security, the TPM's implementation installed inside the enclave should be under strict public scrutiny. However, a systematic black-box analysis of the TPM ecosystem [90] indicates, that many vendors do report significant changes in between firmware updates - including fixed security vulnerabilities. In fact this work is not aware of any open-source TPM implementation with exception of a partial FPGA implementation called TwPM project [35]. As of this writing, it is impossible to publicly verify a manufacturer's security claims. Even worse, two studies [84, 90] uncovered severe security vulnerabilities in certified TPMs in the past, including one vulnerability compromising private portions of signing keys. After observing the creation of less than 40,000 signatures the researchers extracted the signing key from a EAL4+ certified discrete TPM [84]. The EAL or 'Evaluation Assurance Level' is a numerical grade linked to a security evaluation [36] - „*The intent of the higher levels is to provide higher confidence that the system's principal security features are reliably implemented*" One of the results of the black-box analysis is, that some of the examined TPM's show signs of timing side-channels [90]. The work theorizes that it may be not only be due to a lack of public oversight, but "*that formal verification of only the specification, but not the implementation, is required to achieve higher EALs*" [90 p. 3]. This poses a tangible threat to the Verifier, since the Verifier either has to believe that all TPMs with a high certificate are trustworthy or that the Provider who chose the enclave's TPM did not install a TPM which is affected by an undisclosed security vulnerability.

Even if the enclave's TPM provides perfect security against attacks, this does not mean that TPM can protect the attestation protocol against an adversary. On the contrary, the adversary has three other meaningful ways to cheat on it.

If the Verifier trusts an EK other than enclave's one, all security guarantees are void. Albeit the Verifier can request a certificate from the manufacturer, the certificate proves that the EK originates from an authentic TPM device. It is of the adversaries own choosing, from which TPM the EK originates. Once an adversary stole the EK from another TPM, the adversary can impersonate an authentic TPM and cheat during the remote-attestation. Consequently, the Reviewer must create a certificate which links the enclave to the enclave's public EK.

TPM's are suspectible to downgrade attacks. If an insecure firmware-version exists an adversary may be able to compromise the TPM's private keys. If the TPM public key - the EK - is affected, the entire platform must be viewed as compromised. For a fresh AK, the existence of a downgrade attack may not pose a security risk at all. In addition, the recent TPM specification allows for so-called firmware-limited keys, which are bound to a specific firmware version. Such a key should mitigate potential down-grade attacks. On older platforms an authorization policy may be used.

The enclave must protect the platform from being tampered with. To be more specific, a MITM attack on the Bus allows an adversary to tamper with the values extended into the PCRs. An adversary can alternatively try to corrupt the bootloader's initial boot-code or the CPU's firmware. Most platform's should be able to protect their firmware against purely software driven attacks. With software means alone, this type of attack should be unlikely to succeed.

4.2. Security of the USB Armory Mk II

The Mk II has protection mechanisms against of physical tampering. This section discusses the different protection mechanisms, and their limits. The protection mechanisms fall into three categories: Disclose of Side-Channel leakage, physical tamper resistance, and authenticated code execution.

Protection of secrets

The CAAM's protection mechanism provides believable confidentiality guarantees to its managed secret's, and it's associated cryptographic operations. Even if an adversary takes control over the token's execution flow, the adversary supposedly cannot extract the token's secrets, and crucially the token's authentication key. This feature significantly enhances the implementation's security guarantees, since the adversary is able to briefly tamper with the token's memory with the goal of leaking the token's secrets. Without tamper detection of the external memory, an adversary can forge arbitrary signatures, by taking control over the execution flow, and then proceed with querying the CAAM as an oracle.

Security of the Secure-Boot Mechanism

The token has different mechanisms to enforce that it executes authenticated code only. The **High Assurance Boot** (HAB) works similar to secure-boot, and enforces that the token boots signed images only. The hardware reset logic enforces that the boot process starts from the on-chip boot ROM as followed [43 p. 231]:

1. Before starting the boot-process, the ROM checks first that the platform does not wake up from the low-power mode.
2. The ROM locates the signed image and loads the program into the RAM. Typically, the image is located in the persistent eMMC memory.
3. The HAB verifies the authenticity and integrity of the signed image using a Public-Key-Infrastructure composed of up to 4 keys. The Hab verifies the trusted keys - which the image provides - by comparing the hash over the keys with an expected digest. The expected digest is fused into the hardware by the token's provisioner.
4. The HAB executes the program stored in the image. In this implementation, the program is a second-stage bootloader which loads the program into an encrypted section of the RAM.

   The boot process is vulnerable to tampering after the HAB verified the loaded image, since the image loaded into memory must activate the tamper mechanisms. An adversary can therefore tamper with the token or inject the code after the HAB loaded the image into the code and before the loaded image activated the tamper intervention mechanism. The token's provider confirmed this weakness and recommends implementing a second stage boot loader which fits into the internal memory. Due to the large binary size of Go applications, such a bootloader cannot fit into the imx6UL's 128kB internal RAM. The processor's documentation does not indicate, that the imx6UL protects the internal RAM. Even if the entire application would fit into the internal RAM however, an attacker can still induce faults to tamper with the RAMs internal content or use invasive attacks to gain access to the internal memory.

Although such a bootloader would reduce the attack vector, the adversary may still be able to induce faults or tamper with the internal RAMs content. Once the program is running, the token must protect the RAM's integrity, since otherwise secure-boot does not over meaningful security during run-time.

Authenticated Code Execution

An adversary who can execute arbitrary code on the token may not be able to steal secrets protected by the CAAM, but instead use the CAAM as an oracle to forge arbitrary responses. In the context of remote-attestation, the adversary can use the compromised token to sign arbitrary Verifier provided nonce. In other words, it is of utmost importance that the token protects itself against code injection under any circumstances.

Although the token has access to a variety of tamper-detection mechanisms, neither the HAB nor the RTIC provide sufficient means to protect the RAM's content from code injection. During the boot process, the HAB verifies an image created by the TamaGo framework. Once the HAB loaded the image however, no protection mechanisms are in place to protect the image residing in the RAM. Even after the booted image (re-)activated the RTIC, an adversary can still tamper with an applications Heap and Stack by probing the Bus. Even worse, the RTIC does leave the memory unprotected for an extended period of time, since the RTIC reads out the memories content piece-by-piece. In other words, once an adversary has unmonitored access to the token for an extended period of time, the token must be deemed compromised. In a monitored environment, the theorized attack should not be possible. First, it requires conspicuously looking tools to tamper with the Bus communication. Second, the attack presumably requires some time to set up, since the adversary must first learn the application's exact memory layout - obscured by the RAM encryption - before the adversary can inject malicious code.

Protection of Memory

The imx6UL's RAM encryption protects the RAM's confidentiality, but not its integrity. The RTIC can allegedly protect an applications code, but not its Stack or Heap. Due to the existence of so-called ROP attacks [37], an adversary can hijack an applications control flow by tampering with the applications Stack or Heap alone. Even worse, a sophisticated MITM attack on the token's Bus can evade detection by reporting false values for memory sections currently scanned by the RTIC. After an adversary hjiacks the control flow, the adversary can make himself persistent by injecting code in an unprotected memory region of the token and mark it as executable. In other words, the token is vulnerable to code injection during runtime. Interestingly, an adversary with control over the token's execution flow cannot steal CAAM protected black keys.

Obstacles to an arbitrary Code-Injection

An adversary with physical access must launch the initial code injection successfully. Three obstacles remain:

1. The adversary must avoid detection from the RTIC, which periodically reads out the external memory areas. Due to the deterministic nature, the adversary should be able to identify which memory areas to avoid during screening by monitoring the token's access patterns.
2. The initial code injection can only be performed on the executable memory areas. Due to the memory encryption and ASLR (Address Space Layout Randomization), the adversary does not have access to the exact memory layout, and in turn does not

now the memory's content. As a result the adversary is at risk of crashing the application, which can cause the destruction of the token's secrets, due to a time-out of the RTIC monitor. However, the adversary should be able to circumvent the problem by carefully monitoring access patterns to the memory, and only hijack the code, once it identified safe memory areas to inject code to.

3. The adversary must make himself persistent in an unmonitored memory area. This can be achieved by allocating memory inside the Dynamic memory area, where the adversary stores its malicious code. Once the adversary marks the memory area as executable, the adversary restores the content's of the monitored memory areas, and sets the instruction pointer's to the newly allocated memory. The adversary has now full control over the token.

Remark that the described attack is generic enough to exploit the HAB's secure boot feature. However, it is far more difficult to launch the attack described against the internal RAM, since it requires invasive tampering. It would be of tremendous help, if the Go application would fit into the internal RAM. With the size of current Go applications, this however is infeasible.

Physical Tamper-Resistance

The token's Secure Non-Volatile Storage (SNVS) provides a tamper-intervention mechanism, which can lock down the token upon tamper detection, and erase stored volatile secrets. The application must set a security policy which activates the behavior upon tamper-detection. The token allows for monitoring the integrity of memory areas, which is a feature independent of memory encryption. After activation the feature cannot be deactivated anymore, which protects the token from code injection. However, the token does verify the memory's integrity only upon every few clock cycles, which means that an attacker has a small attack window where it could exfiltrate data.

In summary, it is plausible to assume, that the token can protects the confidentiality of secrets, if the CAAM handles the secret's internally. However, an adversary, with physical access to the token, can circumvent the HAB's secure boot, and therefore launch arbitrary code after resetting the token. Luckily, this is not a problem, since the provisioner can ship the activated token to the enclave, and prohibit the usage of non-volatile secrets. Due to the limitations of the RTIC, an adversary with unrestricted physical access, can hjiack the token's execution flow. Therefore, the token can neither protect itself against relay attacks, nor against sophisticated MITM attacks on the bus.

In conclusion, the token's transportation mechanism must guarantee, that the token reaches the enclave without being tampered with. It is not necessary to send in the token already activated, since an attacker with physical access can always hjiack the token's execution flow.

4.3. Security of the Key-Exchange protocol

The following discusses the security of the key-exchange protocols used throughout this work.

Key Establishment between Auditor and Verifier

Assumptions: During the provisioning phase the Verifier communicates with the token through a secure channel. An adversary must not have access to the communication, while the Verifier requests fresh keys from the Key-Oracle. An adversary with physical access to the token must not be able to infer information about the token's OTPMK. Furthermore, the token's KDF behaves like a pseudo-random function (PRF) which - given a random seed as input - cannot be distinguished from a true random value.

The collision resistance of the hash-function ensures that the adversary cannot manipulate the key which the Auditor receives. The question is, if the exchange also hides the key from the adversary such that the adversary is unable to learn significant amounts of information about the key exchanged. The following sketches a proof.

Pseudo-Random Function (PRF)

A function fulfills the PRF, if the function is indistinguishable from a Random Function (RF) which uses lazy sampling as followed:

```
var oracleCache map[[k]byte][]byte

func RF(x [k]byte) []byte {
    if oracleCache[x] = nil {
        oracleCache[x] = GetTrueRandomness()
    }
    return oracleCache[x]
}
```

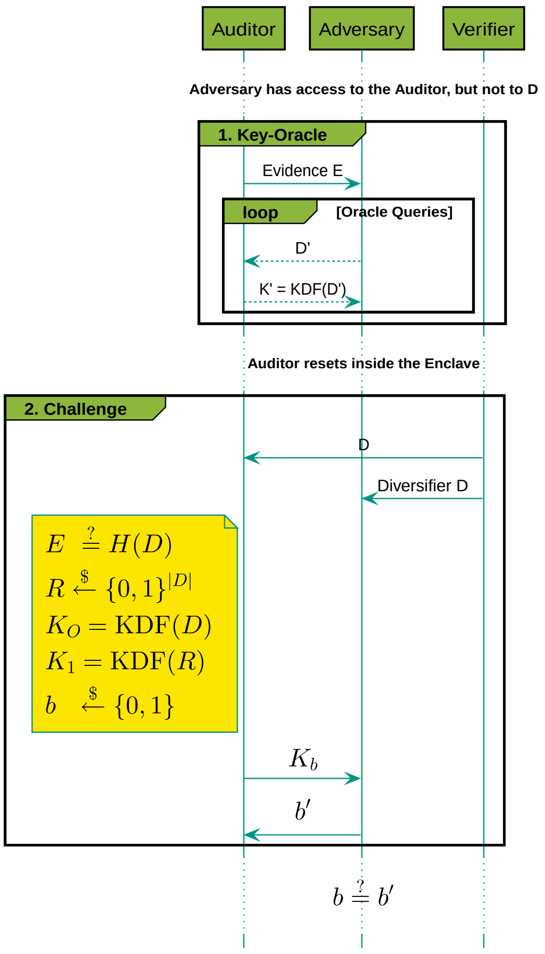


Figure 7. Security Game of the Key-Exchange Protocol

Security

The protocol is secure if the adversary cannot differentiate the created key from randomness, then the adversary does not learn any information about the key. In the context of the security game, the winning probability of any efficient adversary must be at best negligibly better than guessing.

Informal Security Proof

The following proof uses a hybrid argument, where the adversary A participates in modified versions of the aforementioned games. A played game is a protocol executed between the adversary and the Adversary. The idea of the argument is to modify the protocol piece by piece until the adversary does see random values only. The argument shows that no efficient distinguisher exists who can distinguish between the execution of the original protocol and the modified one. As a consequence, an external entity like the adversary cannot infer significant information from the key which shows the protocol's security.

Hybrid Games

The following Hybrid Games are used to show the security of the protocol:

- H_O: The previously defined security game.
- H_1^1: The first call to the key-oracle is replaced by a call to the RF (Random Function).
- H_1^n: Every oracle call made by the adversary is replaced by a call to the Random Function. With respect to k the adversary made not more than a polynominal number of calls to the key-oracle where k is KDF's output size.
- H_2: The challenger always reports a random value to the Adversary.

Indistinguishability between H_O and H_1^1

Both games are indistinguishable from each other, since a distinguisher between both games also distinguishes between the PRF and the RF.

Indistinguishability between H_1^1 and H_1^n

As already shown, replacing a single call to the oracle with a call to the RF both games are indistinguishable from each other. As shown in the literature [81] the argument extends to a fixed amount of indistinguishability games, but not if the number of Hybrids grows in k. It is left to the reader to formulate a proper proof by consulting the literature provided, but it can be constructed as follows. One has to show that the distinguisher's advantage in distinguishing the game's H_1^1 and H_1^n has a negligible upper bound. To do so, one has to first define consecutive games H_1^{i-1} and H_1^i where only a single call to the KDF is replaced by a call to the RF. The distinguisher's advantage in two consecutive games H_1^{i-1} and H_1^i can be written as a subtraction between the adversaries winning probability in game i-1 and i. The resulting telescope sum results in a negligible upper bound.

Indistinguishability between H_1^n and H_2

If A queried the Oracle on the Diversifier beforehand, both experiments are distinguishable, since the distinguisher knows which value to expect. The change for this to happen is negligibly, however. If such an adversary would have a noticable chance of querying the oracle on the Diversifier, the adversary can also find a preimage for the hash function - a contradiction to our assumption.

If the Diversifier has not been requested however, then the key is guaranteed to be always drawn at random. Therefore, both games are indistinguishable from each other.

Hiding Properties of the Key-Exchange

Since the successive Hybrid games are indistinguishable from each other, the Hybrids H_O and H_2 are also indistinguishable from each other. In the Hybrid H_2, the adversary cannot be better than guessing. Consequentialy, the adversary has only a negligible chance of winning the game H_O. Since the Auditor will not reveal the KDFs response in the real world and this argument showed that the created key is indistinguishable from randomness, an adversary does not learn a significant amount of information about the key created.

Additional considerations

The security of the protocol uses the fact, that the one-way property of the hash-function hides a sufficient amount of bits from the Diversifier such that an adversary cannot query the Key-Oracle on the Diversifier. If the diversifier is significantly longer then the hash-function's output then the hash function does not even have to fulfil the one-way property. Assume that a function f maps $3n$ bits onto an image with n bits and x is chosen at random. A statistical argument shows, that the pre-image of f(x) with overwhelming probability contains at least 2^n many elements. In addition, the hash-function is required to be only second-preimage resistance which states that it is hard to find a collision for a hash derived from a random value.

Security of the Key-Exchange Protocol between Prover and Verifier

To recap, the Key-Exchange protocol sketched in the section Section 3.6 uses a One-Pass Diffie-Hellman Key-Exchange to establish a shared secret between Prover and Verifier. In this section we assume that both participants make a black-box call to a secure One-Pass Key-Exchange protocol, which is secure if the following conditions are met:

1. An adversary can read the messages exchanged, but cannot alter them.
2. The Sender (Verifier) knows the Receiver's (Prover's) asymmetric key-share.
3. The Receiver received a fresh key share.

The protocol proposed ensures the following conditions:

1. Before the attestation starts, the Prover receives the Verifier's public signing key and hashes it into one of the PCRs. The transcript contains fresh evidence that the Prover received the Verifier's public key. The Verifier will abort the protocol, if an adversary altered the public key which the Prover received. Albeit the adversary can imitate the Verifier from the perspective of the Prover, the Prover will not receive the Verifier's secret-share. Therefore an adversary cannot compromise the public key in a meaningful way.
2. The same argument holds for the Prover's key-share, since the transcript provides fresh evidence for the share's authenticity.
3. Since the Prover knows the Verifier's public signing key, the Prover has evidence that the key-share is authentic. Since the Verifier uses the signature only once, the Prover has assurance that the key-share received is fresh.

In summary, the underlying Key-Exchange protocol fulfils the aforementioned conditions. Therefore the proposed Key-Exchange protocol is secure as well.

4.4. Security of the Architecture

4.4.1. Stakeholders

The architecture defines the following **stakeholders**. Each associated with their own security goal and threat model:

Silicon Provider

The silicon provider manufactures the hardware token. The provider wants to uphold its reputation, but may act as a covert attacker during the manufacturing process. The manufacturer can for example exfiltrate secrets like the OTPMK from the token.

Enclave Provider

It provides access to the enclave to an external Party. The provider fears a loss in reputation, since a malicious provider who gets detected may be punished by a court. Therefore, it upholds the integrity and confidentiality guarantees of the enclave. The goal of a malicious provider is to covertly exfiltrate data from the server or to tamper with the enclave's computations.

The provider fears an attack from external party who intends to inflict financial damage or deny access to the enclaves service. Therefore, the provider is interested in minimizing the risk for such an attack and may prohibit physical access to the area where the Auditor resides to trusted personal only. In addition, the provider may also wish to examine the token closely, before it is allowed to interact with the enclave.

Envelope Provider

The envelope is supposed to be tamper-evident and protects the token during its shipment from undetected tampering. The provider fears a loss in reputation, if the envelope cannot uphold its protection guarantees.

Reviewer

The reviewer inspects the enclave at the end of the setup phase and assesses if the enclave can be trusted at the time of the inspection. After the inspection, the reviewer creates a certificate which also includes the public portions of the EK which identifies the enclaves TPM. A malicious Reviewer acts covert, since it fears a loss in reputation and punishment from a court. The reviewer may implant a hardware trojan, install malicious firmware on the enclave or steal important secrets, if possible.

Verifier

The Verifier rents the enclaves computing power to perform computations on confidential data. The Verifier wants to protect the confidentiality of data which the verifier send to the enclave. In addition, the verifier is interested in a correct result or none at all. A malicious Verifier tries to implant persistent malware on the firmware level.

4.4.2. Security Goals

The architecture has the following informal security goals:

1. **Secure Remote-Attestation**: The Prover provides the Auditor with cryptographic evidence that the enclave is in an expected software state. A compromised Provider must

not be able to forge false evidence. The token and the remote Verifier require assurance over the enclave's software state.
2. The establishment of a **Secure and Authenticated Channel** between the Prover and the Verifier. An adversary who compromised the Auditor must not gain access to the shared secret between Prover and Verifier.

According to [77, 71] a secure **Remote-Attestation** protocol must guarantee the following security goals:

Comprehensive	The attestation protocol provides the Auditor with comprehensive information, which allows the Auditor to reason about the entire state of the Prover. In the context of TPMs, the information must include information about the Prover's entire execution environment. This includes the platforms firmware and the entire software state of the booted OS.
Fresh	The Prover's response to an attestation request can reliably be linked to the request and the protocol reflects the state of the Provers current state. If the Auditor accepts old information, an adversary can falsely convince the Auditor by replaying a past protocol execution.
Atomic execution	The attestation procedure must be completed in its entirety or not be completed at all. A malicious Prover cannot interrupt the attestation by any action invoked on the device, without stopping the attestation in its entirety.
Trustworthy	The Auditor is able to reason that the information received from the Prover is correct. Even in the presence of an active adversary, the attestation mechanism remains trustworthy.
Exclusive access	Only the Provers attestation mechanism - the TPM - has read access to the attestation secret. The secret provides authentic proof to the Auditor, that the information received originates from the attestation mechanism.
Immutability	An adversary - with local or remote access - cannot modify the attestation mechanism.
No leaks	The attestation mechanism - provided by the TPM - must not leak any information that allows an adversary to reason about the attestation secret before the attestation took place. An adversary with access to the secret is able to imitate an honest Prover.
Invocation	The attestation mechanism must only be invoked from its intended entry point.

4.4.3. Trust Model and Scope

The chapter collects implicit trust assumptions made throughout this work and associates these assumptions with individual components or participants.

Enclave's Trust Assumptions

- A remote entity like the Verifier can learn the EK associated with the enclave's TPM.
- A corrupted prover can not cheat on the remote-attestation and will be detected by an uncompromised Verifier or Auditor.
- The enclave shields its private keys from a compromised Prover. This includes EK's and AK's.
- An adversary cannot roll-back the enclave's firmware to a vulnerable firmware or will be detected in the attempt of doing so.
- The macro-enclave has means to uphold the confidentiality of data even if an adversary launches an intrusive attack mid-execution. The enclave can protect the confidentiality of data residing on the server against physical intrusion. It can do so by erasing a memory encryption key.
- The enclave is tamper-evident and can provide proof to an external entity that no intrusion occurred since its setup. The enclave may hold an erasable secret in volatile memory, which it can use to authenticate itself to an external entity. Once tampering is detected, it destroys the secret.
- An adversary must not be able to steal the authentication secret from the enclave.
- The platforms core-root-of-trust assumptions must hold. The enclave must protect the integrity of the BIOS initial boot code, as well as the CPU's and TPM's firmware. In addition, the communication channel between the aforemention components must be protected against tampering.
- The macro-enclave discloses inherent side-channels of the server and the monitoring devices must not open up additional side-channels which would allow an external adversary to infer information from an honest Prover. This is not trivial, since even a variation in the brightness of power-LED's reveals some information. The Provider should not have means to exfiltrate data by monitoring side-channels like power-consumption or voltage-fluctuations.
- The data-diode must prohibit bidirectional traffic. This is a challenge, since it also makes the usage of a TCP connection between two entities impossible. A clever solution may use some form of buffering to account for package loss. Care must be taken however, as the data diode must be free of side-channels.

Prover related trust assumptions

- A trustworthy Prover must provide evidence for all information which it delegated from an external entity to the Auditor.
- The Auditor and the Prover should communicate over a confidential channel.

Envelope's Trust Assumptions

- The envelope is tamper-evident and can be opened at most once. An adversary with access to the sealed envelope cannot open the envelope without leaving evidence behind.
- The envelope must prohibit visual probing of the shipped token. If the token is hard to

forge due to some visual patterns, the envelope must prevent the creation of an inconspicuously looking forgery.

Token’s Trust Assumptions

- The token protects the confidentiality and integrity of the burnt-in master secret called OTPMK.
- The cryptographic operations implemented in hardware are free of side-channels. An adversary must not learn underlying secrets - like the OTMK, by monitorng the token while it performes cryptographic operation. For the OTPMK this must be true even in the context of physical tampering.
- The token has been properly configured during the setup phase. The HAB is activated and an adversary does not gain access to the HAB’s signing keys.
- The token’s HAB does not boot from an unsigned image. An adversary can execute malicious code on the token only, after a signed image booted on the device.
- A physicaly uncompromised token running malicious code reverts to a trusted state once it lost power. In other words, a compromised Auditor can be reset.
- The token is free of hardware trojans.

Auditor’s Trust Assumption

- The Auditor is provisioned with unforgeable evidence that the Verifier’s data - like the Diversifier - originated from the Verifier.
- The evidence is hiding. An adversary with knowledge of the evidence must not be able to infer information about the Verifier provided data.

This work makes more trust assumptions than necessary, since out-of-scope attacks are quite common in practice. In an architecture which makes precisely the assumptions necessary, a single out-of-scope attack can result in catastrophic consequences for the architecture’s security. For example an adversary who corrupted the Prover and the Auditor, should not be able to falsely convince the Verifier of the enclave’s trustworthiness. Consequently, this work overcompensates in order to obtain robust security guarantees.

4.4.4. Verifier specific Security Considerations

The chapter analyses the security guarantees which the architecture provides under idealized parties and discusses which security guarantees can be provided by the architecture if some security assumptions are violated.

Secure remote attestation

The Verifier receives the Auditor's transcript over the attestation it allegedly performed. In the context of a trusted Auditor, the Verifier receives exactly the same information as the Auditor would. A compromised Auditor or Prover cannot alter the Audit's content, assuming the TPM protects the confidentiality of the EK and AK used in the attestation. Therefore the transcript retains that the information received is **Comprehensive**. Since the Auditor acts for the most parts just as a mediator - one can think of the token as a MITM attacker - most properties are transitive. A compromised Auditor can however deviate from the protocol without being noticed and reduce the entropy of the nonce used during the attestation. From the View of the Verifier, the attestation is Atomic in the sense that the Verifier cannot interrupt the TPM's audit. Albeit some bytes of freshness are contributed by the Auditor, each party contributes in total 24 bytes of nonce - which is plenty enough. Therefore, the attestation is fresh.

Atomic: The attested Audit is atomic. If the Auditor is trustworthy, the Auditor creates either the entire transcript or none at all, and since the entire transcript is signed it can also not have been altered. A compromised Auditor can however interrupt the Prover in between requests. As long as the Auditor cannot compromise the AK however, this is not a security concern.

Fresh Information: Both the Auditor and the Verifier contribute some bytes of freshness. Since a compromised Auditor must include the Verifier provided nonce in all requests send to the Prover, the resulting transcript provides evidence, that the transcript where recently created. Care must be taken if the Activate Credential command is used to attest the AK. Without extra measurements the command does not proof that the attested AK is fresh.

Goal	Assessment
No leakage to a compromised Prover.	Neither the token nor the client will send its key-share to the compromised server. Albeit the Prover learns the public diversifier which the token uses to derive the key, the Verifier uses a secret OTPMK burnt into the token's hardware as an additional input.
Physical attack during the transportation of the Auditor which attempts to extract secrets.	The Auditor does not have knowledge of the private key. Without physically destroying the token an adversary cannot extract the tokens OTMP. The adversary must therefore inject malicious code onto the token during its runtime.
An adversary which compromises the Auditor, but with no access to the Verifier's key-share.	After the adversary compromised the Auditor, the adversary learns the Auditors key-share. Without access to the Verifier's key-share, the adversary does not learn anything.
A compromised Auditor and a compromised Prover.	The Verifier has access to the Auditors Transcript, which includes the Verifier provided freshness. Therefore, the Verifier will not trust the Prover, assuming that the adversary cannot launch a dynamic attack against an uncompromised Prover after the audit took place.
An adversary which compromised the Auditor after its deployment and which has access to the communication between Prover and Verifier.	The adversary learns both key-shares, since the Verifier sends its key-share over an unencrypted channel. The current architecture does not fend off this type of attack. To do so, two possible mitigations are possible. Either the verifier prevents the adversary from learning the Auditors secret or the Auditor relays a public encryption key from the Prover to the Verifier - similar to the AK which is included in the transcript. The Verifier must have means to reset the Auditor before providing the Auditor with the diversifier - otherwise a compromised token can leak the Auditor's secret to an adversary. On the Mk II cutting the power is enough to reset the token's software state

Table 3. Secure Key Exchange

Security considerations:

The current architecture does not provide forward security for the keys exchanged. If an adversary learns the key share of the Verifier and gets physical access to the token later on, they can exploit the token to reveal its secret. At the end of the attestation, the Auditor can revoke its own signing key to prevent an adversary from accessing the tokens secrets. Future work may implement a key exchange which provides forward secrecy.

4.4.5. Threats and Mitigations

By default, the Threat Model assumes that the adversary performs an MITM attack between the enclave and the token, and that the adversary has access to the video feedback of the cameras monitoring the enclave and token. The adversary is not able to swap out the token during transportation. However, the adversary may swap out the token after the token arrived at the enclave.

Threat	**Mitigation**
An adversary who provides the enclave with a compromised VM.	A faithful hypervisor - who starts the DRTM - must measure the entire image which loads the VM. A faithful VM must measure every piece of firmware and software which the VM will execute over the VM's entire lifetime. In addition, the measurements must also encompass all associated configuration files.
An adversary who corrupts the enclaves hypervisor and the Prover.	The DRTM starts from initially trusted piece of firmware which acts as the root of trust. The platforms BIOS and CPU protect this piece of firmware from corruption.
An adversary who attempts to clone the token's KDF after it learns the Auditor's diversifier at the start of the attestation phase.	To protect against cloning the token's key-derivation function uses the token's fused-in OTPMK as a secret input. Therefore an adversary can not use the publicly known diversifier alone to infer information about the Auditors secret keys.
An MITM attack which alters the diversifier the Auditor receives during the attestation phase.	The Auditor is provisioned with evidence - a cryptographic hash - which allows the Auditor to detect tampering. The attack will fail and the Auditor will not send the Verifier a signed transcript.
An MITM attack between Verifier and Auditor to falsely convince the Verifier that the Prover attested for the expected state.	The Auditor signs the created transcript with an EUF-CMA secure signature or MAC scheme. The transcript contains verifier provided nonce and protect against replay attacks.
A compromised Auditor which sends a forged transcript to the Verifier.	The TPM created data is signed by the TPM and prevents the Verifier to forge entries. The verifier in addition checks that it contains the Verifier provided nonce to ensure its freshness. Thus the malicious auditor cannot lie or replay a prerecorded transcript.

Threat	Mitigation
An adversary who attempts to steal the Auditors key-share during its transportation.	The sealed envelope should protect against unauthorized access. However, even if the adversary can exploit the token to query the token's KDF, the adversary does not have access to the Verifier's Diversifier. Therefore, an adversary cannot steal the Auditor's key-share or authentication key during its transportation. To forge the token's KDF responses, the adversary must extract the token's OTPMK. The silicon provider must ensure that the token's SoC protects the OTPMK, including side-channel attacks, fault-injections or invasive attacks which destroy the chip. In addition, the Verifier's implementation must protect against a hardware trojan implanted during its transportation.
An adversary who stole the Auditors key-share.	The Verifier must establish an authenticated channel to the Prover using asymmetric cryptography, as explained in the section Section 3.6.
A malicious Auditor who launches an attack against the Prover or the Verifier.	To prevent the corruption of an honest Prover or Verifier, the communication interface between all parties is kept minimalistic and written completely in the memory-safe language Go.
An MITM attacker or a compromised Prover tamper's with the Verifier provided data - which contains the Diversifier and some nonce.	A trustworthy Auditor will not accept tampered data, since the Auditor is provisioned with evidence about the Verifier's data.
A malicious Auditor who conspires with a compromised Prover to leak data to an outsider.	The adversary learns the Auditor's key-share and gains access to all encrypted data which resides inside the enclave. However, the attack is not covert, since a compromised Prover won't be able to create a verifiable transcript. If the Verifier wants to protect proprietary code, the Verifier sends the encrypted code to the enclave, which will be loaded and executed after the Prover received the decryption key. Care must be taken however, since the encrypted code must also be attested for.
A malicious provider sends a compromised EK to the Verifier.	The Reviewer created some evidence, which links the enclave to its associate EK.
A malicious Reviewer who creates a false certificate.	A malicious Reviewer can alter the certificate. To protect against this type of attack, both the Verifier and the Provider must exchange the certificate over an authenticated channe. Assuming that the Reviewer and Provider mutually distrust each other, the Reviewer cannot provide the Verifier with an altered certificate.

Threat	Mitigation
The TPMs firmware is vulnerable to a publicly known side-channel attack which compromises the TPM's EK in the long term.	Once the enclaves EK is leaked, the entire enclave cannot be trusted. To protect the verifier's interests, the Reviewer should offer a revocation mechanism.
A rollback attack against the TPM's firmware to an old and insecure firmware version.	Wi TPMs which implement the Specification with Version 184 or higher support Firmware limited keys. The verifier can request such an AK to prevent rollback attacks and the provisioner can provision the TPM with an EK fixed to a specific firmware version. The latter mitigates rollback attacks against EKs, but also prevents upgrades of the TPM.
An adversary who tampers with the auditor's platform during its transportation to the Prover.	The token cannot protect itself against code injection against an adversary who gains unsupervised access to the token. By cutting power however, the Auditor resets into a trusted state. In addition, a tamper-evident envelope - which can be opened at most once - should protect the token from undetected tampering. During the envelope's opening, the opening must create fresh proof which the Verifier can observe in the cameras recording. The adversary should not be able to steal important secrets from the Auditor, since the Auditor does not have access to its secret during transportation either.
A provider who replaces the token with a forgery after the provider opened the enveloped, but before the attestation took place.	The provider can play being suspicious about the token and first pretend to inspect the token. During the inspection the provider can swap out the token with a forgery and compromise the Auditor running on the token, before the provider puts the token back into the enclave. To protect the Auditor against such an attack, the verifier can request a reset of the Auditor just before the attestation takes place. In addition, a unique pattern printed on the token prevents the adversary from replacing the token with a believable forgery unnoticed, since the Verifier monitors the provider at all times over the camera feedback. Simple nail-glitter patterns placed onto the token should be enough in practice. To deny the adversary time to prepare the creation of such a forgery, the adversary must not possess knowledge of the pattern in beforehand.

Table 4. Threats and Mitigations

Threat	Challenge
A compromised Prover which evades detection from the Auditor and Verifier.	After the Prover allegedly finished its computations on the Verifier provided data, the Verifier requests access to the data by inverting the direction of the data diode. The compromised Prover can use this to leak arbitrary amounts of data. A sophisticated adversary may be able to act covert by for example reporting encrypted data while side-channeling the decryption key to the adversary.
The adversary gains access to both the Auditors and the Verifier's key-share.	Such an adversary can decrypt all messages which the Verifier and Prover exchanged. In addition, the adversary can also gain access to the final result.
Dynamic Adversarial Model	A dynamic adversary can decide at which point of time it corrupts the server. This is for example the case, if the Prover can be exploited over the network interface exposed to a corrupted Auditor or if the software running on the enclave does not properly parse Verifier provided data. Consequently, the adversary can corrupt the Prover after the DRT measurements took place and therefore evade detection completely. An outsider who sends tampered data to the enclave in the name of a client should not be able to exploit most vulnerabilities, since the Data Diode discloses any read primitives which many exploits require. The interface between Auditor and Prover must not provide an exploitable attack surface, however.

Table 5. Threats which are out-of-scope

This section identifies security considerations that must be evaluated by the stakeholders who implement the attestation framework.

Security Consideration	**Stakeholder**
The enclaves set of requirements must be met which provide the foundation for the confidential enclave computing.	Silicon/Enclave Provider, Reviewer
Once the provider finished the enclave’s setup phase, the enclave must provide fresh proof to the Verifier, that the enclave’s environment was continuously monitored and that no physical tampering occurred.	Silicon/Enclave Provider
The security of the enclaves firmware components must continuously be monitored for critical vulnerabilities. In addition, a revocation mechanism must be in place which can inform the Verifier upon the discovery of a firmware exploit.	Reviewer, Silicon Provider
The security of the attesting software running inside the enclave must be evaluated. Most and foremost the application must not expose an exploitable attack surface to the outside, including the network interface communicating with the token. If the application is exploitable, a dynamic adversary is able to evade detection.	Verifier, Software Provider
The Prover must provide proof for the enclave’s complete software state.	Verifier, Software provider
The token must protect volatile secrets from any physical tampering until next reset occurs.	Verifier, Silicon/Software provider
The Verifier must properly verify the token’s transcript and must should provide a fault-tolerant mechanism to ensure that the token receives the diversifier and nonce.	Verifier, Software Provider
The token must be shipped in a tamper-evident fashion.	Verifier
The token must be visually hard to forge. The verifier should print a unique identifier in the tokens circuit board by using nail polish for instance.	Verifier

Table 6. Security considerations for Stakeholders

Chapter 5. Implementation

This chapter describes certain aspects of the implementation [89] which puts the architecture into practice. It discusses some design decisions and solutions to the encountered issues. The focus of the implementation lies on providing of a proof-of-concept solution which can be easily adapted in future work. The solution is however not complete, as it is discussed in the Section 5.5.

5.1. Goals and Contributions

The main purpose of this work lies on defining a minimalistic interface between Auditor and enclave, as well as implementing the interface on the token and server-side. Meanwhile, the main focus of this work concerns the Auditor applet. The following section describes the design ratios which are used throughout the implementation.

All three parties, the Prover, the Auditor, and Verifier mutually distrust each other's software state. The Prover can be compromised from the beginning, the Auditor could be compromised during its transportation or be replaced by a forgery. Since the token should not be trusted to fend off all types of attacks, one of the goals is to provide plausible security guarantees even if an adversary compromised the Auditor.

Hardware based attacks against the tokenr are not the only concern however, since each party may be subject to a software exploit. Any data exchanged with the Verifier can be read by an adversary or may have been tampered with. To minimize the attack surface, all parties exchange as little data as possible. A malicious Prover may attempt compromise the Auditor or vice versa. Fewer data exchanged also means that fewer data has to be verified. The Auditor for example should therefore not be allowed to interact with the Prover's TPM directly, but the Prover must expose only the smallest possible interface possible. Ideally, almost the entire request of the Verifier can be reconstructed by the Prover, while the Verifier should only accept and parse the data which it cannot derive on its own. The implementation makes heavy usage of templates, which can reconstruct most TPM data from few bytes and some enumerations. Each API request and response contains only few pieces of information - like the type of the request, Auditor provided nonce, or large integer values which represent a public key or signature.

The network interaction between all parties are written entirely in memory safe code and the data is serialized using CBOR-encoding.

To minimize the Auditor's code dependencies, the Go application executed on the token is not launched by an encumbered OS typically written in millions of lines of memory-unsafe code. Instead a small hypervisor written in the TamaGo framework executes the Go applications on the token - which this chapter refers to as Auditor-application. The entire implementation - including all dependencies - has less than 2000 lines of memory unsafe code dependencies. The clear separation between hypervisor and Auditor-application also enhances portability, as the hypervisor has a small code-basis, while the Auditor meant to be a standalone solution unaware of the underlying OS.

5.2. Network Communication

In spirit of a stateless RESTful API, the Auditor and Prover exchange information in challenge-response style. The chapter explains the implementations design decisions at exemplary code snippets. To minimize the attacker surface, the implementation of the communication interface makes the following design decisions:

- To simplify the protocol the Prover is written as a stateless program. The sole exception lies in the established attestation key.
- Request or Responses over the api are modelled as data structures which contain enumeration's, arrays, or the `big.Int` type from the go's standard library.
- Both parties marshal data using the CBOR codec [75]. This work uses the encoder from the `fxamacker/cbor` library [54]. The library is purely written in Go, does not use unsafe primitive, claims to be hardened against adversarial inputs, and underwent a security assessment from Microsoft.
- To improve the implementation's readability and to ease the maintainability, the implementation uses generic key and signature interfaces. The CBOR encodes handles the parsing of said data structures, by mapping a concrete type to a unique identifier. To achieve compatibility across languages and ensure forward compatibility, the mappings are set explicitly.
- In the implementation, the Prover establishes the network connection to the Auditor , before the Auditor can issues a request. This approach allows the Prover to sit behind a firewall which drops all incoming packages except for packages associated with a connection which the Prover established.

5.2.1. Serialization

Each API request or response will first be marshaled into a byte-representation which forms the so-called payload. The payload will then be encoded via TLV encoding as follows: be sent as byte-stream message over the network using TLV encoding The payload is encode CBOR encoding forming the payload of length of the message is represented by a two byte integer in the BigEndian format, the type is an byte-encoded enumeration and the payload is a CBOR encoded byte-array.

```go
// Code from src/nethelper/nethelper.go
struct Message type { ❹
  len      [ 2 ]byte
  envelope [len]byte
}
// Code from src/api/envelope.go
type Envelope struct {
    Type    payload.MsgType `cbor:"type"`    ❶
    Payload cbor.RawMessage `cbor:"payload"` ❷
    Digest  []byte          `cbor:"digest"`  ❸
}
```

Exchanged Messages

❶ The type of the payload is an enumeration. Based on the type of the message, the parser will attempt to parse the receives message.

❷ The CBOR encoded payload contains an api request or response which has been CBOR-encoded.

❸ The digest is computed from the SHA256 hash function over the type and message content, which allows the Verifer to detect the corruption of messages due to network errors.

❹ The Envelope is marshaled into its binary representation and send as length encoded message over the network.

5.2.2. Remote Attestation

The section explains how a remote entity like the Auditor can interact with the Prover the remote attestation. Each step of the Remote-Attestation, described at Section 3.3, corresponds to a request of the Auditor and the response of the Prover. The implementation minimizes the data which has to be exchanged between both entities. The following excerpt from the API demonstrates its design.

The Auditor requests a fresh AK under the Key-Parameter's which the Auditor has been provisioned with. The `KeyScheme` datatype specifies all parameters related to a TPM key as defined in the appendix at Key-Scheme related data structure. The Scheme can be seen as a key-template, where each entry of the scheme is represented by an enumeration or bit. The template is secure by default, since only approved values can be set. The KeyScheme's `TPMParameters()` creates a byte representation of the key's parameters, which are also contained in the key's public data. Knowing these values is essential for computing a key's name.

```
type AkRequest struct {
    Nonce     []byte                `cbor:"Nonce"`
    KeyScheme cryptoutil.KeyScheme `cbor:"KeyScheme"`
}

type KeyScheme struct {
    // Hash algorithm used to compute digests.
    HashAlg            tpm2.TPMIAlgHash        `cbor:"HashAlg"`
    KeyParameter       KeyParameter            `cbor:"KeyParameter"`
    TPMMetadata        datatype.TPMKeyMetadata `cbor:"TPMKeyMetadata"`
    SymmetricParameter SymmetricKeyParameter   `cbor:"SymmetricKeyParameter"`
}

func (scheme KeyScheme) TPMParameters() (parameters []byte)
```

The Prover's response contains the freshly created key - in this example and ECDSA key - and some TPM specific and unverifiable Metadata represented as byte-arrays. The provided key must implement the Key-Interace Public Key Interface like the `ECDSAKey`. From there on the attester refers to the AK by name by using a wrapper function called `PublicKeyContext` as shown in the appendix Public Key Interface.

```
type AkResponse struct {
    Ak cryptoutil.PublicKey `cbor:"Ak"`
}

type ECDSAKey struct {
    X       *big.Int     `json:"X" cbor:"X"`
    Y       *big.Int     `json:"Y" cbor:"Y"`
    KeyType KeyParameter `json:"KeyParameter" cbor:"KeyParameter"`
}
```

The Auditor can establish trust in the AK by requesting for instance a fresh certificate from the Prover's TPM that the key is authentic. The Prover responds to the request, which contains the Key's name and some fresh nonce, with a so-called `CertifyAKResponse`. The response contains some unverifiable metadata and a signature which implements the `Signature` interface - like an ECDSA signature. The usage of interfaces - rather than concrete types - makes the implementation future-proof and forward compatible, since the API does not have to be changed.

```go
type CertifyAKResponse struct {
    // The signature of the certificate.
    Signature cryptoutil.Signature `cbor:"Signature"`
    // The metadata associated with the certificate which the verifier has to trust.
    Metadata datatype.TPMAttestationInfo `cbor:"TPMAttestationInfo"`
}

type ECDSASig struct {
    R *big.Int `cbor:"r"`
    S *big.Int `cbor:"s"`
}

type TPMAttestationInfo struct {
    QualifiedSigner []byte `cbor:"QualifiedSigner"`
    // ClockInfo: The values of Clock, resetCount, restartCount and Safe.
    ClockInfo [17]byte `cbor:"ClockInfo"`
    // FirmwareVersion which is relevant to security.
    FirmwareVersion [8]byte `cbor:"FirmwareVersion"`
}
```

The Auditor verifies the signature received by recomputing the TPM's entire certificate from the received Public Key and the TPM's reported metadata. The certificate's head can be computed from the TPM reported Metadata. For the attestation specific data the Auditor recomputes the TPM's `creationInfo` which contains the AK's name and its associated creation hash.

Finally, the Auditor requests a freshly signed certificate from the Prover similarly to the attestation of the TPM's key.

5.2.3. Transcript of the Attestation

The Auditor creates a verifiable transcript over the remote-attestation performed. The section discusses how the Transcript is designed and how an external party can verify the transcript.

Each of the transcript's entries is associated to a single interaction between Auditor and Prover. In the first round the Prover creates a fresh AK, in the second it attests for the AK, and finally attests for the PCR values. To ensure freshness, the Verifier must have sent some nonce to the Auditor in beforehand. The nonce in the transcript must therefore be composed of both Verifier and Auditor provided nonce.

```go
type Entry[T payload.Message] struct {
    Nonce    []byte `cbor:"Nonce"`
    Response T      `cbor:"Response"`
}

type Transcript struct {
    AkRequest  Entry[payload.AkResponse]         `cbor:"AkRequest"`
    CertifyAk  Entry[payload.CertifyAKResponse]  `cbor:"CertifyAk"`
    Credential Entry[payload.CredentialResponse] `cbor:"Credential"`
    Audit      Entry[payload.AuditResponse]      `cbor:"Audit"`
}
```

Transcript

The Verifier verifies the transcript as follows:

1. Verify that each entry contains the verifier provided nonce.
2. For each entry, recreate the Auditor's request based on the Nonce reported, and the TPM data structure's from the Prover's response.
3. Verify the response which the Prover allegedly created.

In addition, the Auditor encrypts and signs the transcript before sending it. A wrapper function implements the functionality.

5.3. Mk II Application

The following section discusses how an application can be executed on the USB Armory Mk II. A generic ARM device like the token cannot execute a compiled Go application directly, since a compiled program requires the OS to implement a set of syscalls and drivers to communicate with the hardware.

A precompiled linux OS for the MkII exists, but it would introduce a large amount of memory unsafe code into the project. The application is instead written in the TamaGo framework to remove the necessity for an underlying OS. The framework does however require some modifications on the applications, since the application must interact with the platform directly. The applications code must for example specify the memory layout, initialize driver, and handle interrupts appropriately. To retain maintainability and portability, the application is split intro three components. A second-stage bootloader which initializes the RAM encryption, a hypervisor which initializes the platform and who handles interrupts, and the actual Go program which can be compiled to an arbitrary platform. The following describes the different components bundled with an image which boots the application.

Component	Purpose
Boot-loader	The bootloader is a second stage bootloader, which loads the rest of the application into an encrypted RAM section, before booting from it.
Hypervisor	Before the main program executes its code, the hypervisor initializes and configures drivers required by the program. During the execution of the main program, the hypervisor handles interrupts cased by for example the network driver.
Program	The Program is supposedly agnostic of the underlying platform and has at most minor dependencies to the TamaGo framework. The Auditor for example interacts with the Prover, performes the remote attestation, and once finished sends the Verifier a transcript.

Table 7. Components of an Application-Image

Auditor-Application

The Auditor interacts with the Prover and carries out the remote attestation. Once finished, the Auditor sends a verifiable transcript to the Verifier. Due to the unreliability of the token, the Auditor is designed to be stateless. The token does not require access to a power-source during its transportation and is supposed to boot inside a protected environment. The implementation is robust and if the token does not boot properly due to occurring power-glitches while being plugged in, the Provider can simply reset the token until the Auditor booted properly. The approach also enhances security, since a compromised Auditor can be reset into a trusted state by cutting the Auditor's power-supply. If the adversary is not physically present, the Auditor cannot be exploited. In other words, once the Auditor reset inside a secure environment, the Auditor is trustworthy. After the attestation phase took place, the Auditor creates a signed transcript for the Verifier and sends its own key-share to the Prover.

+ Since the Auditor is not in possession of secrets, the Auditor must create its secrets on an on-demand basis. The Verifier establishes a set of secrets shared with the Auditor, by sending a diversifier over an untrusted channel to the Auditor. The diversifier is the same as the one created during the setup phase. If the Auditor receives an untampered diversifier, both the Auditor and the Verifier share the same set of keys. If an adversary tampers with the diversifier, the Verifier receives a faulty signature and aborts.

Loading Configuration Files

The token loads its configurations from a JSON files. To protect the configuration files against tampering the configuration files are embeded into the Auditor's image, since the token's secure boot feature will refuse to boot from a modified image.

The module `src/configReader` parses these configuration files which closely resemble the underlying data structure which they will parsed to. Entries which will be parsed to enumerations are designed to be human readable as the following two examples show:

```
{
    "X" : "bd14fbb52d5fe8b5eec4848c7909f138ea5cc5d743ff1dec551b33a13beb4e7a",
    "Y" : "26fecab7d08f44524f17b0f5aacfa2748b83f6fa1dc69d5b2134f7337af79937"
}
```

EK JSON

```
{
    "HashAlg": "sha256",
    "KeyParameter": "ecdsaP256",
    "SymmetricParameter": "aesCFB128",
    "TPMMetadata": {
        "STClear": false,
        "UserWithAuth": false,
        "AdminWithPolicy": true,
        "NoDA": false,
        "Decrypt": true,
        "AuthPolicy": "g3GXZ0SEs/gakMyNRqXXJP1S124GUgtk8qHaGzMUaao="
    }
}
```

EK Scheme

5.4. Proof-of-Concept Verifier

As it stands, the Prover implements the Verifier to provide a proof-of-concept solution, showing that a remote entity can in principle perform the remote-verification.

The Verifier's transcript is wrapped in a helper function, where each entry represents a single interaction between the Auditor providing some nonce and the response of the Prover. Crucially, the Verifier can reconstruct the Auditor's entire request with exception to the nonce. In the current implementation, both the Auditor and the Verifier contribute some nonce, so that each party individually has assurances over the attests freshness. Using the KDF as a PRF, exchanging nonce between both parties can be omitted. This is however left for future work.

The entire transcript is verifiable, with exception to the MakeCredential command. Since the Auditor creates the encrypted challenge, and the TPM does not create some form of signed Proof, the Auditor can forge the TPM's response. The solution to this is, that the Verifier must create a credential blob, which it sends to the enclave. Without access to the AK in beforehand, an additional round of interaction between Prover, Auditor, and Verifier is required. Future work which wishes to use the command must assess the security implications.

5.5. Contributions and Shortcomings

The section briefly highlights the contributions and shortcomings of the implementation provided. The implementation provided is not complete, but is designed to be easily adaptable for future work. To be more specific, the following components are missing:

Shortcomings

- The TPM specification is vast, and this work does not claim to be complete regarding every possible configuration. For a secure implementation thorough tests are required.
- The Key-Oracle image is not implemented and the current solution emulates the token's KDF by using a test-vector. For a proper implementation, the hypervisor can be reused. In fact with very few modifications of the Auditor, it can be provisioned twice. Based on a boolean attribute the token could either respond to key-requests only or perform the attestation.
- The currently established key is created from a test-vector using a software KDF used in implementation. The Auditor's code implements a currently unused switch which uses the token's KDF once deployed, and very little changes are necessary to activate a secure implementation.
- The current implementation of the Auditor does not make use of asymmetric cryptography. In the context of multiple verifier's this may be undesirable.
- The implementation does not provide any mechanism for a secure Key-Exchange. Implementing the secret-sharing based Key-Exchange is trivial and this work already used an ECDH based key-exchange protocol used in the MakeCredential command. Since this type of key-exchange is not post-quantum secure, the Verifier and Prover must find and implement a suitable Key-Exchange protocol instead. This however is out-of-scope for this work.
- The Provisioning of the token has not been implemented yet, but a provisioning guide is provided. Care must be taken, as a false configuration can render the token inoperable.
- The Auditor has still few code dependencies to the TamaGo framework which have to be cleaned up.

Contributions

1. The implementation of an Auditor-application written purely written in memory-safe Go.
2. A hypervisor which provides the Auditor-application an executing platform which is written purely written in Go. The hypervisor is runs bare-metal on the token created in the TamaGo framework. The framework removes almost all dependencies to memory unsafe code by introducing around 6000 lines of code written almost entirely in Go.
3. The implementation minimizes the amount of data exchanged between Prover, Verifier, and Auditor to reduce the attack surface to a bare minimum.
4. A Prover written entirely in Go.
5. A high-level interface between the TPM and Prover written entirely in Go to aid in the attestation.

6. A high-level network interface written in Go which can (de-)serialize requests and responses between the communication parties.
7. The implementation of a proof of concept Verifier which can Verify an Auditor provided transcript.

Chapter 6. Summary and Future Work

This work demonstrated that TPM based attestation is feasible in the context of a remote-server which cannot send data due to a data diode in place. The work proposed a generic architecture where a hardware token - so-called Auditor - interacts with the server in the name of a remote Verifier. In the architecture, the Auditor acts as a hardware-firewall between Prover and Verifier. Once the attestation succeeded, the Auditor sends a verifiable transcript of the interaction to the Verifier over another unidirectional data diode.

The Auditor is an unencumbered Go application implemented using the TamaGo framework. The framework patches the standard Go library, so that the application can be executed bare-metal on the token's hardware. Including the frameworks patches and drivers, the entire implementation depends on less than 2000 lines of memory unsafe code.

The Auditor's target platform is the USB Armory Mk II, a secure hardware token supported by the TamaGo framework. To understand the security guarantees which the implementation can provide, the work evaluates the security guarantees which the USB Armory Mk II offers. The token is not tamper evident against a physically present adversary, as one can inject arbitrary code into the token's external RAM. Interestingly though, the adversary cannot make itself persistent across boot cycles and the token's CAAM can plausibly protect secrets even if the executed code is corrupted. The analysis concludes, that the token can be trusted only after it is reset inside a protected environment.

The architecture and the implementation take these shortcomings into account and consequently minimize the trust assumptions into the token. In an Attestation scenario, a compromised Auditor can cause significant harm only if the Auditor can exploit the Prover after the attestation took place. In the context of data diodes, a compromised Auditor can furthermore act as a detection evading side-channel for a corrupted Prover.

The work furthermore proposes different symmetric and asymmetric Key-Exchange protocols between Verifier and Prover in the context of data-diodes. For a symmetric key-exchange, the Verifier must trust that the Auditor's hardware is tamper-evident and that it can protect entrusted secrets. The implementation uses a secret sharing based approach. The approach is secure if an adversary does not gain access to the Auditor's communication channel, cannot extract the secrets after the Auditor destroyed itself, and if the Auditor reseted itself into a trusted state. Furthermore the work sketches a generic key-exchange protocol between Verifier and Prover, which does not make trusts assumptions for the Auditor token. Future work should combine the key-exchange protocol with the much simpler key-share based approach to protect the confidentiality of the established key against an adversary who is much more capable.

The architecture makes efforts to reduce the trust assumptions necessary during the setup and provisioning phase, and should be extendable to a multi-client scenario. However, more research is required to sufficiently design a secure setup and provisioning phase for the token in a multi-client scenario. Future work should also reduce the size of the communication channel between Auditor and Prover. With respect to the number of Verifiers, the Auditor has to send a linear amount of signatures to the Verifiers. But not only may the communication channel be severely limited in size, but an adversary may also find a way to hide large amounts of data inside a large communication channel.

Future work should make efforts to build a new token which succeeds the Mk II. The token should be able to retain and protect erasable secrets when no external power supply is available. The token must furthermore provide proper authenticated code execution or be at least tamper-evident. The focus should lie on finding secure and - if possible - open source hardware, building a platform suitable for the use-case, and to provide the TamaGo framework with appropriate drivers. The Auditor is designed to be a standalone solution and can ported onto another platform by replacing the hypervisor.

Bibliography

[1] “Smart card application protocol data unit.” Accessed: Oct. 09, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Smart_card_application_protocol_data_unit.

[2] “Bus (computing).” Accessed: Oct. 13, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Bus_(computing).

[3] “Challenge-response authentication.” Accessed: Oct. 09, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Challenge%E2%80%93response_authentication.

[4] “Unidirectional network.” Accessed: Aug. 09, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Unidirectional_network.

[5] “Endianness.” Accessed: Oct. 15, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Endianness.

[6] “Physical unclonable function.” Accessed: Oct. 09, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Physical_unclonable_function.

[7] “System Management Mode.” Accessed: Oct. 11, 2025. [Online]. Available: https://en.wikipedia.org/wiki/System_Management_Mode.

[8] “System on a chip.” Accessed: Oct. 10, 2025. [Online]. Available: https://en.wikipedia.org/wiki/System_on_a_chip.

[9] “Trusted Platform Module.” Accessed: Oct. 08, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Trusted_Platform_Module.

[10] “Trusted execution environment.” Accessed: Aug. 09, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Trusted_execution_environment.

[11] “NIST Special Publication 800-77 - Revision 1 - Guide to IPsec VPNs.” Accessed: Oct. 09, 2025. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-77r1.pdf.

[12] “NIST Special Publication 800-107 - Revision 1 - Recommendation for Applications Using Approved Hash Algorithms.” Accessed: Oct. 09, 2025. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf.

[13] “Block cipher mode of operation.” Accessed: Oct. 09, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

[14] “NIST Special Publication 800-77 - Revision 1 - Guide to IPsec VPNs.” Accessed: Oct. 09, 2025. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175Br1.pdf.

[15] “Message authentication code.” Accessed: Oct. 09, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Message_authentication_code.

[16] “Random oracle.” Accessed: Oct. 09, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Random_oracle.

[17] “RSAES-OAEP Encryption Scheme - Algorithm specification and supporting documentation.” Accessed: Oct. 12, 2025. [Online]. Available: https://www.inf.pucrs.br/calazans/

graduate/TPVLSI_I/RSA-oaep_spec.pdf.

[18] "GitHub - armory-boot." WithSecure Corporation, Accessed: Nov. 07, 2025. [Online]. Available: https://github.com/usbarmory/armory-boot.

[19] "Github Armory Witness OS - HAB.go." WithSecure Corporation, Accessed: Nov. 08, 2025. [Online]. Available: https://github.com/transparency-dev/armored-witness-os/blob/main/internal/hab/hab.go.

[20] "Github - crucible - NXP HABv4 Secure Boot utility." WithSecure Corporation, Accessed: Nov. 08, 2025. [Online]. Available: https://github.com/usbarmory/crucible/tree/master/cmd/habtool.

[21] "Github - Makefile-pki." WithSecure Corporation, Accessed: Nov. 08, 2025. [Online]. Available: https://github.com/usbarmory/usbarmory/blob/master/software/secure_boot/hab-pki/Makefile-pki .

[22] "Creating a Certificate Authority." Nitrokey, Accessed: Jul. 13, 2025. [Online]. Available: https://docs.nitrokey.com/nitrokeys/features/openpgp-card/certificate-authority#.

[23] "AN5215 - i.MX 6 Temperature Sensor Module." NXP, Accessed: Nov. 08, 2025. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN5215.pdf.

[24] "Coreboot." Accessed: Nov. 07, 2025. [Online]. Available: https://www.coreboot.org/.

[25] "Keylime - Documentation." Accessed: Oct. 07, 2025. [Online]. Available: https://keylime.readthedocs.io/en/latest/index.html.

[26] "Host Integrity at Runtime and Start-up (HIRS)." Accessed: Nov. 07, 2025. [Online]. Available: https://github.com/nsacyber/HIRS.

[27] "Trenchboot - Use Cases." Accessed: Oct. 07, 2025. [Online]. Available: https://trenchboot.org/theory/UseCases/.

[28] "Github - tpm2-tools." Accessed: Nov. 07, 2025. [Online]. Available: https://github.com/tpm2-software/tpm2-tools.

[29] "Github - Go-Attestation." Accessed: Nov. 07, 2025. [Online]. Available: https://github.com/google/go-attestation.

[30] "Github - Go-TPM." Accessed: Nov. 07, 2025. [Online]. Available: https://github.com/google/go-tpm.

[31] "tinygo.org - A Go Compiler For Small Places." Accessed: Nov. 07, 2025. [Online]. Available: https://tinygo.org/.

[32] "tinygo.org - Go language features." Accessed: Nov. 07, 2025. [Online]. Available: https://tinygo.org/docs/reference/lang-support/.

[33] "Github - tamago-example." Accessed: Nov. 07, 2025. [Online]. Available: https://github.com/usbarmory/tamago-example.

[34] "Github - armored-witness." Accessed: Nov. 07, 2025. [Online]. Available: https://github.com/transparency-dev/armored-witness.

[35] "Trustworthy Platform Module (TwPM)." Dasharo, Accessed: Oct. 16, 2025. [Online]. Available: https://twpm.dasharo.com/.

[36] “Common Criteria for Information Technology Security Evaluation.” Wikipedia, Accessed: Nov. 10, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Evaluation_Assurance_Level.

[37] “Return-oriented programming.” Accessed: Oct. 17, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Return-oriented_programming.

[38] *The keyed-hash message authentication code (HMAC)*. National Institute of Standards and Technology (U.S.), 2008.

[39] *Security Reference Manual for i.MX 6Dual, 6Quad, 6Solo, and 6DualLite Families of Applications Processors - IMX6DQ6SDLSRM*. NXP, 2012.

[40] “TCG D-RTM Architecture.” Trusted Computing Group, 2013, Accessed: Aug. 02, 2025. [Online]. Available: https://trustedcomputinggroup.org/resource/d-rtm-architecture-specification/.

[41] *QorIQ LS1046A Security (SEC) Reference Manual, Rev. 0, 05/2017 - LS1046ASECRM*. NXP, 2017.

[42] “Digi Embedded Yocto - CAAM (Cryptographic Accelerator and Assurance Module).” 2018, Accessed: Sep. 24, 2025. [Online]. Available: https://docs.digi.com/resources/documentation/digidocs/90001548/reference/bsp/v4-9_6ul/r_caam_6ul.htm?TocPath=Digi%20Embedded%20Yocto%7CSystem%20development%7CLinux%20kernel%7CLinux%20%20v4.9%20Board%20Support%20Package%7CDevices%20and%20interfaces%7C_5.

[43] *i.MX 6UltraLite Applications Processor Reference Manual*. NXP, 2019.

[44] “’Shocking’ hack of psychotherapy records in Finland affects thousands.” The Guardian, 2020, Accessed: Oct. 15, 2025. [Online]. Available: https://www.theguardian.com/world/2020/oct/26/tens-of-thousands-psychotherapy-records-hacked-in-finland.

[45] “UM11225 - NXP SE05x T=1 Over I^2C Specification.” NXP, 2020, Accessed: Sep. 22, 2025. [Online]. Available: https://www.nxp.com/webapp/Download?colCode=UM11225.

[46] “AN12662 - Binding a host device to EdgeLock SE05x.” NXP, 2020, Accessed: Sep. 22, 2025. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN12662.pdf.

[47] “tpm2-software community - Remote Attestation With Tpm2 Tools.” 2020, Accessed: Nov. 07, 2025. [Online]. Available: https://tpm2-software.github.io/2020/06/12/Remote-Attestation-With-tpm2-tools.html#simple-attestation-with-tpm2-tools.

[48] *UM10204 I^2C - bus specification and user manual - Rev. 7.0*. NXP, 2021.

[49] “AN12514 - SE050 - User Guidelines.” NXP, 2021, Accessed: Oct. 10, 2025. [Online]. Available: https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/secure-authentication/1757/1/AN12514.pdf.

[50] “AN12413 - SE050 APDU Specification.” NXP, 2021, Accessed: Sep. 22, 2025. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN12413.pdf.

[51] “AN12436 - SE050 configurations.” NXP, 2023, Accessed: Jul. 16, 2025. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN12436.pdf.

[52] “GitHub - Nitrokey HOTP Verification Tool.” 2025, Accessed: Sep. 02, 2025. [Online].

Available: https://github.com/Nitrokey/nitrokey-hotp-verification.

[53] *Heads - Wiki*. 2025.

[54] “CBOR Codec - GO.” 2025, Accessed: Oct. 05, 2025. [Online]. Available: https://github.com/fxamacker/cbor.

[55] *Encyclopedia of Cryptography, Security and Privacy - Third Edition*. 2025.

[56] “USB armory wiki - Ordering information.” WithSecure Corporation, 2025, Accessed: Nov. 05, 2025. [Online]. Available: https://github.com/usbarmory/usbarmory/wiki/Ordering-information .

[57] “TamaGo’s Cryptographic Acceleration and Assurance Module (CAAM) driver.” WithSecure Corporation, 2025, Accessed: Sep. 24, 2025. [Online]. Available: https://github.com/usbarmory/tamago/tree/master/soc/nxp/caam/rtic.go.

[58] “TamaGo’s Secure-Nonvolatile Storage (SNVS) driver.” WithSecure Corporation, 2025, Accessed: Jun. 25, 2025. [Online]. Available: https://github.com/usbarmory/tamago/blob/master/soc/nxp/snvs/snvs.go.

[59] “TamaGo’s Cryptographic Acceleration and Assurance Module (CAAM) driver.” WithSecure Corporation, 2025, Accessed: Sep. 24, 2025. [Online]. Available: https://github.com/usbarmory/tamago/tree/master/soc/nxp/caam.

[60] “One-Time-Programmable (OTP) fusing tool.” WithSecure Corporation, 2025, Accessed: Jul. 01, 2025. [Online]. Available: https://github.com/usbarmory/crucible.

[61] “u-boot repo - Introduction habv4.” NXP, 2025, [Online]. Available: https://source.denx.de/u-boot/u-boot/blob/master/doc/imx/habv4/introduction_habv4.txt.

[62] “Secure boot (Mk II).” WithSecure Corporation, 2025, Accessed: Jul. 13, 2025. [Online]. Available: https://github.com/usbarmory/usbarmory/wiki/Secure-boot-(Mk-II).

[63] “Trusted Platform Module 2.0 Library - Part 1: Architecture - Version 184,” Trusted Computing Group, 2025. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-1-Version-184_pub.pdf.

[64] “Trusted Platform Module 2.0 Library - Part 3: Structures - Version 184,” Trusted Computing Group, 2025. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-3-Version-184_pub.pdf.

[65] “Trusted Platform Module 2.0 Library - Part 2: Structures - Version 184,” Trusted Computing Group, 2025. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-2-Version-184_pub.pdf.

[66] “AMD64 Architecture Programmer’s Manual - Volume 2: System Programming.” AMD, 2025, Accessed: Aug. 02, 2025. [Online]. Available: https://docs.amd.com/v/u/en-US/24593_3.43.

[67] “ARM DEN 0113 - DRTM Architecture for Arm.” ARM, 2025, Accessed: Aug. 02, 2025. [Online]. Available: https://developer.arm.com/documentation/den0113/latest/.

[68] “GitHub - tpm2-totp.” 2025, Accessed: Nov. 07, 2025. [Online]. Available: https://github.com/tpm2-software/tpm2-totp.

[69] “USB armory wiki - Secure-boot-(Mk-II).” WithSecure Corporation, 2025, Accessed: Nov.

05, 2025. [Online]. Available: https://github.com/usbarmory/usbarmory/wiki/Secure-boot-(Mk-II).

[70] W. Arthur, D. Challener, and K. Goldman, *A practical guide to TPM 2.0: Using the new trusted platform module in the new age of security*. Springer Nature, 2015.

[71] A. S. Banks, M. Kisiel, and P. Korsholm, “Remote Attestation: A Literature Review,” *CoRR*, vol. abs/2105.02466, 2021, [Online]. Available: https://arxiv.org/abs/2105.02466.

[72] E. Barker, L. Chen, A. Roginsky, A. Vassilev, and R. Davis, *Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography*. National Institute of Standards and Technology, 2018.

[73] M. Bellare and P. Rogaway, “Introduction to modern cryptography,” *Ucsd Cse*, vol. 207, p. 207, 2005.

[74] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, “Remote ATtestation procedureS (RATS) Architecture,” no. 9334. RFC Editor, Jan. 2023, doi: 10.17487/RFC9334.

[75] C. Bormann and P. E. Hoffman, “Concise Binary Object Representation (CBOR),” no. 7049. RFC Editor, Oct. 2013, doi: 10.17487/RFC7049.

[76] L. Chen, *Recommendation for key derivation using pseudorandom functions*. National Institute of Standards and Technology (U.S.), 2024.

[77] E. Dushku and N. Dragoni, “Remote Attestation in IoT Devices,” in *Encyclopedia of Cryptography, Security and Privacy*, S. Jajodia, P. Samarati, and M. Yung, Eds. Cham: Springer Nature Switzerland, 2025, pp. 2089–2092.

[78] M. J. Dworkin, *Recommendation for block cipher modes of operation - methods and techniques*. National Institute of Standards and Technology, 2001.

[79] M. J. Dworkin, *Recommendation for block cipher modes of operation - GaloisCounter Mode (GCM) and GMAC*. National Institute of Standards and Technology, 2007.

[80] M. J. Dworkin, *Recommendation for block cipher modes of operation - the CMAC mode for authentication*. National Institute of Standards and Technology, 2016.

[81] M. Fischlin and A. Mittelbach, “An Overview of the Hybrid Argument.” Cryptology ePrint Archive, Paper 2021/088, 2021, [Online]. Available: https://eprint.iacr.org/2021/088.

[82] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, 2nd ed. Chapman & Hall/CRC, 2014.

[83] B. Kauer, “Oslo: improving the security of trusted computing.,” in *USENIX Security Symposium*, 2007, vol. 24, p. 173.

[84] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger, “TPM-FAIL: TPM meets Timing and Lattice Attacks,” *CoRR*, vol. abs/1911.05673, 2019, [Online]. Available: http://arxiv.org/abs/1911.05673.

[85] A. Nilsson, P. N. Bideh, and J. Brorsson, “A Survey of Published Attacks on Intel SGX,” *CoRR*, vol. abs/2006.13598, 2020, [Online]. Available: https://arxiv.org/abs/2006.13598.

[86] N. I. of Standards and Technology, “Security Requirements for Cryptographic Modules,” U.S. Department of Commerce, Washington, D.C., Federal Information Processing Standards Publications (FIPS) 140-2, 2015. doi: 10.6028/NIST.FIPS.180-4.

[87] N. I. of Standards and Technology, “Security Requirements for Cryptographic Modules,” U.S. Department of Commerce, Washington, D.C., Federal Information Processing Standards Publications (FIPS) 202, 2015. doi: 10.6028/NIST.FIPS.202.

[88] S. W. Smith, “Attestation,” in *Encyclopedia of Cryptography, Security and Privacy*, S. Jajodia, P. Samarati, and M. Yung, Eds. Cham: Springer Nature Switzerland, 2025, pp. 126–127.

[89] A. Sperrle, “tpm2-audit,” *GitLab repository*. GitLab, 2025, [Online]. Available: https://gitlab.kit.edu/uymtd/tpm2-audit.

[90] P. Svenda *et al.*, “TPMScan: A wide-scale study of security-relevant properties of TPM 2.0 chips,” in *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024, vol. 2024, No. 2, pp. 714–734, doi: 10.46586/tches.v2024.i2.714-734.

[91] B. Tao and H. Wu, “Improving the Biclique Cryptanalysis of AES,” in *Information Security and Privacy*, Cham, 2015, pp. 39–56.

Appendix

Key-Scheme related data structure

```
type KeyParameter uint32

func (parameters KeyParameter) ECCCurve()   tpm2.TPMECCCurve
func (parameters KeyParameter) RSAKeybits() int
```

KeyParameter Datatype

```
type SymmetricKeyParameter uint32

func (symKeyParams SymmetricKeyParameter) Bits() int
```

Symmetric KeyParameter Datatype

```go
type TPMKeyMetadata struct {
    // SET (1):
    //      Previously saved contexts of this object may not be loaded after Startup(CLEAR).
    // CLEAR (0):
    //      Saved contexts of this object may be used after a Shutdown(STATE) and subsequent Startup().
    STClear bool `cbor:"STClear"`
    // SET (1):
    //      Approval of USER role actions with this object may be with an HMAC session
    //      or with a password using the authValue of the object or a policy session.
    // CLEAR (0):
    //      Approval of USER role actions with this object may only be done with a policy session.
    UserWithAuth bool `cbor:"UserWithAuth"`
    // SET (1):
    //      Approval of ADMIN role actions with this object may only be done with a policy session.
    // CLEAR (0):
    //      Approval of ADMIN role actions with this object may be with an HMAC session
    //      or with a password using the authValue of the object or a policy session.
    AdminWithPolicy bool `cbor:"AdminWithPolicy"`
    // SET (1):
    //      The object is not subject to dictionary attack protections.
    // CLEAR (0):
    //      The object is subject to dictionary attack protections.
    NoDA bool `cbor:"NoDA"`
    // SET (1):
    //      The private portion of the key may be used to decrypt.
    // CLEAR (0):
    //      The private portion of the key may not be used to decrypt
    Decrypt bool `cbor:"Decrypt"`
    // If UserWithAuth is false the authPolicy must be specified.
    // For EKs which abide to a template specified in the TPM specification, it is either policyA, policyB,
or policyC.
    AuthPolicy []byte `cbor:"AuthPolicy"`
}

func (metadata TPMKeyMetadata) TPMAObject() tpm2.TPMAObject

func (metadata TPMKeyMetadata) GetAuthPolicy() tpm2.TPM2BDigest
```

TPMKeyMetadata Datastructure

Representation of Keys

```go
type PublicKeyContext struct {
    Pk     PublicKey
    Scheme KeyScheme
    Nonce  []byte
}

func (pkc PublicKeyContext) Name() []byte

func (pkc PublicKeyContext) QualifiedName(qualifiedNameParent []byte)
```

Public Key Context Datastructure

```go
type PublicKey interface {
    // The specified Key-Type aids in casting.
    Type() KeyType
    // The key's unique data in correspondence to the TPM specification
    Unique() []byte
}

func (pkc PublicKeyContext) Name() []byte

func (pkc PublicKeyContext) QualifiedName(qualifiedNameParent []byte)
```

Public Key Interface

```go
type TPMAttestationInfo struct {
    QualifiedSigner []byte `cbor:"QualifiedSigner"`
    // ClockInfo: The values of Clock, resetCount, restartCount and Safe.
    ClockInfo [17]byte `cbor:"ClockInfo"`
    // FirmwareVersion which is relevant to security.
    FirmwareVersion [8]byte `cbor:"FirmwareVersion"`
}

func (metadata TPMAttestationInfo) Bytes() []byte
```

TPMAttestationInfo Datastructure

```go
type TPMCertificiateHead struct {
    // A marshaled tpm-attest always starts with the following data, for reference visit (Trusted Platform Module 2.0 Library Part 1: Architecture | v184) page 214
    //
    // magic[4B]      : A magic value which allows the TPM to differentiate between internal and external data structures.
    //                  An unrestricted key can for example sign all data except of data starting with the magic value.
    // attestationType[2B] : The type of the attestation.
    attestationType [2]byte
    // qualifiedSigner(len[2B] | qualifiedSigner[len]): The qualified name of the key signing the certificate.
    qualifiedSigner []byte
    // extraData(len[2B] | ExtraData[len]): External data providing freshness.
    extraData []byte
    // clockInfo(Clock[8B] | ResetCount[4B] | RestartCount[4B] | Safe[1B]): The TPMs current clock among others.
    clockInfo [17]byte
    // firmwareVersion[8B]
    firmwareVersion [8]byte
}

func (head *TPMCertificiateHead) Bytes() []byte
```

TPMCertificiateHead Datastructure

Glossary

ACPI	Advanced Configuration and Power Interface
AES	Advanced Encryption Standart is a Block Cipher.
APDU	Application Protocol Data Unit
BIOS	Basic Input Output System
CAAM	Crypto Accelerator and Assurance Module - a cryptographic processor.
CBC	Cipher-Block-Chaining - a Block Cipher Mode.
CBOR	Concise Binary Object Representation
CFB	Cipher Feedback a Block Cipher Mode.
CMAC	MAC-Scheme, see
CTM	Chain-of-Trust-Measurements
CTR	Counter - a Block Cipher Mode.
DES	Data Encryption Standard - an old block-cipher succeeded by AES
DRT	D-RTM Resources Table
DRT(M)	Dynamic Root of Trust (Measurement)
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Diffie-Hellman Key Exchange
Endian(ess)	See Endianess
EUF-CMA	Existentially Unforgeable Under an adaptive Chosen-Message Attack
GCM	Galouis Counter Mode a Block Cipher Mode.
HAB	High Assurance Boot
HMAC	A MAC-Scheme, see
I2C	Inter-Integrated Circuit
imx6ul	i.MX 6UltraLite Processor
IND-CCA	Indistinguishability under Chosen-Ciphertext Attacks
IND-CCA1	See IND-CCA.
IND-CCA2	See IND-CCA.
IND-CPA	Indistinguishability under Chosen-Plaintext Attack
IND	Indistinguishability
KDF	Key-Derivation Function
KDFa	A KDF used by the TPM, see.
KDFe	A KDF used by the TPM, see.

MAC	Message Authentication Code
MITM	Machine-In-The-Middle attack
Mk II	A synonym for the USB Armory Mk II.
NIST	United States National Institute of Standards and Technology
OTPMK	One Time Password Master Key
PCR	Platform Configuration Register
PUF	Physical Unclonable Function
RO	Random Oracle - an idealized Hash-Function
RSAES-OAEP	An RSA Encryption Scheme combined with OAE Padding
RT	Root-of-Trust
RT(M)	Root of Trust (Measurement)
RTIC	RunTime Integrity Checker
SE050	Secure Element - a cryptographic processor
SHA	Secure Hash Algorithm - a standartized Hash Algorithm.
SMM	System Management Mode
SNVS	Secure Non-Volatile Storage
SoC	System on a Chip
SRT(M)	Static Root of Trust (Measurement)
TPM	Trusted Platform Module
TCG	The *Trusted Computing Group* defines the TPM specification.