# Engineering Trustworthy Systems by Minimizing and Strengthening their TCBs using Trusted Computing

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Andreas Fuchs aus Frankfurt am Main
Tag der Einreichung: 30. Dezember 2020, Tag der Prüfung: 17. Mai 2021

1. Gutachten: Prof. Dr. Michael Waidner
2. Gutachten: Prof. Dr. Carsten Bormann
Darmstadt 2021

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
Security in Information
Technology

Engineering Trustworthy Systems by Minimizing and Strengthening their TCBs using Trusted Computing

Doctoral thesis by Andreas Fuchs

1. Review: Prof. Dr. Michael Waidner
2. Review: Prof. Dr. Carsten Bormann

Date of submission: 30. Dezember 2020
Date of thesis defense: 17. Mai 2021

Darmstadt, Technische Universität Darmstadt

Jahr der Veröffentlichung der Dissertation auf TUprints: 2022

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-196767
URI: http://tuprints.ulb.tu-darmstadt.de/id/eprint/19676

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Attribution – ShareAlike 4.0 International
http://creativecommons.org/licenses/by-sa/4.0

# Danksagung

An dieser Stelle möchte ich meinen besonderen Dank allen Personen entgegen bringen, ohne deren Hilfe diese Dissertation nicht zu stande gekommen wäre:

An erster Stelle möchte ich meinem Doktorvater Prof. Dr. Michael Waidner sowie meinem Zweitgutachter Prof. Dr. Carsten Bormann für die Betreuung dieser Arbeit sowie der freundlichen Hilfe, Ideengebung und Themenfokussierung danken.

Außerdem danke ich meiner Lebensgefährtin Nina Kasper für ihre Geduld, ihr Verständnis und ihren Ansporn insbesondere in den letzten Etappen meiner Promotion, ihre Unterstützung bei der Reflexion und thematischen Strukturierung sowie ihrer tatkräftigen Hilfe bei der Vorbereitung meiner mündlichen Prüfung. Ich danke auch meinen Eltern, Annemarie und Joachim Fuchs, die meine Entscheidung zu promovieren unterstützt und mich die gesamte Zeit über begleitet haben.

Des Weiteren danke ich meinen Vorgesetzten Prof. Dr. Carsten Rudolph und Prof. Dr Christoph Krauß dafür, dass sie eine Arbeitsumgebung geschaffen haben, in der diese Promotion entstehen konnte, sowie für die zahlreichen Gespräche auf fachlicher wie persönlicher Ebene. Mein besonderer Dank gilt auch Dr. Sigrid Gürgens dafür, dass sie mich das Handwerkszeug zur Durchführung dieser Promotion gelehrt hat. Unser reger bereichernder und kostruktiver Austausch wird mir immer als Highlight in Erinnerung bleiben.

Letztlich möchte ich mich bei den vielen Kollegen, Projektpartnern und Co-Autoren, Christian Plappert, Dustin Kern, Dr. Florian Schreiner, Hendrik Decke, Henk Birkholz, Lukas Jäger, Maria Zhdanova, Prof. Dr. Nicolai Kuntze, Peter Hüwe, Dr. Roland Rieke und Jürgen Repp, für die tolle Zusammenarbeit bedanken. Außerdem danke ich Susanne Koch für ihre Unterstützung bei der Korrektur des endgültigen Texts.

# Erklärungen laut Promotionsordnung

## §8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

## §8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

## §9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

## §9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 30. Dezember 2020

A. Fuchs

# Abstract

The Begriff Trusted Computing Base (TCB) beschreibt den Teil eines IT-Systems, der für die Durchsetzung einer bestimmten Sicherheitseigentschaft des Systems verantwortlich ist. Um ein vertrauenswürdiges System entwickeln zu können, muss die TCB so sicher wie möglich sein. Dies läßt sich erreichen, indem die Anzahl, Größe und Komplexität der Systembestandteile, die Teil der TCB sind, reduziert wird und indem gehärtete Komponente als TCB eingesetzt werden. Im schlechtesten Fall umspannt die TCB das gesamte IT-System. Im besten Fall besteht die TCB nur aus einem gehärteten Root of Trust, wie beispielsweise einem Hardware Security Module (HSM). Ein solches besonders sicheres HSM mit vielen Fähigkeiten ist das Trusted Platform Module.

Diese Arbeit demonstriert, wie die TCB eines Systems größtenteils oder sogar ausschließlich auf ein TPM reduziert werden kann, um eine Vielzahl von Sicherheitsvorgaben, insbesondere für Eingebettete Systeme zu realisieren. Die betrachteten Szenarien beinhalten Vorgaben zur Absicherung von gerätegespeicherten Daten auch während Updates der Firmware, der Durchsetzung von Firmwareproduktlinien zur Laufzeit, der Absicherung von Bezahlidentitäten in Plug-and-Charge-Kontrollgeräten, der Aufzeichung von Auditinformationen über Attestationsdaten und ein generisches rollenbasiertes Zugriffsmanagement. Um diese Lösungen analysieren zu können, wird der Begriff der TCB um eine weitere Dimensionen erweitert, die den Lebenszyklus des Geräts umfasst. Außerdem wird ein Ansatz zur Konstruktion solcher Systeme basierend auf einem formalen Rahmenwerk präsentiert.

Diese Szenarien zeigen auf, dass selbst komplexe Sicherheitsvorgaben in kleinen und damit starken TCBs umgesetzt werden können. Die Herangehensweise zur Umsetzung solcher Vorgaben kann häufig durch einen Entwicklungsprozess basierend auf formalen Methoden oder durch additiv funktionales Entwickeln geleitet werden, bei dem das Basissystem in jedem Schritt um weitere Funktionalitäten erweitert wird. In allen Fällen kann ein System mit starken Sicherheitseigenschaften erzielt werden.

# Abstract

The Trusted Computing Base (TCB) describes the part of an IT system that is responsible for enforcing a certain security property of the system. In order to engineer a trustworthy system, the TCB must be as secure as possible. This can be achieved by reducing the number, size and complexity of components that are part of the TCB and by using hardened components as part of the TCB. Worst case scenario is for the TCB to span the complete IT system. Best case is for the TCB to be reduced to only a strengthened Root of Trust such as a Hardware Security Module (HSM). One such very secure HSMs with many capabilities is the Trusted Platform Module (TPM).

This thesis demonstrates how the TCB of a system can be largely or even solely reduced to the TPM for a variety of security policies, especially in the embedded domain. The examined scenarios include the policies for securing of device resident data at rest also during firmware updates, the enforcement of firmware product lines at runtime, the securing of payment credentials in Plug and Charge controllers, the recording of audit trails over attestation data and a very generic role-based access management. In order to allow evaluating these different solutions, the notion of a dynamic lifecycle dimension for a TCB is introduced. Furthermore, an approach towards engineering such systems based on a formal framework is presented.

These scenarios provide evidence for the potential to enforce even complex security policies in small and thus strong TCBs. The approach for implementing those policies can often be inspired by a formal methods based engineering process or by means of additive functional engineering, where a base system is expanded by increased functionality in each step. In either case, a trustworthy system with high assurance capabilities can be achieved.

# Contents

# 1 Introduction

Today almost all electric systems in automotive industry, production system and civil infrastructure are based on IT systems. The security of these IT systems is essential for their correct operation whilst their impact on the overall systems' safety is increasing. Evolving technologies such as autonomous driving, Industry 4.0 and digitalized critical infrastructure clearly illustrate the significance.

The security of any IT system directly results from the trustworthiness of its underlying components. In order to constitute a trustworthy system, one relies on the so-called Trusted Computing Base (TCB). This term originates from the Trusted Computer System Evaluation Criteria (TCSEC) also known as the Orange Book[35] and refers to the hardware, firmware and software of this system, that are critical for its trustworthiness / security. The TCB is responsible for executing and enforcing a security policy that the main application code – running on top of the TCB – cannot circumvent. In order to design a TCB, a single or a set of Roots of Trust (RoT) can be used. The term RoT originates from the Trusted Computing Group (TCG). It refers to those components of a TCB that need to be trusted without being verifiable as opposed to those that can be verified.

However, the term TCB – as opposed to the term RoT – is not an unambiguous differentiating element. The term TCB usually describes the computing basis for a certain system layer or functionality and a corresponding security policy. As such, the RoT can be referred to as a TCB for the RoT-related functionalities and security properties of the system. The complete platform can also be referred to as TCB for the overall system's functionality, if the security policy extends up to the highest layers of application logic.

The term Trusted Computing was introduced by the Trusted Computing Platform Alliance and its successor, the TCG [92, 120]. It refers to a set of technologies that can be used to establish a (larger) TCB on a platform in order to enable a trustworthy platform design. These technologies include different RoTs – such as the TPMs (see Section 2.4) – as well as protocols and infrastructure elements necessary to establish trustworthy systems.

In an ideal scenario, the TCB for a security policy will only consist of its minimal but strong RoTs. The further the TCB can be reduced towards its RoT, the smaller the part of the platform that can be subject to an attack. This implies an increased level of security for the platform.

## 1.1 Motivation

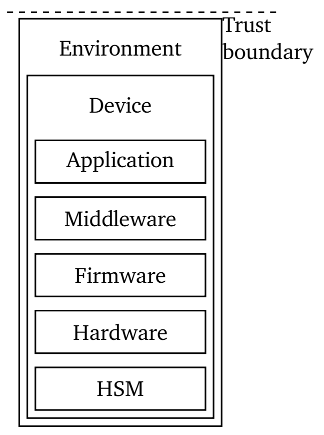
In order to implement a strong Trusted Computing Base (TCB), one of the main goals is to reduce its size and complexity and to ensure it is based on hardened system components. For any given security policy, this can be achieved by different means and the TCB for different security policies of a system may vary accordingly. This general idea can be demonstrated in these two following simple examples.

The first example is the security policy for file access permissions. In order to only allow access to certain files by the authorized application the TCB consists of the vfs file system modules of the OS kernel, the main CPU hardware, the hard disk controller, the hard disk itself as well as the firmwares of all these components and the connections between them. Furthermore, all other parts of the system with access to these components are part of the TCB, e.g. other drivers being executed in the same authorization area. One means to reduce the TCB of such systems is to enable hard disk encryption with a key stored on the main CPU or entered by the user. In these system, the hard disk, the hard disk controller and all affected connections are removed from the TCB; hence the TCB is accordingly reduced.

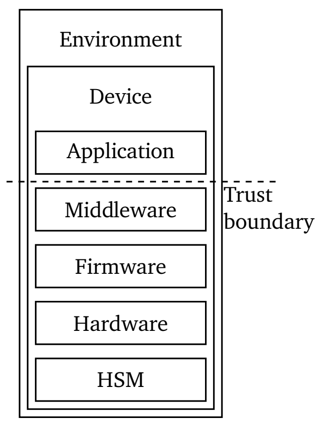
The second example is the authentication of an embedded device using a cryptographic signature. The security policy states that only the specific device must be able to perform signature operations and that the cryptographic key must not readable to entities outside of the device. In a simple implementation, a cryptographic key is stored on the hard disk. The application loads the key into memory and calculates the signature. In this implementation the TCB consists of two parts, the TCB for file storage from the previous paragraph as well as the application using the key. This simple implementation represents the worst case because the TCB covers the whole device.

The security in these examples can be increased by reducing the size of the respective TCBs. This reduction can be achieved by using Hardware Security Modules (HSMs), such as embedded smartcards, secure elements or Trusted Platform Modules (TPMs). In an implementation like this, the cryptographic key is stored on the HSM and the signature operation is performed by the HSM. The cryptographic key will never be known outside the HSM and the HSM ensures the absolute confidentiality of this key. Hence, the TCB would only include the HSM instead of the whole device. The trustworthiness of this device regarding this security policy would thus be higher.

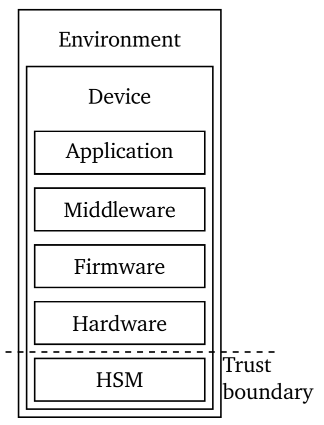
This thesis investigates and implements possibilities to reduce and strengthen the TCB of devices for a certain set of security policies. These policies target (i) the selective confidentiality of data based on device attributes, (ii) the confidentiality of data during firmware updates, (iii) the protection of remotely provided credentials, (iv) the storage of integrity audit logs and (v) implementation of a dynamic role-based access control.



(a) TCB Example 1



(b) TCB Example 2



(c) TCB Improvement

Figure 1.1: Trusted Computing Base Examples

In order to differentiate these approaches with respect to their security strength a new notion of TCB using a second lifecycle based dimension is introduced. In order to create links between different security policies and steer TCB design a formal decision making process model is introduced.

## 1.2 Research Goals

This thesis demonstrates that it is feasible to design a trustworthy platform implementing a multitude of security policies based on Trusted Computing technologies and the TPM.

The first contribution is the introduction of a broader understanding for “TCBs”. The concept of a TCB is extended by a second dimension that represents the different phases in a device’s lifecycle. This is necessary in order to differentiate the improvements made by the presented technological solutions.

The second contribution is the application of this notion of TCB to set of use cases and security policies where the TCB shall be reduced. This extended notion of TCB helps to analyze the current state of the art and to illustrate the advancement through reduction that can be applied to these security policies when applying Trusted Computing technologies.

These applications stem from a variety of use cases. Two are seemingly simple standard tasks (1. and 2.), one use case demonstrates the challenges when integrating with an existing communication standard (3.) and two use case demonstrates a completely new kind of application (4. and 5.). The security policies to implement in these scenarios are:

1. *Selective Confidentiality of Data and Code based on Device Attributes* The ability to define the specific variation of a device attribute – i.e. product line number – during the boot of the platform to selectively decrypt data and code.
2. *Protection of Device Resident Data bound to Firmware enabling Updates* The ability to protect device resident data – data that remains stored on the device even during updates – notwithstanding firmware updates.
3. *Secure Credential usage in Plug and Charge* The ability to protect provisioning and contract credentials in an ISO 15118 scenario, including import and remote deployment of such credentials.
4. *Offline Integrity Audit Logs* The ability to determine the software status of a platform without a synchronous connection – i.e. unidirectionally or at a later point in time.
5. *Role-based access management* The ability to enforce advanced role systems on cryptographic primitives in a trust gateway device, including ability of rights delegation and revocation in an automotive context.

As third contribution, this thesis shows that a formal framework and engineering process can be constructed for designing trustworthy platforms. This engineering process supports Trusted Computing based TCBs for a wide range of capabilities. It supports the notions of authenticity, confidentiality and trust and is applicable to systems of devices and protocols. It enables abstraction and refinement, composition and decomposition and embeds them in a systematic process for engineering.

## 1.3 Outline

Chapter 2 gives an introduction to TCBs and introduce the extension of the TCB to a dynamic lifecycle-based dimension. It then investigates candidates for TCBs and evaluate the usage of TPMs as TCB. Then a basic introduction into TPMs is provided in Section 2.4. Chapter 3 through Chapter 7 provides technological solutions based on TPMs to implement strong TCBs for the aforementioned security policies. Chapter 8 briefly introduces the formal framework and engineering process that allows the design and implementation of TCBs. Chapter 9 concludes this thesis and gives an outlook to future works.

%

# 2 Background and Approach

## 2.1 The Trusted Computing Base (TCB)

The term Trusted Computing Base (TCB) can be tracked back to the Orange Book[35]. It describes

> the totality of protection mechanisms within it [the system], including hardware, firmware, and software, the combination of which is responsible for enforcing a computer security policy.

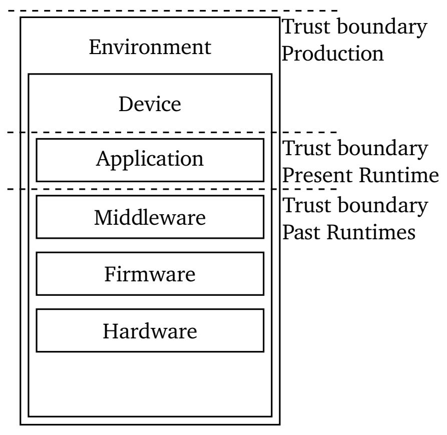
Thus, the TCB denotes that part of the system that forms the basis to enforce a security policy. A typical system implements a variety of security policies and each security policy may rely on a different TCB. The authors of the Orange Book further explain that

> the ability of a trusted computing base to enforce correctly a unified security policy depends on the correctness of the mechanisms within the trusted computing base, the protection of those mechanisms to ensure their correctness, and the correct input of parameters related to the security policy.

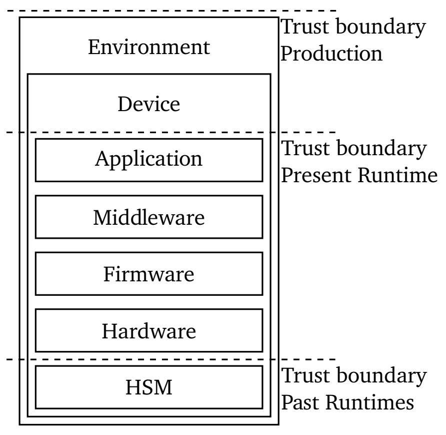
An even earlier mention of the term TCB originates from [108]. The author argues that

> generally speaking, small, simple and localized mechanisms are easier to get correct, and more easily to be shown correct, than large, complex, or diffuse ones. The first task in the design of a secure system, therefore, is to find a way of structuring it so that its security mechanisms are localized as much as possible, and are as small and as simple as possible.

He argues that a Domain Separation Mechanism is required to divide the system into TCB and non-TCB domains. He also argues that *physical* separation is one of the best means to do so. These and other definitions and approaches to a systematic view on TCB vs non-TCB of a system are static instead of dynamic. They only focus on the structural aspects at a snapshot point in time, i.e. they either neglect certain points in time of the



(a) TCB Example 1



(b) TCB Example 2

Figure 2.1: Classic Trusted Computing Bases

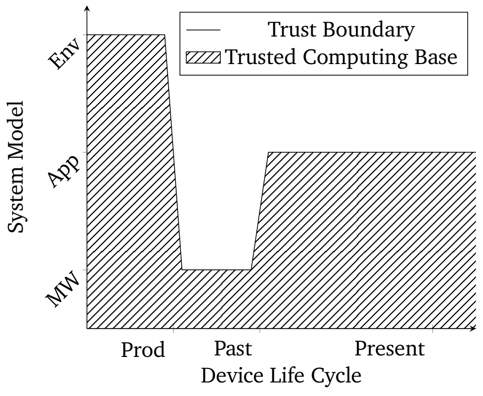
device lifecycle (such as production or setup) or they view the totality of functions over the complete lifecycle and neglect differences of solutions during certain steps of a device lifecycle.

## 2.2 A second dimension for TCBs

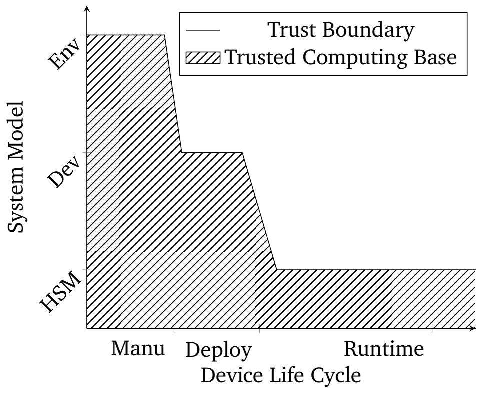
The described typical view of TCBs is spatial. This however does not take into account that a device goes through different phases over its lifetime. A device will be produced, provisioned, used, re-provisioned, re-configured, decomissioned, refurbished, etc. The TCB may (and usually will) differ between many of these phases. Figure 2.1 shows the TCB boundaries for the phases production vs usage for the example given in Section 1.1.

In order to grasp the size and thus the security of the TCB of a given system throughout its complete lifecycle, a different illustration can be used though. Figure 2.2 demonstrates a two-dimensional way to illustrate the TCB of a system over its lifetime. The y-axis describes the spacial relations, similarly to previous illustrations. The x-axis describes the different phases during the systems lifetime.

Using this method it is easier to compare different solutions regarding their overall TCB concerning the complete lifetime of a system. The smaller the area of the system, the smaller the TCB and the higher the assurance for the security policy. Furthermore, it is possible to assess the trustworthiness of different components and lifecycle steps. In this example, we can see that the TCB impact of the HSM is only half as large as that of the



(a) TCB Example 1



(b) TCB Example 2

Figure 2.2: Trusted Computing Base over Device Lifecycle

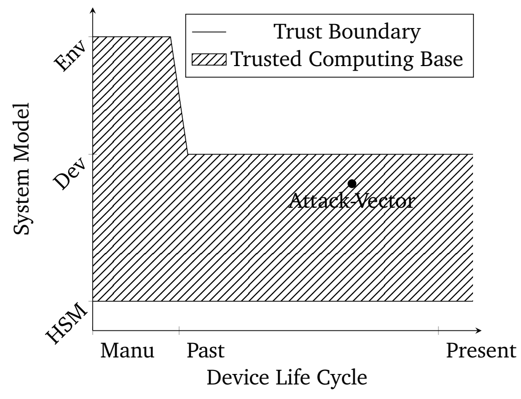
remainder of the device. Also the runtime phase was weighted at three quarters of the overall lifetime impact in contrast to the production and deployment phases.

$$\begin{tikzcd}
 & \boxed{\text{auth device}} & \\
\boxed{\text{conf } k_{prv}} \arrow[ur] & & \boxed{\text{correct device since challenge}} \arrow[ul]
\end{tikzcd}$$

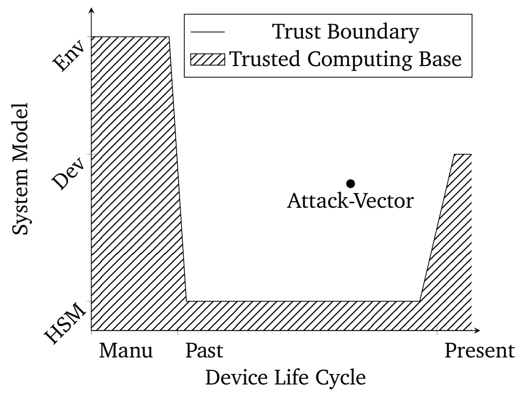
Figure 2.3: Relating Security Policies

The advantages of this approach become even more apparent when looking at the security policy of device authentication. This security policy is dependent on the confidentiality of the private key and the authenticity of the nonce origin. This means that no relay attack for the nonce is performed. Figure 2.4 illustrates this approach. Chapter 8 provides a formal framework for these kinds of reasoning.

Based on this, the comparison between a non-HSM system and an HSM regarding the authentication based on the confidentiality of the private key becomes even more apparent. Figure 2.4 shows this vast difference. The reason is that a successful attack on a non-HSM system at any point in the past will break the device authentication forever. In contrast a successful attack on an HSM based system in the past does not affect a current authentication attempt, i.e. a device can recover from a previous attack whilst the non-HSM system cannot. This became very apparent during the heartbleed recovery times



(a) Non-HSM-System



(b) HSM-System

Figure 2.4: Trusted Computing Base for Authentication

[24] where non-HSM systems required invalidation of certificates and reprovisioning of new private keys whilst HSM systems only required an update of the openssl software libraries.

## 2.3 HSMs as TCBs

Arguably the best means for separation of two domains is the introduction of a second physical entity. By using a separate physical entity for execution of the TCB it is well-separated from the rest of the system. This entity can then excute and enforce the security policy but as few non-security policy related tasks as possible. These kinds of elements are often referred to as Hardware Security Modules (HSM) or Secure Elements (SE). Also several Trusted Execution Environments (TEE) and other CPU extensions claim to support this mode of operation for security policies. All of these approaches differ immensely in their functional richness as well as their security guarantees.

Figure 2.5 gives a qualitative overview of the functional richness on the y axis and the security assurance on the x axis. For the purpose of the works presented in this thesis, a security assurance level in the class of smart cards is desired. To start of a physical or spacial segregation between the TCB and the rest of the system is provided. Secondly, the TCB element is equipped with security measures against diverse attacks, such as voltage glitch detectors, photo resistors, etc. The latter is asserted by the vendor and by Common Criteria certification. Solutions such as segregation through CPU virtualization or TrustZone do not provide these assurances.

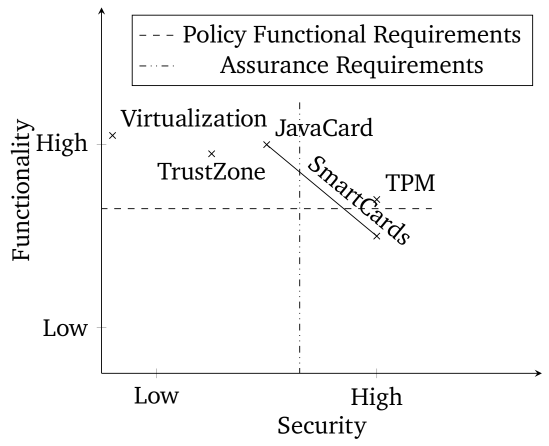


Figure 2.5: Hardware Security Modules comparison

A special case are programmable SmartCards, often referred to as JavaCards – following the name of the application API of these SmartCards. Whilst their base OS and hardware usually provide a very high security assurance, the applets are oftentimes not as highly assured. If the applets would also undergo security certification, the resulting systems would turn out very expensive. Even more so, as it needs to be done for each JavaCard applet and version separately. It can also be argued that the availability of a Turing-Complete interface poses a security problem to begin with.

Regular SmartCards – providing a certified PKCS11 applet – do provide a security certification and are usually cheap, but they are very limited in their functionalities. Given these requirements against functionality, assurance and cost, the TPM is the only suitable candidate. It offers high assurance whilst providing suitable functionalities to implement the desired security policies as a TCB.

## 2.4 Trusted Platform Modules 2.0

The Trusted Platform Module (TPM) is a specification by the Trusted Computing Group describing a security coprocessor that is oftentimes implemented a separate chip or firmware implementation of a secure compartment. It features the capabilities of securely storing cryptographic credentials as well as a secure execution environment for cryptographic operations for said credentials. The second iteration of the TPM, namely the TPM 2.0 Library Specification [128], has been released by the Trusted Computing Group (TCG) in October 2014. It provides a catalog of functionalities that can be used to build TPMs for different platforms. This includes support for cryptographic algorithms, which can be chosen from the TCG's Algorithm Registry [60]. This registry is updated regularly in order to provide and extend cryptographic agility. The TPM supports asymmetric algorithms such as RSA and ECC but also symmetric algorithms such as AES, hash based HMACs, and hash functions such as SHA1 or SHA2. The TPM also supports special features such as measured boot where hash values of each component of the boot chain are calculated and stored in so called Platform Configuration Registers (PCR) enabling the verification of the platform state, i.e., the integrity of the firmware. This enables the concept of sealing where access to keys is only possible when the platform state is trustworthy. Another feature is called enhanced authorization which allows the forming of arbitrary policy statements to use the TPM. The TPM also includes non-volatile memory (NV-RAM) where counters, bitmaps, and even extended PCR can be stored. Additional features are outlined in [32]. In addition, the realization of a TPM is very flexible, realizable both as a dedicated hardware chip as well as a firmware TPM. A dedicated hardware TPM provides a shielded location for key storage and usage which makes it very hard to extract, copy or duplicate the keys from the TPM. It is also possible to generate keys inside the TPM and enforce that keys never leave the TPM. Since the TPM itself is very resource constraint especially with respect to its internal flash memory, keys can also be stored as encrypted key objects in the flash of the host processor and are only decrypted within the TPM. Keys can also be generated outside of the TPM and then be imported. These import key blobs can be encrypted as well, such that no attacker can intercept or duplicate them during transit. It is important to note that the keys used for decrypting these import blobs are so-called *restricted decryption keys* of the TPM (aka. storage keys), which cannot be used for general purpose decryption. In June 2015, the TPM 2.0 was approved by ISO as successor to TPM 1.2 in ISO/IEC 11889:2015 [126].

### 2.4.1 TPM 2.0 Cryptographic Capabilities

First and foremost, a TPM serves as a cryptographic module to a system. Its purpose is not to accelerate cryptographic operations, but to improve security. A such the TPM can securely create and store cryptographic keys and provides an execution environment for cryptographic operations.

In order to work with a cryptographic key, the TPM offers a set of commands. Those relevant to this thesis are:

- **TPM2_CreatePrimary**: This command is used to create a key from the TPM's internal seeds. These keys are the top-level objects in a TPM key hierarchy. The most prominent of these keys are the Storage Root Key (SRK) and the Endorsement

Key (EK). The SRK is used as encrypting storage key for all other keys of a typical TPM enabled system. The EK is used to assess the originality of a TPM.
- **TPM2_Create**: This command is used to create all kinds of objects for the TPM. This includes cryptographic keys usable for authenticating to external entities. During creation, an (enhanced authorization) policy can be provided that restricts usage of the created object.
- **TPM2_Sign**: This command calculates a signature using a private key created via *TPM2_Create*. These signatures can be used for authenticating a device or for asserting data integrity and origin.
- **TPM2_Unseal**: In order to retrieve the content of an encrypted, sealed object, the TPM2_Unseal command can be used. If the object was created or imported with a certain usage policy, this policy needs to be fulfilled for usage. This is done using so-called policy sessions.
- **TPM2_Import**: This command imports an external TPM key object into the TPM. This key object is a TPM data structure and thus contains key attributes as well as an optional policy. The key object is encrypted, such that it can be imported without the host CPU ever gaining access to the private portion.

A cryptographic key in the TPM can be associated with a fine-grained usage policy (cf. Section 2.4.3) as well as a set of basic key attributes. The most important attributes are:

- **sign**: The key can be used for signing data.
- **decrypt**: The key can be used to decrypt data.
- **restricted**: If a key is a restricted signing key it can be used by the TPM to attest TPM data structures. If a key is a restricted decrypt key it can be used to decrypt TPM data structures, e.g. during TPM2_Load or TPM2_Import.

### 2.4.2 TPM 2.0 NV Index storage

Besides offering cryptographic keys, the TPM also offers some of its internal flash as user-defined storage, so-called NV indices. These indices can (similarly to keys) be associated with a policy (cf. Section 2.4.3) as well as a set of attributes. Those attributes used throughout this thesis are:

- **ordinary**: An NV index of type ordinary can be used by the user as a general purpose storage area for reading and writing of arbitrary data.
- **counter**: An NV index of type counter is a 64-Bit unsigned integer value that can only ever be incremented and never decremented or arbitrarily written to. The

initial value of such a counter NV index is the highest counter value ever recorded on such a TPM. Thus even the deletion and re-creation of such a counter NV index does not decrement its value.

In order to interact with the TPM's NV index interface, the following commands are used throughout this thesis:

- **TPM2_NV_DefineIndex**: This command creates a new NV-Index.
- **TPM2_NV_Write**: This command allows the writing of data to an NV-Index of type *ordinary*. Note that after the first NV_Write operation the NV_Written bit of the NV-Index is set.
- **TPM2_NV_Read**: This command is used to read data from an NV-Index.
- **TPM2_NV_Increment**: This command is used to increment an NV-Index of type *counter*.

### 2.4.3 Enhanced Authorization

With Enhanced Authorization, any object that requires authorization can either be authorized using a secret value assigned during creation (similar to TPM 1.2) or using a policy scheme. Enhanced Authorization consists of a set of policy elements that are each represented via a TPM command. Currently, eighteen different policy elements exist that can be concatenated to achieve a logical *and* in arbitrary order and unlimited number. Two of these policy elements – PolicyOr and PolicyAuthorize – act as logical *or*. Due to implementation requirements, policy statements are, however, neither commutative nor distributive. Once defined they need to be used in the exact same order. In this paper, we use the following notation: $Policy_{abc} := PolicyX_1() \wedge PolicyX_2() \wedge \ldots PolicyX_n()$ where $Policy_{abc}$ is the "name" for this policy, such that it can be referred to from other places and $PolicyX_i()$ describes the $n$ concatenated TPM2 Policy commands that are required to fulfill this policy.

- **TPM2_StartAuthSession**: In order to fulfill any authorization policy, the application needs to start a policy session using the TPM2_StartAuthSession command. Then the actual policy statements are subsequently satisfied by invoking the corresponding TPM commands.
- **TPM2_PolicyOR**: This command represents an OR-clause in a policy statement. Up to eight sub-policy branches are unified in such an OR-clause. The OR-clauses can however be stacked in order to allow for more branches.

- **TPM2_PolicyNV**: This command is used to compare an NV index content to a reference value. The possible comparison operations are equal, less than, greater than, etc.
- **TPM2_PolicyNVWritten**: This command checks whether an NV index has been written before or if it has never been written before as determined the “written” attribute of the NV index.
- **TPM2_PolicySigned**: This command requests that a TPM generated nonce has been signed by a referenced public key and thus ensures authorization and freshness of authorization for this policy session.
- **TPM2_PolicyPCR**: This policy statement ensures that the Platform Configuration Registers (PCRs) have a certain value. The PCRs are used to represent the integrity state of a platform by recording its booted software and configurations.
- **TPM2_PolicyAuthorize**: This command allows the activation of policies *after* the definition of an object. In order to achieve this, a public key is registered with a policy. This policy element then acts as a placeholder for any other policy branch that is signed with the corresponding private key.

Given the complicated workflow and calculation of policy digest values, the TCG has defined a declarative language to represent TPM policies as JSON strings [129]. These are easier to work with when using a TPM Software Stack than evaluating policies manually.

### 2.4.4 TPM 2.0 Other Commands

Other commands used in this thesis include the following:

- **TPM2_ActivateCredential**: This command is part of the TPM credential deployment capabilities. A credential provider can encrypt a certificate with a key known to reside on the TPM and denote a certain additional key inside the encrypted data. The TPM decrypts the data and verifies that the denoted key is known and only then returns the credential data back to the user. This way, the credential provider is guaranteed assurance that credential usage is only possible if the denoted key is known to the same TPM and has the reported properties.
- **TPM2_EvictControl**: Since the TPM only has limited persistent internal memory, objects are usually stored externally, encrypted with a TPM-resident key. Any object can be made persistent inside the TPM on request by the owner. The TPM2_-EvictControl command is used to store an object persistently in the TPM or to delete an object from persistent storage.
- **TPM2_Clear**: This command is used to clear a TPM e.g. during a refurbishment process. All keys are deleted or invalidated except for the primary keys used in the

endorsement hierarchy. Similarly, all owner based NV index values are deleted and the initial counter value for NV counter indices are set back to 0.

### 2.4.5 TPM 2.0 Software Stack

Accompanying the TPM specifications, the TCG has engaged in the specification of a TPM Software Stack (TSS) 2.0 for this new generation of TPMs [133]. It consists of multiple Application Programming Interfaces (APIs) for different application scenarios. A so-called Feature Level API has been published [130] with an accompanying specification for a JSON based policy language [129]. It targets high-level application in the Desktop and Server domain. For lower-level applications, such as embedded applications, the so-called Enhanced System API specification (ESYS) [132] was released in conjunction with the TPM Command Transmission Interface (TCTI) [122] for IPC module abstraction. The ESYS API and TCTI were designed such that embedded applications can access TPM 2.0 functionalities whilst minimizing requirements against the platforms. As such, they do not require persistent storage and only link against basic cryptographic code.

# 3 Firmware Update and Device Resident Data

A security policy for device resident data is that this data must not be accessible outside of the device. This obviously includes the prevention of direct readout via a designated port or directly from the storage medium. It should also include the prevention of illegitimate software replacement whilst allowing regular software updates. In order to implement this with a minimal TCB the TPM 2.0 can be used. This chapter builds upon my contributions to and reuses text from [48] which include the complete general approach, the complete realization concept and some implementation contributions.

## 3.1 Background on Firmware Updates and Data Encryption

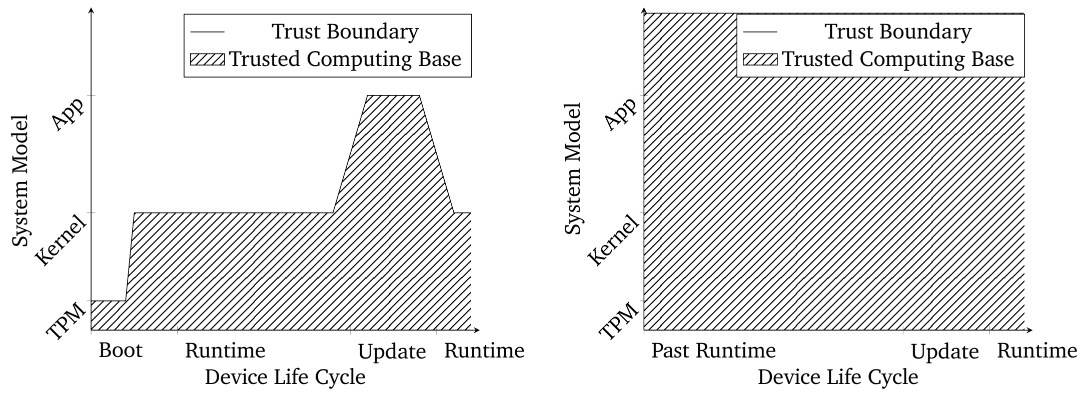
The idea of encrypting data at rest to protect against attacks is nothing new. cryptsetup/LUKS [42], Bitlocker [84] or even TrueCrypt [18] have implemented this feature for a very long time for Desktop machines. For embedded machines this becomes more challenging though, since there is no external source – the user – to provide the volume key or a volume key encryption key to the system (and only that system) during boot. Thus, a different source for securing this key is needed – an HSM or TPM. They provide a storage that is bound, e.g. soldered, into the correct platform and cannot be extracted from the system.

The second challenge to increase the security of such an embedded system is that attackers could pretty easily replace the original software with a software of their own that then dumps the volume key from the connected HSM. As solution, TPM 1.2 [61, 62, 63] introduced so-called `sealing`. Here, an encrypted secret is stored together with a set of reference PCR values that represent the original (and trustworthy) device firmware. Only if the PCR integrity values recorded during boot match those reference PCR values will the TPM release the volume key to the system. Bitlocker [84] and tpm-luks[1] implement

[1]https://github.com/shpedoikal/tpm-luks

this feature as well. However, these approaches do not account for typical firmware update schemes of embedded devices.

Firmware upgrades are a necessity of all software based systems to fix bugs and vulnerabilities. During such a software upgrade, not all data stored on a device shall be replaced by the incoming firmware upgrade. The two categories of data not included in a software upgrade are large unchanged data sets and data created on the device during operation.



(a) Data Protection (b) Rollback Protection

Figure 3.1: Trusted Computing Base for Firmware Update with TPM 1.2

The requirements that come along with this is that consecutive software versions need to have access to this data, while it needs to be stored inaccessible to malicious firmware images. Also this data must become inaccessible to old versions of the firmware after an upgrade in order to prevent so called "downgrade attacks". In a *downgrade attack*, an attacker installs an outdated version of the firmware that has known vulnerabilities in order to exploit them. Such attacks have for example been used for breaking the first generation of PlayStation Portable [139].

The classic realization of sealing, as employed by TPM 1.2 [61, 62, 63] is illustrated in Figure 3.1. The lifecycle axis shows 4 phases of the device's phases that are relevant to the presented solution. It starts with the devices boot, followed by a longer runtime phase. Then an update phase occurs that also includes another boot, followed by the runtime phase with the next version of the firmware. The system model axis includes the TPM, the firmware's OS kernel and the userland of applications.

Such a solution comes with a set of drawbacks, because they do not allow the updating of referenced PCR values from the sealed blobs. This would have to be done by allowing the updater to reseal the data for the state after upgrading, which requires the upgrade module to be privileged. This is reflected in the bump to the TCB during the update phase in Figure 3.1a. Furthermore, it was not possible to disallow usage of the old seal for accessing the data. This is reflected by including the complete system in the TCB in Figure 3.1b.

With the framework for “Enhanced Authorization” (EA) in TPM 2.0, it is possible to achieve this use case respecting the circumstances and requirements outlined above. In the following, the set of requirements is listed, the concept described, and a prototypical implementation outlined.

## 3.2 Requirements for Firmware Updates with Device Resident Data

The requirements for securing large data sets and personal information during a firmware upgrade can be summarized as follows:

1. Provide confidentiality of vendor’s intellectual property and user data.
2. Only allow original manufacturer firmware to read / write data.
3. Prevent access by old firmware after an upgrade to new firmware.
4. Do not require a special update mode, where the updater component has access to the data unsealing credentials.

## 3.3 Concept for Firmware Updates with Device Resident Data

The concept for providing these requirements can be divided into three phases: Provisioning, Firmware Upgrade, and Firmware Usage.

### Provisioning

After provisioning of the device, the very first thing should be the definition of an NV index that represents the device’s currently required minimal firmware version. This has to be done first, in order to ensure that the NV index is initialized with a value of 0 (read Zero), cf. Section 2.4.2 for details. This counter is initialized as a single 64 bit unsigned integer value and then incremented once in order to be readable:

$$NV_{Version} := TPM2\_NV\_DefineSpace(counter)$$
$$TPM2\_NV\_Increment(NV_{Version})$$

The vendor intellectual property and user data is stored in an encrypted container or partition on the devices flash that is stored separate from the firmware binaries. The key to this encrypted container is then sealed with the TPM to the following policy:

$$Policy_{Seal} := TPM2\_PolicyAuthorize(K^{pub}_{Manu})$$

This policy allow the manufacturer as the owner of $K^{priv}_{Manu}$ to issue new policies in the future for accessing the stored data. For the initial firmware of version $v1$, the manufacturer will provide the following policy:

$$\begin{aligned}Sig\big(K^{priv}_{Manu}, \, [&TPM2\_PolicyPCR(Firmware_{v1})\wedge \\ &TPM2\_PolicyNV(NV_{Version} \leq v1)]\big)\end{aligned}$$

The PCRs that represent the integer state of $Firmware_{v1}$ may be used until the nv index $NV_{Version}$ that represents the currently required minimal version exceeds the value of $v1$.

$$\begin{aligned}Policy_{Seal} :=& TPM2\_PolicyAuthorize\big(K^{pub}_{Manu}, \\ &\big[TPM2\_PolicyPCR(Firmware_{v1})\wedge \\ &\ TPM2\_PolicyNV(NV_{Version} \leq v1)\big]\big)\end{aligned}$$

Note that the signed policy cannot be part of those components of the firmware that is measured into the representing PCRs. The reason is that this would lead to a cyclic relation that cannot be fulfilled.

### Firmware Upgrade

Whenever a firmware upgrade is issued by the manufacturer, it will be accompanied by a newly signed policy, that (similar to the original policy) grants access to the encrypted container based on the PCR representation of that firmware. This access again is only granted until the NV index representing the minimum required version exceeds this firmware's version:

$$\begin{aligned}Sig\big(K^{priv}_{Manu}, \, [&TPM2\_PolicyPCR(Firmware_{v2})\wedge \\ &TPM2\_PolicyNV(NV_{Version} \leq v2)]\big)\end{aligned}$$

This leads to $Policy_{Seal}$ being:

$$\begin{aligned} Policy_{Seal} :=& TPM2\_PolicyAuthorize\big(K^{pub}_{Manu}, \\ & \big[TPM2\_PolicyPCR(Firmware_{v1}) \wedge \\ & \ TPM2\_PolicyNV(NV_{Version} \leq v1)\big] \\ & \vee \big[TPM2\_PolicyPCR(Firmware_{v2}) \wedge \\ & \ TPM2\_PolicyNV(NV_{Version} \leq v2)\big]\big) \end{aligned}$$

During the update process, the updater mode stores the firmware including its policy on the device's flash drive. However, it does not require access to the encrypted container.

A similar concept with TPM 1.2 would have required that the updater mode would have required access to the encrypted container and to reseal the secret, or the manufacturer would have to have known the secret and bind it for the new PCR values. This is one of the main benefits of this TPM 2.0 based approach.

### Firmware Runtime

Whenever a legitimate firmware version starts, it can unseal the necessary data and read/write to these storage areas. In order to invalidate access by outdated firmware versions, during each start, the firmware will check the currently stored minimal required firmware version inside the TPM counter and increment it to its own firmware version. This invalidates the usage of PolicyAuthorize branches for previous firmware versions, since they require a lower value for the $NV_{Version}$ counter. Firmware should only perform this increment when it successfully completed its self test and started up correctly, since a recovery of the previous version is impossible afterwards. Instead the issuing of a new firmware version would be required. Of course a manufacturer may choose to issue a certain recovery firmware version, or multiple such version by, e.g., encoding those as the odd version vs. regular firmware as even versions.

Incrementing of the NV counter is denoted as

$$TPM2\_NV\_Increment(NV_{Version}, v1)$$

which invalidates any older policies and thereby any outdated firmwares. This leads to:

$$\begin{aligned} Policy_{Seal} :=& TPM2\_PolicyAuthorize\big(K^{pub}_{Manu}, \\ & \big[TPM2\_PolicyPCR(Firmware_{v1}) \wedge \\ & \ TPM2\_PolicyNV(NV_{Version} \leq v1)\big]\big) \end{aligned}$$

A similar concept with TPM 1.2 could only have removed the sealed blobs for older versions from the flash drive, but in case an attacker was able to read out the flash in an earlier state, there would have been no possibility to actually disable these older policies with the TPM. This is another main benefit of this TPM 2.0 based approach.

## 3.4 Security Considerations of Firmware Updates with Device Resident Data

The presented concept relies on the correct cryptographic and functional execution of a TPM 2.0 implementation for the encryption and correct handling of the policy tickets respectively. Furthermore, the scheme relies upon a set of specific assumptions in order to function properly:

- The ticket signing private key of the vendor must be kept confidential, potentially using a TPM itself as well.
- The provisioning needs to refer to the correct ticket public key for verification.
- The scheme does not protect against runtime-attacks against software.

## 3.5 Prototypical Implementation of Firmware Updates with Device Resident Data

The described concept was implemented in a proof-of-concept demonstrator for a typical automotive Head Unit. An Intel NUC D34010WYK in combination with Tizen In-Vehicle Infotainment (IVI) [110] using Linux kernel 4.0 was chosen for this implementation. These 4th generation NUCs are one of first commercial off-the-shelf (COTS) devices equipped with a TPM 2.0 implementation. The demonstrator is shown in Fig. 3.2.

To protect vendor confidential and privacy sensitive data, the Linux LUKS implementation (Linux Unified Key Setup) is used to create and to open an encrypted container for storing this data. The key for the encrypted container in turn is protected by TPM 2.0's enhanced authorization mechanisms as described in Section 2.4.3.

For ease of demonstration the representation of firmware versions in the PCR values was simplified – namely not calculations of the overall firmware hashes. Instead of measuring the complete firmware at boot and extending a PCR with this measurement, a single file is measured into the PCR called $version\_file$. A TPM monotonic counter is used to represent the version number.

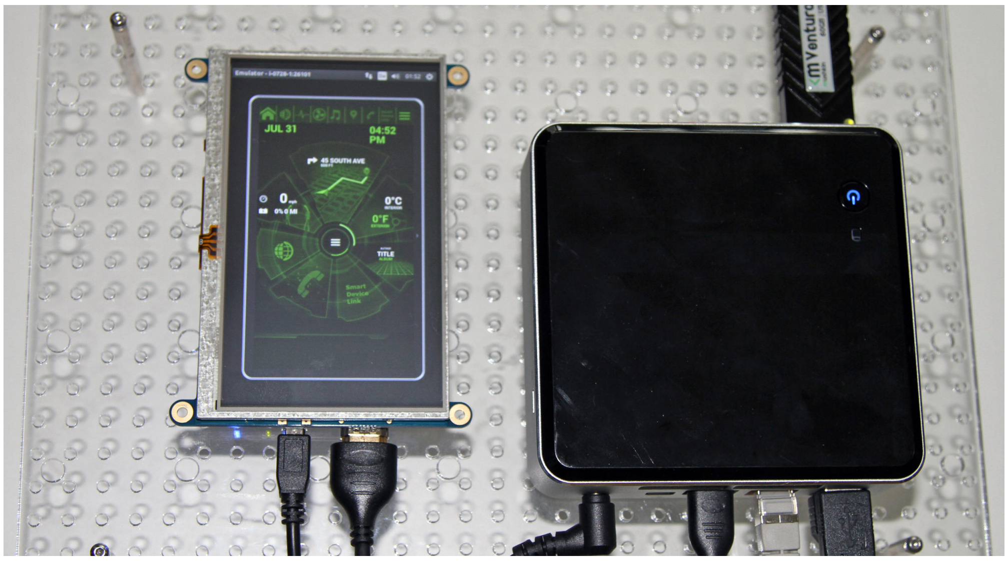


Figure 3.2: TPM 2.0 Head Unit Demonstrator

Note that the standard Linux Integrity Measurement Architecture (IMA) [29] could also not be used in practice for this scenario. Due to the event-driven startup mechanisms under modern Linux systems, the exact order of PCR extension can vary, which renders them unusable for sealing. Instead, the complete image would have to be measured from within the initial ramdisk. A future publication will demonstrate a possible approach based on squashfs [28].

In Algorithm 1 and 4, the TPM 2.0 equivalent of a Storage Root Key ($SRK$) – a TPM Primary Key under the Storage Hierarchy – is already computed. If the NV counter $NVC$ is already defined, it is used directly. Otherwise $NVC$ gets defined.

### Provisioning

When the device is started for the first time, the following steps shown in Algorithm 1 are executed for provisioning the protected storage. It consists of the definition of the NV version counter, the instantiation of the policy, the creation of the LUKS container and the sealing of the LUKS key.

**Algorithm 1** (Provisioning)**.**

```
// NV Creation
if not defined(NVC) then
```

```
    NVC := TPM2_NV_DefineSpace(counter)
  end if
  // Policy Initialization
  pubkey := TPM2_LoadExternal(pub_key_manu)
  policy := TPM2_PolicyAuthorize(pubkey)
  // LUKS creation
  S := generate_random_key()
  create_crypto_fs(S)
  // Sealing
  SRK := get_srk()
  enc(SRK,S) := TPM2_Create(S,SRK,policy)
  save(enc(SRK,S))
```

To ease readability, some details are wrapped within simplified function calls. All functions prefixed with $TPM2\_$ are the equivalent of the corresponding TPM functions. $not$ "defined(NVC)$ will check whether the NV index has been defined by performing a $TPM2\_NV\_Read$ and checking the resulting value. The public key $pub\_key\_manu$ of the manufacturer, loaded into the TPM, is bound to the object $S$ encrypted by $SRK$ via the policy calculated by $TPM2\_PolicyAuthorize$. The corresponding private key to $pub\_key\_manu$ now can be used to alter policies necessary for unsealing the encrypted key $S$.

### Firmware release

Algorithm 2 shows how the manufacturer can produce signatures for new firmware release versions with a corresponding TPM 2.0 policy without using a TPM. The function $compute\_policy$ calculates the policy based on the PCR value, after extending a PCR by the firmware digest, and the version of the firmware. This computation is performed by the manufacturer according to the TPM2.0 specification. This policy will be signed with the private key of the manufacturer. The device later must be able to associate the policy and the signature with the version to be installed.

**Algorithm 2** (Firmware release)**.**

```
  policy(version) := compute_policy(digest(version_file),version))
  signature(version) := sign(policy(version), priv_key_manu)
```

## Upgrade

Algorithm 3 shows the steps executed to install a new firmware version signed by the manufacturer. The new version will be active after the next reboot (see Algorithm 4). The version of a signed firmware to be installed must be greater or equal to the current NV counter because this demonstrator should be able to execute the provisioning process several times and the hardware TPM's monotonic counter cannot be decremented again. To be more precise, even if the counter is undefined by $TPM2\_NV\_UndefineSpace$ it is not possible to reset the counter to a smaller value because for the next definition the counter will be initialized to be the largest count held by any NV counter over the lifetime of the TPM. In practice it would be possible to store the initial policy directly in the firmware image for version 1.

**Algorithm 3** (Upgrade)**.**

```
version := get_latest_signed_version_number()
signature(version) := get_signature(version)
policy(version) := get_policy(version)
save(version_file, version)
```

In order to perform the simulated upgrade of the firmware binary, the value encoded within the firmware representing version file is incremented.

## Runtime

The steps described in Algorithm 4 are executed in the boot process to mount the encrypted container. They consist of the PCR extension with the digest of the version file, the satisfaction of the policy, using this policy for unsealing the container key, the opening of the encrypted container and then the invalidation of old firmware by incrementing the NV version counter.

**Algorithm 4** (Mount encrypted file system)**.**

```
/* Satisfying the policy */
version := load(version_file)
TPM2_PCR_Extend(digest(version_file))
approved_policy := load_policy(policy(version))
signature := load_signature(signature(version))
pubkey := TPM2_LoadExternal(pub_key_manu)
ticket := TPM2_VerifySignature(signature,pubkey,approved_policy)
session := TPM2_StartPolicySession()
```

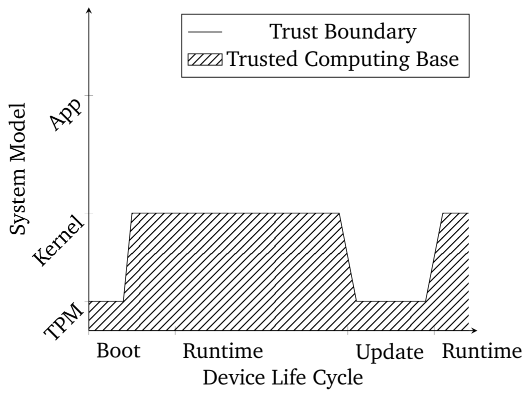
```
session.TPM2_PolicyPCR(PCR)
session.TPM2_PolicyNV(NVC,<=,version)
policy := TPM2_PolicyAuthorize(pubkey,ticket,approved_policy)
/* Unsealing the container key and mount */
SRK := get_srk()
enc(SRK,S) := load()
TPM2_Load(enc(SRK,S))
S := TPM2_Unseal(enc(SRK,S),session)
mount_crpyto_fs(S)
firmware_self_test()
/* Invalidate old firmwares */
while TPM2_NV_Read(NVC) < version do
   TPM2_NV_Increment(NVC)
end while
```

The functions $load\_policy$ and $load\_signature$ will load the policy and the signature associated by the manufacturer with the version stored in the version file. The function $session.TPM2\_Policy$ represents an execution of those policy commands on the TPM, referring to the policy session $session$. For the signed approved policy, a ticket derived from this policy and the public key of the manufacturer is verified against its signature using $TPM2\_VerifySignature$.

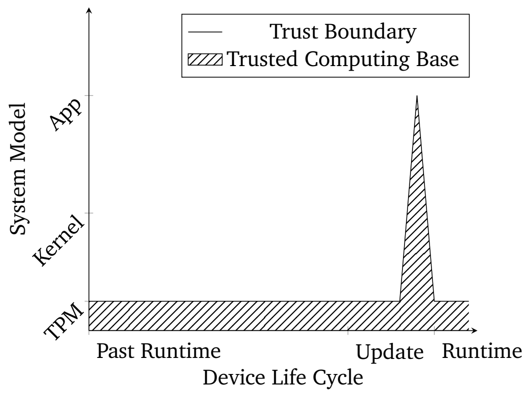
The current policy value of the session will be compared with the approved policy and the TPM then validates that the parameters to $TPM2\_PolicyAuthorize$ match the values used to generate the ticket. After the crypto file system is mounted, the device should perform a self test $firmware\_self\_test$ and the NV counter will be incremented until the value which corresponds to the current firmware version is reached. Thus, the object $enc_{SRK}(S)$ cannot be unsealed by firmware versions less than $version$, providing protection against downgrade attacks.

## 3.6 Evaluation of Firmware Updates with Device Resident Data

This new approach is only possible by using new features introduced by TPM 2.0 such as NV-RAM counters and Enhanced Authorization. It secures device-resident data by ensuring that only new firmware upgrades of the manufacturer can be installed and downgrade attacks or attempts to install malicious firmware upgrades are prevented. The TCB of this new approach is illustrated in Figure 3.3. The down-bump during the update phase in Figure 3.3a represents the fact that the authorization for accessing the data is handled by only the TPM. During runtime however the kernel remains part of the TCB



(a) Data Protection



(b) Rollback Protection

Figure 3.3: Trusted Computing Base for Firmware Update with TPM 2.0

since it handles the bulk en/decryption. The rollback protection illustrated in Figure 3.3b shows that userland is only involved once during the end of the update procedure when the TPM's version counter is incremented to the current version.

The prototypical implementation for an automotive head unit protects device-resident data of the manufacturer (i.e., navigation maps) and of the car user (i.e., contacts and preferred navigation destinations) against unauthorized access before, during, and after an upgrade. This general concept can be applied in different application scenarios using different TPM and TSS 2.0 profiles.

# 4 Runtime Product Lines

One interesting security policy for embedded platforms is related to the trend towards using software product lines. Lately, the difference between a standard and a premium product or even between products of completely different functionalities are not the result of differences in hardware but of the supported functionalities in its software. These are called software product lines, where the firmware for a switch and a router differ in terms of software features and not in terms of hardware. This however has the potential of disallowed changing a standard product into a premium product without reimbursing the vendor appropriately. The approach presented in this chapter showcases the implementation of a software product line that acts upon boot, thus configuring the product for runtime. It further enables the distribution of unified firmware images, i.e. single images for a whole product line of devices. It is based upon individual feature encryption using trusted computing technologies. This chapter builds upon my contributions to and reuses text from [49] which include the majority of the general approach, the complete realization concept and some implementation contributions.

## 4.1 Background on Runtime Product Lines

The development of Information Technology (IT) products has shifted drastically from separate and designated designs per model to unified hardware architectures with different software versions. Different products of similar kind nowadays only differ by their enclosure and firmware – and sometimes additional interfaces. For example, an Original Equipment Manufacturer (OEM) may produce routers and firewalls that only differ in the enclosure, firmware, and number of network ports. The firmware, however, differentiates a router from a firewall.

At the beginning of the industrial production age, an individual production process for a product was used. The next step was the employment of general *product lines*. A product line of similar products is produced by a common factory where the different products are assembled and configured using a common set of parts. A typical example is the production of individual variations of a car model. A buyer can use a shopping tool to choose from a set of predetermined parts and functionalities to build their own vehicle. The goal was to have a unified physical product line to reduce costs by providing variations of a product to a customer.

With software production, *software product lines* were introduced: “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [23]. The goal is a unified source code for generating similar software versions at compile-time from a shared set of software components using conditional compilation techniques, e.g., `#ifdef` in the C programming language or flags such as `--enable-feature[=arg]` in the GNU Autoconf package [21]. The benefits of software product lines are improved productivity, increased quality, decreased cost and labor needs, and much faster time to market [23].

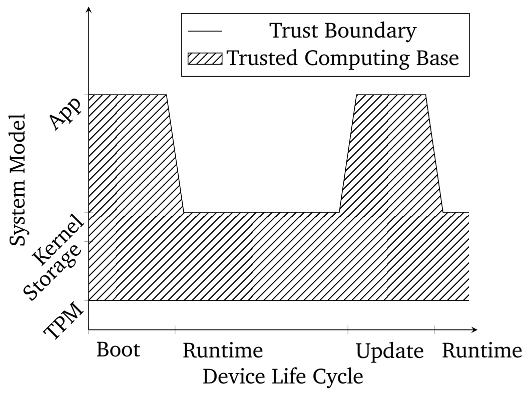


Figure 4.1: Trusted Computing Base for typical Runtime Product Lines

The next iteration of device product lines will go even beyond this and is referred to this step as *runtime firmware product lines*. Instead of compiling and packaging different firmware images from the same code collection, a unified firmware image will be delivered for a whole product series. The product customization will happen on the device itself. Hence, one firmware image contains several features and the device itself decides whether to unlock a certain feature or not.

These decisions are commonly made in software during the boot phase of the system or by the features themselves. Somewhere on the device a map of activated features is stored and then evaluated. The problem here is manyfold. Firstly, the feature map can often be altered by attackers. Secondly, the code evaluating the feature map can be altered, i.e. a comparison operation reversed. Thirdly, an attacker can extract the code representing a feature and insert it into other software. As such, the TCB of such a system spans over the whole device and whole runtime of the device, as depicted in Figure 4.1. The lifecycle axis shows 4 phases of the device's phases that are relevant to the presented solution. It starts with the device boot, followed by a longer runtime phase. Then an update phase occurs that also includes another boot, followed by the runtime phase with the next version of the firmware. The system model axis includes the TPM, the firmware's OS kernel and the userland of applications. An attacker able to manipulate or read out the storage and gaining access to kernel or userspace during boot or update would be able to extract data or (in case of local configuration) change the bits of enabled features. During runtime a typical separation of process execution context defends from such attacks from userland. These are represented by the trust boundary in Figure 4.1.

## 4.2 General Idea for Runtime Product Lines

In the presented approach, a device is equipped with a TPM 2.0 that can be a dedicated hardware chip but also a software implementation to save costs. The TPM acts as security anchor by providing secure storage, secure execution, and additional features for controlling access (cf. Section 2.4).

The general idea of this approach is as follows. A *unified firmware image* for a product line consists of several separate read-only *feature filesystems*. Each feature filesystem contains a specific feature set and is encrypted using a unique cryptographic *feature key*. For each model of a product line, a different model number is stored inside the devices' TPM. This TPM-resident model number serves as a basis of the runtime configuration of the firmware image to be booted. By verifying a TPM-Policy, only the allowed features for a specific model of a product line are unlocked, i.e., decrypted. Note that accessing the unified firmware image is only possible on an original OEM device containing the preconfigured TPM. If required, additional read-write filesystems for local data *localdataFS* (e.g., for `/etc`, `/home`, `/srv`) and temporary files *tmpFS* (e.g., for `/var`) can be mounted. The protection of the *localdata* filesystem against manipulation is out of scope of this chapter, but can be easily realized, for example, using the approach presented in [48].

Figure 4.2 shows an example to illustrate the approach in more detail. The unified firmware image is structured by the means of overlay read-only filesystems. A base file system *BaseFS* contains those parts that are shared between all models. It includes the operating system (OS) kernel and a multitude of basic libraries and services. Each model then comes with its feature filesystems *FeatureFS-X*. The example contains four of these filesystems, *FeatureFS-0* to *FeatureFS-3*. They contain the differences of this feature compared to the underlying filesystem. This may be additional files, removed files, or even altered files. For any given model, a stack can then be built that includes the base

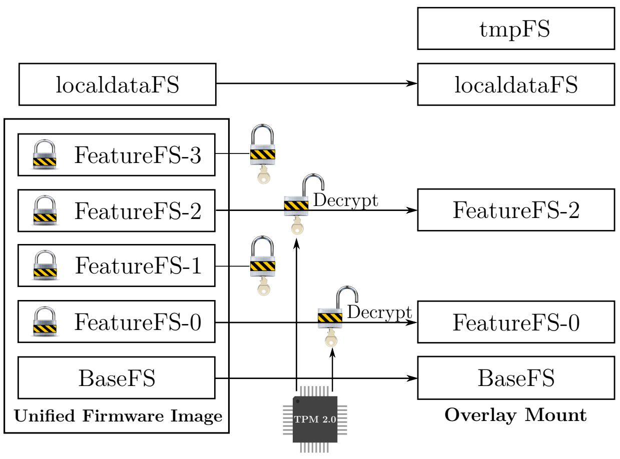


Figure 4.2: General Idea

filesystem and a series of feature filesystems that are overlayed in order to provide the actual final boot system. The TPM decides, based on the model number, which features to unlock, i.e., unseal the respective *feature key* which was used to encrypt the feature filesystem to decrypt it. The unlocking of individual features during runtime configuration is illustrated by the example shown in Figure 4.3. The model number is used to denote a specific combination of encrypted feature filesystems for a model of the product line. In the example, the model number has a length of 4 bits to be able to encode the four different feature filesystems. The binary model number of $0101_b$ will enable the feature filesystems *FeatureFS-0* and *FeatureFS-2* to be loaded, since the respective bits were set within the model number's feature bitmask. The figure shows the unlocking of *FeatureFS-2*. As mentioned above, *FeatureFS-2* is encrypted using *FeatureFS-2-Key*. This key is encrypted using the TPM-resident *ImportTargetKey* and stored together with the *Policy:* $0100_b$ in an integrity-protected *Sealed Key BLOB*. The integrity of the BLOB can be verified using also the *ImportTargetKey*. The BLOB and the encrypted *FeatureFS-2* form the *FeatureFS-2 Data Object*. In step 1, the TPM imports the sealed key blob, decrypts the *FeatureFS-2-Key*, and checks the integrity of the key BLOB. The TPM verifies the policy in step 2, by checking that the third bit from the right of the policy $0100_b$ equals 1 (since it was also set to 1 in the model number $1101_b$). If this is true, the decrypted *FeatureFS-2-Key* is unsealed and transfered to the host CPU (step 3), which uses it to decrypt and mount *FeatureFS-2* as part of the overlay mount (step 4).

In the example, the entire model number was used to encode the used features. It is also possible to define certain parts of the model number for encoding the used features.

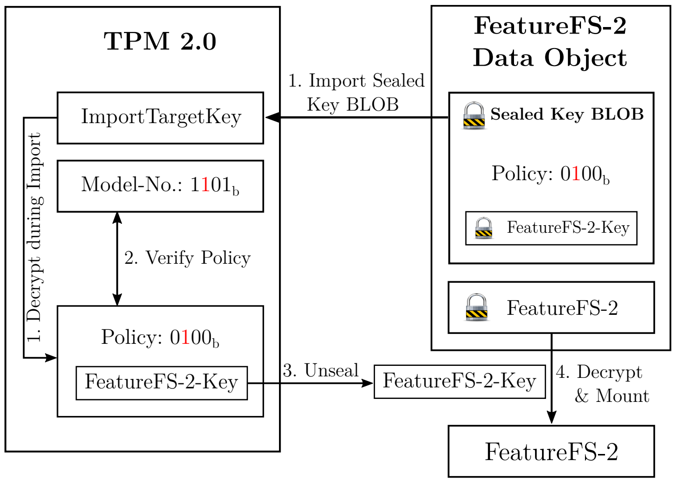


Figure 4.3: Runtime Feature Unlocking

The remaining parts can be used for other purposes, e.g., to encode the color of the casing. In principle, the modeling of the model number can be arbitrary complex to realize certain policies (cf. Section 4.5.2).

## 4.3 Related Work for Runtime Product Lines

### 4.3.1 Secure Runtime Product Lines

The work in this chapter extends the idea and concept of compile-time firmware product lines based upon unified software repositories to run-time firmware product lines based upon unified firmware images. The most prominent example for compile-time firmware product lines are the Yocto [40], OpenEmbedded [1], and BitBake [85] projects. The BitBake project provides the build-chain and environment for a compile-time firmware product line. OpenEmbedded and Yocto provide the core and application level software recipes to build a multitude of firmware images for a multitude of devices.

The Docker project [93] performs some form of packaging-time product lines by using overlay filesystems in their images similarly to this approach in order to stack feature filesystems. We use the same concept but for device boot of the basic operating system and additionally preventing activation and decryption of any unauthorized feature layer.

[1]http://www.openembedded.org

Another closely related case of product lines are so-called (feature oriented) software product lines, e.g., [6]. A lot of research has been conducted on software product lines [27, 9, 58, 69] and even their security [15], practical implementations exist and given the inclusion in mainstream build systems, such as autotools [21], they are at the core of modern software engineering. The presented approach, just as the Yocto and OpenEmbedded projects, facilitate these capabilities and use them for realizing firmware product lines.

Other related technologies that are designed with a closely related focus are called feature activation. This term refers to the activation of features on devices (similar to the presented approach), but based on additional fees payed by the device owner. Though much research has been conducted on securing feature activation [111] and many more patents were filed, such as [78, 41], these works have focused on the secure transfer of activation codes; i.e., on providing the evidence for an activated feature into the device. The actual securing of the feature-relevant programs on the device were to the best of the authors knowledge not yet solved. Also note that feature activation is not the focus of the presented approach and would require a rework of the model number deployment process (cf. Section 4.5.2).

Finally, it is known that many firmwares today are already partially self-reconfiguring upon startup based to their model association by activating or deactivating certain software services. However, the security of these features remains yet to be solved, and a plausible approach is presented in this chapter.

### 4.3.2 Overlay Filesystems

Overlay filesystems combine multiple filesystems into one single virtual filesystem and directories with equal paths are merged to one path. A typical application is the combination of read only devices (e.g., a CD) with writable devices, where all changes in the read only filesystem are stored on the writable device. This example use case could be realized under Windows with the Unified Write Filter [88]. For Linux the overlay filesystems AUFS[2] and OverlayFS [16] are available. AUFS is a reimplementation of UnionFS [103] the first available implementation of an overlay filesystem for Linux. UnionFS is also available for Free BSD and Net BSD. AUFS was the first device driver used in Docker [93] to layer Docker images and is very stable. OverlayFS is included in the mainline kernel and is potentially faster than AUFS. Despite this fact, AUFS was using in the prototypical implementation to present the concept since the structuring of the data was simpler, and AUFS was a better match to the requirements for the integration of encrypted SquashFS read only filesystems into the virtual overlay filesystem.

[2]http://aufs.sourceforge.net/

## 4.4 Concept for Runtime Product Lines

The concept for implementing *Secure Runtime Firmware Product Lines* consists of three parts: (1) configuration of the device during production, (2) creation of the firmware image, and (3) booting a model-specific firmware.

### 4.4.1 Device Production

During device production, two steps need to be taken. A key must be deployed to the device, that is used for importing the sealed key blobs, and the model type number must be stored inside the device's TPM. The following TPM commands are used to realize this:

**Algorithm 5** (Production). *// Set model number*

```
  TPM2_NV_DefineSpace(modelIdx, policyWritten, UserRead PolicyWrite)
  // Create policy session
  sess = TPM2_StartAuthSession(PolicySession)
  TPM2_PolicyNVWritten(modelIdx, false, sess)
  TPM2_NV_Write(modelIdx, modelNo, sess)
  // Deploy (import) ImportTargetKey
  ImportKeyBlob := TPM2_Import(ImportTargetKey)
  tmpHandle := TPM2_Load(ImportKeyBlob)
  TPM2_EvictControl(tmpHandle, ImportKeyHandle)
```

#### Model Number

The model number is stored inside the NV-storage of the TPM and can be read by anybody but cannot be altered. The number will subsequently be used to test whether a certain software feature shall be decrypted and activated at boot time for the given device. It is even possible to store the model number as part of the serial number, since the TPM-operations also allow the testing of only parts of a number stored in NV memory.

The purpose of the TPM2_PolicyNVWritten is to disallow subsequent writes by anybody to the model number NV index. Only as long as the NV index has not been written, can it be written. This also enables partial pre-production of products, where for example the board is assembled and fit into a chassis and the TPM is partially pre-provisioned but the model number is not yet set.

### ImportTargetKey

The anchor for the decryption operations during boot is the ImportTargetKey. This key needs to be deployed to all devices of a product line. It is stored persistently inside the TPM for the lifetime of the device. This key is imported into the TPM and stored persistently using the TPM2_EvictControl command. When booting a firmware, the keys used for each of the features are imported underneath this key before being unsealed for actual usage during firmware decryption and activation.

## 4.4.2 Firmware Creation

Two additional steps are required for building a firmware image: the product line elements must be created by means of an overlay filesystem and the filesystems must be encrypted with a sealed TPM key bound to the corresponding model number bit. The process involving the TPM can be realized as follows:

**Algorithm 6** (Firmware Creation). *firmware = createBaseImage()*

```
for fs=overlay_1 to overlay_N do
    tmpfs = createFeatureLayer()
    fs.bitmask = createFeatureBitmask()
    key = TPM2_GenRandom()
    fs.data = encrypt(key, tmpfs)
    policyDigest = sim_PolicyNV(modelIdx, fs.bitmask, BITSET)
    fs.seal = create_Import(importTargetKey, featureKey, policyDigest)
    firmware.add(fs)
end for
release(firmware)
```

### Filesystem Creation

The firmware for Secure Runtime Firmware Product Lines needs to be constructed in a specific way. For this example, a base firmware image is assumed that is common to all models of the given product line. Note though that even different base images can be provisioned, this requires more space in the resulting firmware image. For the sake of explanation, the construction of a firmware image for two models ModelA and ModelB with two mutually exclusive features FeatureA and FeatureB is considered.

A base image is constructed as file image on the build PC. For a Linux system, this would include things such as libc, base-tools, init-system, etc. Next, a second (empty) image file is created and mounted as overlay to the base image. Then the software for FeatureA is installed into the overlay filesystem. This step can also include the rewriting or even the deletion of files. Some overlay filesystems represent deletion as entries to the "0-inode" for example. When unmounting the filesystem, the differences from base image to ModelA are stored inside the FeatureA filesystem. Now, a second file image can be created for FeatureB and the process of overlay mounting and installation can be repeated.

The approach can be further generalized by, e.g., mounting the images for base, FeatureA, and FeatureC versus base, FeatureB, and FeatureC if models differ for example in some intermediate layer, but not in the highest layer.

### Filesystem Encryption

For each of the feature filesystem layers, a bitmask is created regarding which bits inside the model number represent the activation of the corresponding feature. For example, any model number with bit 0 set (i.e., uneven numbers) will include the filesystem for FeatureA. With this information, a corresponding TPM2_PolicyNV statement can be constructed that represents the test for this bit inside the model number NV index. It can also test for multiple bits or even the complete model number. This depends on the architecture for assignment of features to model numbers. The result of the policy statement creation is a policyDigest, i.e., a hash value that represents this policy.

Then a symmetric key is created that is used for encrypting the feature filesystem image. This key is then embedded within an import blob for the target TPM. This import blob is of type keyedHash, which means that it can be unsealed on the target TPM. It includes the key for the filesystem and also the policyDigest that restricts the unsealing of the key to those devices that have a model number corresponding to the policy's requirements. Finally, this import blob is encrypted using the ImportTargetKey as decryption key.

The bitmask, the encrypted key seal blob for TPM import, and the encrypted filesystem are then provided with the firmware image as a package *FeatureFS-X Data BLOB*. Note that the order of mounting the overlay images plays an important role, especially if file alterations or removals exist. This can be represented by naming of the files that contain the feature packages (if they are stored as files inside the firmware image) or by the order in which they are stored, if e.g., a partition table is used, where each package is contained inside its own partition.

## 4.4.3 Booting a Model-specific Firmware

During firmware boot, the specific firmware for the given model of the product line is unlocked and mounted. This process uses the following algorithm:

**Algorithm 7** (Booting). *current = mount(basefs)*

```
modelNo = TPM2_NV_Read(modelIdx)
for fs=overlay_1 to overlay_N do
    if fs.bitmask == modelNo & fs.bitmask then
        seal = TPM2_Import(importKeyHandle, fs.seal)
        pSess = TPM2_StartAuthSession(PolicySession)
        TPM2_PolicyNV(modelIdx, fs.bitmask, BITSET, pSess)
        key = TPM2_Unseal(seal, pSess)
        overlay = decrypt(key, fs.data)
        current = overmount(current, overlay)
    end if
end for
current = overmount(current, localdatafs, localdata-directory)
current = overmount(current, tmpfs, runtime-directory)
chroot_and_boot(current)
```

### Basic Preparations

During boot, the first step is to perform some basic operations. During this step, the basefs image is mounted and the model number is read from the TPM. This reading of the model number is not restricted in any way and can be performed by any running software. Only writing is restricted to the vendor's production step.

### Feature Loop

The loader will loop over all features that are provided within the unified firmware image and test for each of these, whether they are activated for the given model number. Those that are not active will be skipped. For all activated features, the loader will import the sealed feature key into the TPM under the ImportTargetKey. Then it will provide a policy session that proves to the TPM that the policy for this sealed feature key is fulfilled by the model number stored inside the TPM. Then it will request the unsealing of the feature key from the TPM, based on the policy session that proves the correctness of this attempt. With the unsealed feature key, the loader can decrypt the feature filesystem and mount it on top over the currently mounted system – either over the basefs or over the stack of basefs and previously mounted feature overlays.

#### Local Configuration and Runtime Data

Before switching into the final stack of overlay filesystems, the loader mounts two final filesystems. The runtime overlay is a temporary filesystem held only in RAM for the current boot cycle. This is necessary for the firmware in order to create sockets for local IPC connections, storing process identifiers, or to create temporary files for locking to schedule access to certain resources. The configuration overlay is an additional filesystem that contains local configuration data of the device. The reason for performing an overlay mount for this filesystem instead of a regular mount is that the feature filesystems can provide default configurations within their images. Thus, only changed configuration files are actually stored on local storage.

## 4.5 Discussion of Runtime Product Lines

### 4.5.1 Security

To upgrade their device to a higher-level model, an user / attacker can try the following attacks.

#### Attacking the TPM

The attacker can try to read out the feature keys of the TPM. The secrecy of the feature keys depends directly on the secrecy of the ImportTargetKey stored inside the TPM. An extraction of this key is unlikely if a hardware TPM certified by Common Criteria (usually using EAL4+) targeting the TCG's defined Protection Profile for TPMs is used. To further mitigate the impact of a successful key extraction from the TPM, the vendor could also rotate the import key with every production batch. This will, however, require the vendor to provide import-blobs of the sealed feature keys for all of these rotated import keys. In order to compensate for the latter, the vendor can instead choose to deploy an additional intermediate key together with the firmware blobs. In this approach, every production batch would have its own import target key. Each firmware version would then include a specific intermediate import key that is prepared for importing under all import target keys. The sealed feature keys would then be encrypted for import under the intermediate keys. Given a realization with $n$ import target keys, i.e., production batches, and $m$ sealed feature keys, this would mean that each firmware blob includes one (unique) intermediate key that is packaged $n$ times for the different import target keys and $m$ sealed feature keys that are encrypted for the intermediate import key. This is a significant simplification compared to the $n \cdot m$ sealed feature key that would be required without an intermediate import key.

#### Attacking the unsealed feature keys

To decrypt and mount a feature filesystem, the required feature key is decrypted by the TPM and transferred to the host CPU. An attacker could sniff on the transmission. This threat can be mitigated either by physically securing the bus between TPM and main CPU or by using the TPM protocol's built-in encryption capabilities. For this, the boot-code of the device could include the public portion of the import key and use this to encrypt a salt value during session establishment. Attacks against the host CPU are not addressed by the presented approach. This includes attacks against the OS kernel or even cold boot attacks. To address the latter, mechanisms such as full memory encryption could be applied [75].

#### Attacking a feature-rich model

An attacker could attack a feature-rich model and try to extract an unencrypted firmware image and inject it into a low-feature device. This attack would require access to the raw RAM during runtime via a software exploit. To cope with potentially unknown vulnerabilities, appropriate mechanisms for secure code update should be applied [48].

#### Manipulation of the model number

An attacker could try to change the model number to a number of a device with more features. The integrity of the model number depends on the inability of the user to write to the model number NV index. This needs to be ensured by disallowing TPM2_-UndefineSpace and TPM2_UndefineSpaceSpecial with Platform-Authorization, which is supposed to be under vendor-control anyways. In order to mitigate attack potential even further, the vendor can set TPM2_NV_WriteLock on model number explicitly on each boot.

### 4.5.2 Extensions

#### Refurbishment of Devices

In many environments with device product lines, there exists the necessity to refurbish devices that are e.g. produced but never delivered. This can occur due to canceling of orders or because a certain hardware feature is defective that is only required for one of the product lines of the device. In such scenarios, the vendor will refurbish the device to a new model by exchanging the chassis or a model number sticker on the chassis and reconfigure the firmware on the device to the new device type.

In order to support this process with the presented scheme, the policy for the model number NV index on the device can be set to a policy that allows the vendor (and only the vendor) to perform write operations on this index. Such a case can be achieved using a TPM2_PolicySigned. This policy requires an external entity to sign a challenge from the TPM in order to perform a certain operation.

#### Model Number Attestation

Many devices nowadays are provided with the inclusion of additional services, such as cloud integration. This may, however, be a premium feature that is only activated for premium devices. The presented scheme of storing the model number inside the TPM supports such scenarios using the TPM2_NV_Certify command. Using this command, the vendor's cloud service can send a challenge to the device and the TPM will certify that the device has a given model number.

#### Model Numbering Schemes

The presented approach uses a very simplistic scheme for activation of feature filesystem for a given model number by querying whether certain bits in the model number are set. In addition, it is possible to extend these schemes further using a combination of AND and OR in the policy statements. For example, it is possible to require a (set of) bits to be zero using the TPM2_PolicyNV with the operation TPM_EO_BITCLEAR. These policies can then be extended using the TPM2_PolicyOR to enable a certain feature filesystem for other bit combinations as well.

## 4.6 Implementation of Runtime Product Lines

The concept was implemented on an Intel NUC D34010WYK equipped with a TPM 2.0 implementation running Ubuntu 16.04 with kernel version 4.4. An Apache and a Samba server were used as product line element (PLE) examples. For accessing the TPM, a TPM Software Stack (TSS) implementation of the TPM 2.0 System API [131, 122] was used together with accompanying `bash` tools for rapid prototyping.

In the device production phase, the storage root key (SRK) of the TPM is generated, and the model number is written to the NV-Storage.

During firmware creation the overlay filesystems is constructed. For this purpose, AUFS was used.

For every product line, an encrypted SquashFS filesystem is created for overlay mounting. This filesystem is encrypted using dm-crypt [17] container files. Dm-crypt is a Linux module used for transparent disk encryption. The AES encryption key of the encrypted SquashFS filesystem is encrypted with a private asymmetric key. The created TPM object is protected by a policy testing the flag corresponding to the PLE in the TPM's NV model number.

For booting a model-specific firmware, the mounting of these filesystems is integrated into the boot process by using initramfs which makes necessary preparations before switching to the systemd init process.

A shell script for the creation of the overlay filesystem is added to the bottom stage of initramfs. Scripts in this stage are executed before procfs and sysfs are moved to the real rootfs and execution is turned over to the init binary of the rootfs.

The command `mount -t aufs -o br=/r/tmps=rw:/r/02_apache:/r/ none /r/chr/` mounts the overlay filesystem with a temporary top filesystem `/r/tmps`, the mounted Apache Squash filesystem `/r/02_apache`, and the Ubuntu base system root directory `/r` to the mount point `/r/chr`. This overlay is finally mounted over the root filesystem `/r/tmps` and the init process is started to boot the system. Since the top filesystem of the current overlay system is a temporary filesystem, all file changes of the running system will be temporary. In this example, only the Apache SquashFS is mounted because the flag for enabling the Samba container was not set. Only the key for the encrypted Apache SquashFS file system could be unsealed because the corresponding flag was set. The booted system then produces the same state as during the firmware creation after the installation of the apache packages. This also includes correct configurations, since no overrides are provided via a local configuration overlay. A first performance analysis showed only a negligible delay in the boot process introduced by this concept for runtime firmware product lines.

## 4.7 Evaluation of Runtime Product Lines

The approach for secure runtime firmware product lines allows unified firmware images to be provisioned to a whole series of products while preventing unauthorized activation of features that belong to a different model instance. Using the features of TPM 2.0 Enhanced Authorization, Sealing, and Importing this scheme can be implemented by using standard hardware. The TCB for the presented approach now only consists of the TPM itself as depicted in Figure 4.4. Firstly, the storage was completely removed from the TCB. Regarding the disallowed activation of features the TCB only consists of the TPM

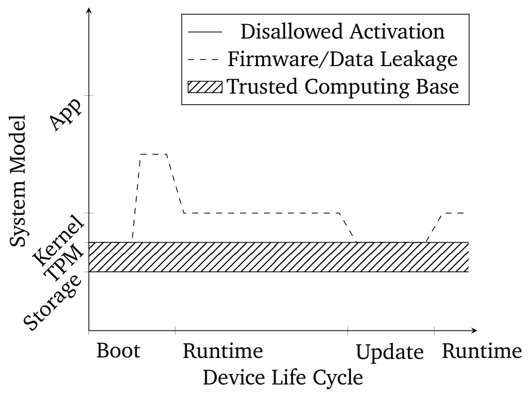


Figure 4.4: Trusted Computing Base for Secure Runtime Product Lines

itself. Regarding the disallowed extraction of feature code, the firmware's OS kernel, that does the bulk encryption, is part of the TCB. The userland software, that contains has the large attack surface however is not part of the TCB.

The implementation shows the feasibility of this approach and its integration with a Linux-based system. Future work includes the extension of the presented approach to include unified images for hardware product lines, similar to the OpenEmbedded's Board Support Packages and the tight integration with the build process of unified firmware images as well as a thorough performance evaluation using different embedded platforms.

# 5 Securing Payment Credentials in Plug and Charge

As mentioned in the introduction, the most common usage of HSMs / TPMs is the protection of credentials used for authentication or non-repudiation. This integration can trivially be implemented using off-the-shelf technologies such as tpm2-pkcs11[1] or tpm2-tss-engine[2]. Whilst the usage of said credential does not pose a challenge, their deployment to devices in the field does. Furthermore, then integration with existing protocols and standards can pose a major challenge for these scenarios.

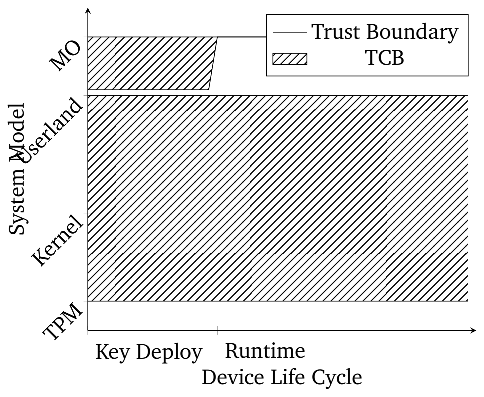
This chapter investigates the deployment process of payment credential for Plug and Charge scenarios following ISO 15118 called TrustEV and HIP. The contributions in this chapter build upon my contributions to [56, 54, 53] which include major parts of the general approach and detail concept.

## 5.1 Background on Plug and Charge

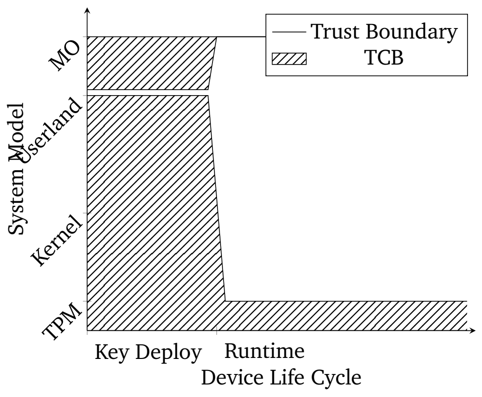
The automotive industry is shifting away from gasoline powered engines to electric engines. Besides a common reduction in vehicle range, the biggest difference between those two approaches comes with refueling. A gasoline powered vehicle can be refueled by the driver within a matter of minutes – usually at a gas station. Electric cars require a longer refueling process for their batteries in the range of hours. Thus refueling usually does not happen at a gas station but distributed over parking garages, public parking, supermarket parking lots, at work and at home. Other than a gas station these distributed charging solutions require different payment methods than a human cashier.

The first generation of charging stations used a wide variety of RFID payment cards and mobile apps, requiring vehicle drivers to have a large collection of contracts and credentials and requiring manual interaction. In order to unify and simplify this process (and as such to increase acceptance) the standards for Plug and Charge (PnC) ISO 15118

[1]`https://github.com/tpm2-software/tpm2-pkcs11`
[2]`https://github.com/tpm2-software/tpm2-tss-engine`



(a) HSM-less



(b) Trivial HSM Integration

Figure 5.1: Trusted Computing Base for basic PnC

[79, 80] have been developed. They define the communication protocols between the Charge Point (CP) and the Electric Vehicle (EV) either via Power-Line Communication (PLC) in case of wired charging (by hand or with a charging robot [136]) or via WiFi in case of inductive charging.

ISO 15118 is based on public key authentication and signing using X.509 certificates and a dedicated Public Key Infrastructure ([80], Annex E). Two major credentials are defined. Firstly, the *OEM Provisioning Credential* is installed in each vehicle during production by the vehicle OEM. This credential serves a a lifelong identity for the specific vehicle. Secondly, the *Contract Credential* is used for actual payment of charging processes. The contract credential is provided by a Mobility Operator (MO) when a driver registers its Provisioning Credential with said MO – by signing a charging service contract. The Contract Credential is then downloaded automatically during the first charging of the vehicle.

The process for Contract Credential establishment currently expects the MO to generate a public private key pair and corresponding certificate. The private key is encrypted with the provisioning key for protection during the aforementioned downloading. On the Electric Vehicle Charge Controller (EVCC) the contract private key is decrypted and subsequently used for billing purposes. This process opens up two weak points. Firstly, even if the private keys for provisioning and contract keys are protected by a HSM/TPM, the data format of ISO 15118 for the encrypted private key requires at least a temporary availability in the EVCC's main application CPU. Secondly, the private keys are (at least temporarily) known by the MO backend. A successful attack on the MO (cf. the Sony hack from 2014 [144]) would give an attacker access to these keys. The leak of these credentials poses the threat of impersonation and privacy infringement on the driver. Furthermore, a successful attack on such systems will reduce trust in and reliance of the public on this essential technology for the energy revolution.

Figure 5.1 illustrates the corresponding TCBs. The lifecycle is divided in only two phases, key deployment and the runtime with key usage. The standard implementation from Figure 5.1a contains the complete device as part of the TCB. Further, since the backend generates the keys to be imported are also part of the TCB for the deployment process step. If a trivial HSM integration was attempted – similar to Section 2.2 and Figure 2.4b – the resulting TCB is illustrated in Figure 5.1b. During key deployment, the whole device and backend are part of the TCB, but afterwards only the TPM is in charge of enforcing the security policy.

ISO 15118 security concept relies on the implicit assumption that such leaks do not happen and that the EVCC and the MO are both protected [11]. The standard however gives no advice on how to handle those keys. Quite the contrary, the standard requires the key generation in the backend, whilst the generation and storage of keys in an HSM would be much more secure; following the recommendations of NIST [12].

This chapter describes TrustEV and HIP, two protocol extension for ISO 15118, where keys are protected by a TPM within the vehicle. It considers the secure generation, storage, provisioning/enrollment, use and revocation of OEM provisioning and contract credentials, and thus, address the shortcomings of the current standard regarding credential management.

With these extensions, private keys never leave the vehicle's TPM and with HIP only public keys need to be stored in the backend systems of the Original Equipment Manufacturer (OEM) and Mobility Operator (MO). Thus, even a successful attack on these backend systems cannot compromise private keys. The protocol extensions are backwards compatible, i.e., even if intermediary systems such as the Charge Point (CP) do not support the extension, an Electric Vehicle (EV) and MO can still use it.

### 5.1.1 PnC Charging Infrastructure

The infrastructure required for EV charging requires the connection of energy production, energy delivery, the CP networks, the EVs and various service providers. Given this complexity a variety of approaches have been defined that enable management and billing in such scenarios [104]. A unification of the concept for Plug and Charge was introduced by ISO 15118 [79, 80]. These standards specify the communication protocols and interfaces between the EV and the CP. They include automated identification and authorization of the EV and automated billing of the EV driver. ISO 15118 defines message and sequence requirements, data models, and XML/EXI-based data representation format. For securing the PnC process the use of TLS, encryption and digital signatures are introduced. The second edition ISO 15118-20 is currently under development.

The standard defines a set of actors, including mainly the Electric Vehicle Charge Controller (EVCC) Electronic Control Unit (ECU) of the Electric Vehicle (EV) and the Supply Equipment Communication Controller (SECC) of the CP. The EVCC and the SECC handle the communication, authentication and handling of the charging sessions. Furthermore the SECC communicates with backend systems, including the Charge Point Operator (CPO) that manages the CP, the Contract Clearing House providing eRoaming services and the Certificate Provisioning Service and MO that offer credential services. Finally, the OEM provides the Provisioning Credential (PC) to the EVCC.

While ISO 15118 defines the communication protocols between EVCC and SECC, the communication between the SECC and the backend systems are not defined. Presumably Open Charge Point Protocol (OCPP) [3, 4, 5] is used to transfer the necessary data to the CPO who in turn sends the necessary data to the MO for billing the driver or owner. The latter might be a direct communication or use a Contract Clearing House eRoaming service using any of the protocols OCHP 1.4 [96], OICP 2.2 [97], OCHPdirect 0.2 [94], or OCPI 2.2 [95].

### 5.1.2 Related Work on Plug and Charge security

Given the increased digitalization and connectivity of modern vehicles, automotive security is gaining a lot of focus and investment. The risks associated with this shift were very famously demonstrated in the "Jeep hack" in 2015 [59]. But other research analyzing the attackability of e.g. wireless tire pressure sensors [106], diagnostics ports [91], or remote attacks [25, 90] demonstrate the need for ongoing and further research in defensive technologies. The effects of successful attacks were discussed in [83]. They all have in common that the trust and willingness to rely on these technologies by vehicle drivers are at stake.

In order to secure confidential data and cryptographic credentials, a set of security extensions to automotive SoCs have been introduced. The first being the SHE module [115] by the HIS consortium. Then the EVITA project defined a set of three possible security extensions for SoCs [138] and the TCG released a profile for automotive TPMs [121]. The Car 2 Car Communication Consortium released a definition for HSMs for C2X communication [22]. In order to secure on-board communication, the EVITA project introduced CAN bus authentication [113] and the AUTOSAR consortium defined SecOC [10]. For Automotive Ethernet systems, TLS was proposed [143]. The security of these protocols and the defined security extensions rely upon the implementation of said extensions. Typically, they are implemented as SoC HSMs that provide no certification for the assurance level they provide.

The lack of certification of typical automotive security extensions poses a problem for billing applications. Consequentially, they may pose a problem for PnC solutions as well. Even if no regulation exists yet (compared to e.g. the Credit Card industry), user trust should not be gambled with. An alternative exists with current TPM chips that provide Common Criteria Certification of EAL4+ and even qualification according to AEC-Q100.

Another reason for using TPMs is the ability for fine-grained access control provided by TPMs, that are neither provided by SHE or EVITA modules. Also, the proposal of using TPMs in automotive applications is not new. [141] proposes the use of TPMs for integrity verification in V2X communication scenarios. The anonymization of users via Direct Anonymous Attestation using TPMs in PnC scenarios was discussed in [145, 142] and privacy-aware architectures for vehicle-to-grid communication is discussed in [57] and [109]. Using TPMs to secure value-added-services in EV charging was introduced in [20].

## 5.2 System and Threat Model for Plug and Charge

In the following the actors and protocols of the system and the corresponding threat model are described.

### 5.2.1 PnC System Model

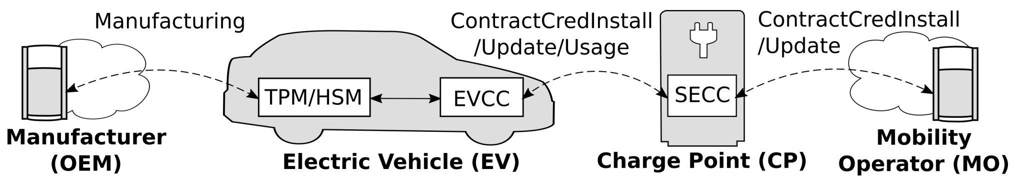


Figure 5.2: System Model [56]

The system model is illustrated in Figure 5.2. It consist of the following actors:

- *Manufacturer / OEM*: The original manufacturer of the EV. The OEM will install the *OEM Provisioning credential (PC)* in the EVCC during production.
- *EV*: The electric vehicle.
- *EVCC*: The electric vehicle charge controller. The EVCC performs all PnC communications on behalf of the EV or the driver.

- *TPM/HSM*: The security module that is part of the EVCC ECU. It protects the PC private key and the CC private key from unauthorized access and performs the signature and decryption operations using these keys.
- *CP*: The charge point.
- *SECC*: The Supply Equipment Communication Controller. The SECC communicates with the EVCC on the one hand and to the CPO and hence the MO on the other hand. It also controls access to the charging current of the CP.
- *MO*: The Mobility Operator. The mobility operator issues the Contract Credentials and bills the customer based on the charging information gathered by the CPO.

The system includes the following credentials:

- *OEM Provisioning Credential (PC)*: The PC serves as lifelong identity for the EV, stored inside the EVCC. It consist of a keypair $\{PC_{pub}, PC_{pr}\}$ and an X.509 certificate $Cert_{PC}$ containing the Provisioning Certificate Identifier and $PC_{pub}$.
- *Contract Credential (CC)*: The contract credential enables the authentication of the billing processes for the charging processes. I consists of a key pair $\{CC_{pub}, CC_{pr}\}$ and an X.509 certificate $Cert_{CC}$ including a unique E-Mobility Account Identifier (EMAID) linked to a charging contract.
- *Storage Root Key (SRK)*: This is a new credential introduced with TrustEV / HIP that serves as decryption target key for the $CC_{pr}$ / $Cert_{CC}$ [125].

The system contains the following steps:

- *Manufacturing*: The assembly of the EV by the OEM, including the PC installation.
- *CertInstallReq*: The certificate install request sent by the EVCC via the SECC and CPO to the MO. It requests a $Cert_{CC}$ downloading.
- *CertInstallResp*: The response to a CertInstallReq containing the Contract Credential.
- *Charging*: The regular charging and billing processes remain unaltered by the TrustEV / HIP solutions. The EVCC uses the $Cert_{CC}$ and $CC_{pr}$ – i.e. the TPM – to authenticate to the SECC and CPO and to sign off the billing information. This part remains unaltered by TrustEV / HIP compared to a naive HSM solution

### 5.2.2 PnC Security Policy and Threat Model

The basic security policy to be implemented by the TrustEV / HIP approaches is the protection of the private keys $PC_{pr}$ and $CC_{pr}$. With those keys secured, an attacker is only able to perform attacks via the application code of an EVCC by tunneling all traffic to the target and hav data be signed continuously during a charging session. However, the attacker is not able to perform impersonation attacks without live access to the victim and a software update to the victims EVCC allows a mitigation of this attack path without reissuing of credentials (cf Figure 2.4).

The threat model assumes an adversary that is able to gain physical or remote control of an EVCC module of a target's EV. As such he/she can read all data at rest and all data in RAM of the EVCC application CPU. The adversary cannot gain access to encrypted data unless he/she has knowledge of the decryption key. The adversary cannot gain access to the TPM-internal RAM and storage.

In case of the HIP approach, the adversary may even be able to gain access to the MO's backend system and retrieve customer credentials from there; not however the MO's own PKI private key.

## 5.3 TrustEV: A concept for Secure ISO 15118 Key Distribution

The general idea of TrustEV is the integration of TPM functionalities into the concept and flow of ISO 15118 with only minimal changes to the protocol. These changes only change encrypted data such that only the EVCC, the OEM and the MO require to know this approach. An intermediate SECC or CPO are not required to be TrustEV aware.

The main benefit will be the enabling of the TPM import features during the ISO 15118 Contract Credential downloading. As such the EVCC's application CPU is removed from the TCB for the $CC_{pr}$. Additionally, the usage of the $CC_{pr}$ can be restricted with the TPM's Enhanced Authorization policies, for instance binding it to measured boot states.

### 5.3.1 TrustEV architecture

#### TrustEV components

The *EVCC* serves the communication with the CP, CPO and MO in order to authenticate the charging process and billing information. For TrustEV it is extended with a hardware *TPM 2.0* that shall secure all credentials of the EV.

The OEM initialized the EVCC and its TPM with the application software and necessary initial keys. These keys are the so-called Storage Root Key (SRK) that is used as storage decryption key for protecting all other private keys and also used as target for the import of credentials. The OEM further establishes the OEM provisioning key pair in the TPM, which is generated inside the TPM itself for the most secure version. The OEM also deploys the provisioning certificate to the EVCC including the TrustEV certificate extension *TrustEVExt*.

An MO capable of the TrustEV approach evaluates the PC certificate for the *TrustEVExt* X.509 extension. If such an extension is found the MO will encrypt the Contract Credential

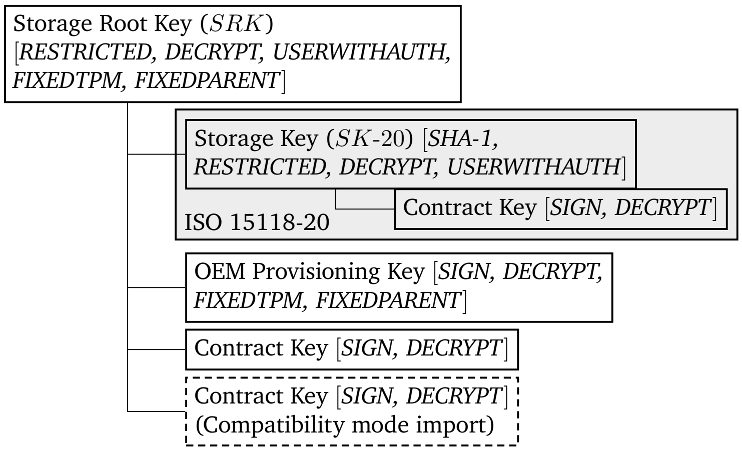

Storage Root Key ($SRK$)
[*RESTRICTED, DECRYPT, USERWITHAUTH, FIXEDTPM, FIXEDPARENT*]
Storage Key ($SK$-20) [*SHA-1, RESTRICTED, DECRYPT, USERWITHAUTH*]
Contract Key [*SIGN, DECRYPT*]
ISO 15118-20
OEM Provisioning Key [*SIGN, DECRYPT, FIXEDTPM, FIXEDPARENT*]
Contract Key [*SIGN, DECRYPT*]
Contract Key [*SIGN, DECRYPT*]
(Compatibility mode import)

Figure 5.3: TrustEV Key Hierarchy [56]

private key in a TPM import blob instead of the regular plain encryption as defined by ISO 15118. If no extension is found then the MO will behave in an ISO 15118 compliant regular way; thus preserving backwards compatibility. A non TrustEV capable MO will ignore the TrustEVExt since it is non-critical and perform a regular ISO 15118 process. The latter compatibility modes will of course not benefit from the added security of TrustEV.

As such the TrustEV approach requires the OEM and EVCC as well as the MO to be aware of the extension. The TrustEV will remain transparent to all other components. Especially the unencrypted parts of ISO 15118 messages remain unaltered.

### TPM Key Hierarchy

The hierarchy of keys inside the TPM is depicted in Figure 5.3. The top most key is the Storage Root Key (SRK) $\{SRK_{pub}, SRK_{pr}\}$. It is an asymmetric restricted decryption key that is used for storage operations. It is derived from seed inside the TPM and a device secret. The SRK is used to en-/decrypt other key material for storage outside of the TPM as well as for decrypting keys to be imported into the TPM.

The OEM Provisioning Key is stored under the SRK. It serves for signing CertInstallReq messages as well as decrypting Contract private keys in original ISO 15118 operation mode. The SK-20 is another storage key with a different name algorithm, since ISO 15118-20 will offer more space for transport of the encrypted key.

| OEM Provisioning Certificate | | |
|---|---|---|
| Version: | | X.509v3 (0x2) |
| Serial Number: | | 12345 (0x3039) |
| Signature Algorithm: | | ecdsa-with-SHA256 |
| Issuer: | | CN=OEMSubCA2, O=ISO, C=US |
| Validity | Not Before: | May 7 08:40:32 2019 GMT |
| | Not After: | May 6 08:40:32 2021 GMT |
| Subject: | | CN=OEMProvCert, O=ISO, DC=OEM |
| Subject Public Key Info | Public Key: | OCTET STRING |
| | Algorithm: | id-ecPublicKey |
| | Parameters: | namedCurve secp256r1 |
| X509v3 Extensions | Basic Constraints:[c] | CA:FALSE |
| | Key Usage:[c] | Digital Signature, Key Agreement |
| | Subject Key Identifier:[nc] | keyIdentifier (SHA-1) |
| | **TrustEV Extension:**[nc] | **TPM 2.0 EC Storage Root Key ($SRK_{pub}$) 512 bit OCTET STRING** |
| | | **TPM 2.0 SHA256 Policy Digest 256 bit OCTET STRING** |
| | | OPTIONAL **TPM 2.0 EC Storage Key ($SK\text{-}20_{pub}$) 512 bit OCTET STRING** |
| Signature | Algorithm: | ecdsa-with-SHA256 |
| | Value: | OCTET STRING |

Figure 5.4: Provisioning Certificate with TrustEV Extension [56]

### TrustEV Certificate Extension

The TrustEVExt X.509 extension defines a non-critical extension to the OEM Provisioning Certificate. It is non-critical in order to allow backwards compatibility with TrustEVExt unaware MOs. Figure 5.4 illustrates this extension.

The extension contains all necessary information for the MO to construct the import blob for the Contract private key $CC_{pr}$. The Storage Root Key $SRK_{pub}$ is used for encrypting the blob. The Policy Digest represents the policy that shall be associated with the $CC_{pr}$ according to the OEM. (Note that the MO only need to know the digest of the policy. The EVCC knows the actual policy represented by said hash. See Section 2.4.3 for more information.)

In case of the original ISO 15118 specification this same extension needs to be added to the contract certificate as well, since it allows Contract Credentials to be used for CertInstallReq calls. Starting with ISO 15118-20 only OEM Provisioning Certificates are used for CertInstallReq calls and Contract Credential Certificates can be used unaltered.

## 5.3.2 TrustEV Process and Protocols

This section describes the processes and protocols of TrustEV regarding manufacturing, contract certificate generation, installation and upddate, and credential usage.

### EV / EVCC Production

During production the OEM initializes the TPM by generating the SRK and reading out the public key $SRK_{pub}$ and the Provisioning Credential key pair $\{PC_{pub}; PC_{pr}\}$ under the SRK. The PC keypair can be associated with an additional TPM policy such as *TPM2_PolicyPCR* or *TPM2_PolicyAuthorize*. The $SRK_{pub}$ and the digest of the TPM policy are both included in the TrustEV extension for the $Cert_{PC}$ certificate generation. The $Cert_{PC}$ is signed by the OEM CA that is part of the ISO 15118 PKI.

### Contract Credentials Generation

The MO validates the correctness of the $Cert_{PC}$ and extracts all the necessary information. The MO generates a key pair $\{CC_{pub}; CC_{pr}\}$ for the user. It then creates the corresponding X.509 *Contract Certificate* $Cert_{CC}$ that contains the $CC_{pub}$, EMAID, and (in case of ISO 15118 and not ISO 15118-20) a copy of the TrustEV certificate extension field from the $Cert_{PC}$.

Next, the $CC_{pr}$ needs to be encrypted such that it can be imported into the TPM. The $CC_{pr}$ is encoded in a TPM private key structure that includes the algorithm ID (ECC), an

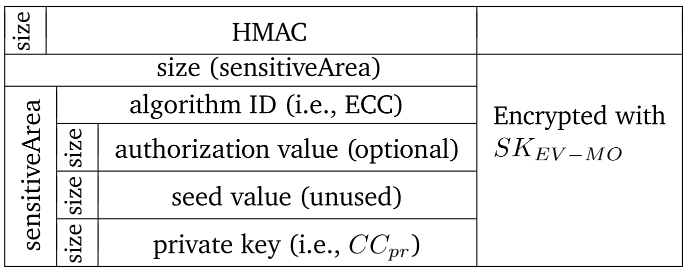


Figure 5.5: Encrypted Contract Key for Direct Import [56]

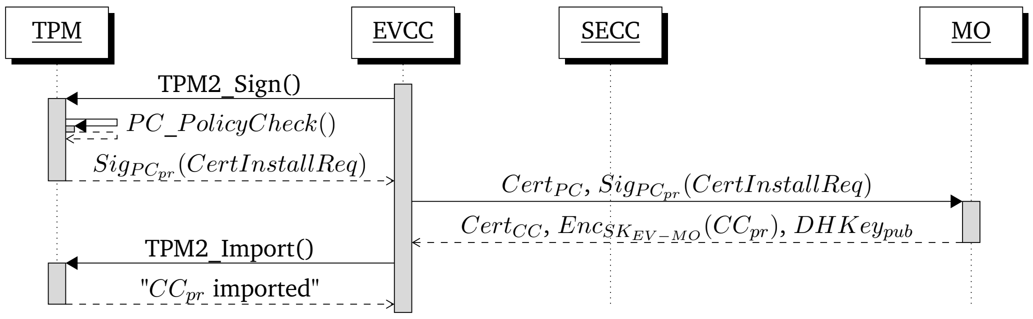

TPM
EVCC
SECC
MO
TPM2_Sign()
$PC\_PolicyCheck()$
$Sig_{PC_{pr}}(CertInstallReq)$
$Cert_{PC}, Sig_{PC_{pr}}(CertInstallReq)$
$Cert_{CC}, Enc_{SK_{EV-MO}}(CC_{pr}), DHKey_{pub}$
TPM2_Import()
"$CC_{pr}$ imported"

Figure 5.6: Installation of Contract Credentials [56]

empty authorization value, a seed value (that is not used for this type of key) and the actual private key $CC_{pr}$. This sensitive area is encrypted using a symmetric key $SK_{EV-MO}$. This key is derived from an new (ephemeral) keypair $DHKey_{pr}$ and the $SRK_{pub}$. Finally, an HMAC is prepended. The resulting data structure is depicted in Figure 5.5 and follows the scheme for TPM key duplication [127] Chapter 23.3.2.3 “Duplication Process with Outer Wrapper and No Inner Wrapper”.

This data structure is used instead of the regular ISO 15118 encrypted private key. Since it is opaque to all intermediate nodes besides the EVCC, the protocol messages do not need to be altered.

### Installation and Update of Contract Credentials

The communication between the EV’s EVCC and the CP’s SECC is always started with a TLS channel establishment. During this, the SECC authenticates itself (as the TLS server). All subsequent communication is sent through this one-side authenticated channel. The following will ignore the TLS channel and only describe the messages sent over this channel, since it does not contribute to the security of the certificate installation process.

The sequence of messages are depicted in Figure 5.6. The EVCC creates a CertInstallReq message targeted at the MO. It calls the TPM to sign this message using the $PC_{pr}$. In order to have the TPM perform this operation the policy associated with the $PC_{pr}$ has to be fulfilled, e.g. the integrity of the EVCC firmware. The signed CertInstallReq together with the $Cert_{PC}$ are sent via the SECC and CPO to the MO.

The MO generates a Contract Credential key pair with an encrypted private key as described in the previous section. Further the MO creates a Contract Credential certificate and sends all these back to the EVCC.

The EVCC then imports the $CC_{pr}$ into its TPM and stores the resulting keyblob alongside the certificate. Then the Contract Credentials are ready to be used by the EVCC.

In case of an update instead of installation the process is exactly the same for ISO 15118-20. For the original ISO 15118, the EVCC can use the previous contract credential and certificate for the CertInstallReq in place of the Provisioning Credential.

#### Legacy Installation and Update of Contract Credentials

In case an MO does not support TrustEV, the non-critical certificate extension included in the received $Cert_{PC}$ (or $Cert_{CC}$ in case of an update) is ignored and the generation of Contract Credentials stays unchanged from ISO 15118. The resulting encrypted private key is decrypted by the EVCC's TPM and returned back to the EVCC. Then it can be imported into the TPM as plain private key. As such, the private key is revealed to the EVCC RAM during the installation process.

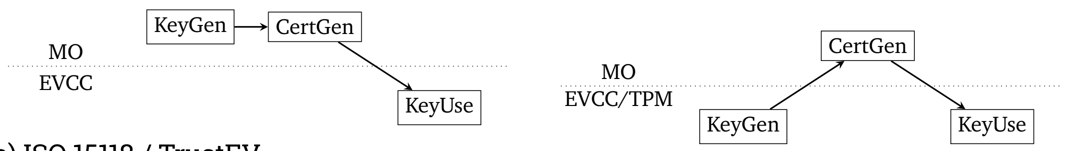
If an EVCC does not support the TrustEV approach, it will send a regular CertInstallReq to the MO. The MO can detect such an EVCC since the TrustEVExt certificate extensions are not present in the $Cert_{PC}$. It can then proceed with the usual ISO 15118 process.

#### Contract Credentials Usage

In order to use the Contract Credentials the EVCC sends the data to be signed to the TPM. It further needs to satisfy the E/A policy associated with the $CC_{pr}$. The returned signature can then be used in the ISO 15118 message as usual.

## 5.4 HIP: A concept for secure ISO 15118 Plug and Charge

An analysis of TrustEV shows that even though the security is greatly enhanced compared to the usual ISO 15118 approach, one additional requirement is not fulfilled. That



(a) ISO 15118 / TrustEV

(b) Protocol Extension HIP

Figure 5.7: Contract Credential Provisioning

requirement is the confidentiality of the private key from the MO, thus the removal of the MO from the TCB. Hence, a successful attack on the MO backend would render the security of many private keys void.

Thus another extension to ISO 15118 is developed and explained in this section. This new extension HIP “HSM-based Identities for Plug-and-Charge” require minor modifications to the PnC protocol but for the benefit of allowing complete confidentiality of the private keys, even from the MO.

### 5.4.1 General Idea of HIP

The idea of HIP as opposed to regular ISO 15118 and TrustEV is to generate all key pairs inside the TPM such that the private keys are never known by any other entity. This shall be especially true for the Contract Credentials. Figure 5.7 illustrates this basic difference. As a consequence, the HIP-enabled approach is not compatible with ISO 15118 anymore, but requires a revision to ISO 15118.

The creation of SRK and PC are the same as with TrustEV. The SRK now does not serve as import key anymore but as activation key for the Contract Credential certificate $Cert_{CC}$. The EVCC uses its TPM to generate a Contract Credential key par and sends a CertInstallReq containing the public key $CC_{pub}$ to the MO. The MO then generates a Certificate $Cert_{CC}$ and encrypts it in such a way that only a TPM that contains the SRK as well as the $CC_{pr}$ will decrypt it. This so-called credential activation process can then ensure that the requesting EVCC TPM is authenticated a posterior once it presents its $Cert_{CC}$.

Of course, HIP also supports the abilities of TrustEV to associate TPM Enhanced Authorization policies with the PC as well as CC private keys. Such policies could reference integrity values for the EVCC firmware or require authorization of the EVCC against the TPM.

The changes to ISO 15118 are the presentation of the $CC_{pub}$ in the CertInstallReq and the sending of an encrypted certificate instead of encrypted private key in the CertinstallResp.

Also the PC certificate extension (similar to TrustEV) is needed. Thus, the EVCC, OEM and MO need to be aware of the HIP approach and the SECC and CPO need to forward the altered messages.

### 5.4.2 Definition of Components and Processes

#### TPM 2.0 Keys

In the HIP approach, all keys used for EV authentication are generated inside the EVCC's TPM. Their attributes are listed in Table 5.1. Most notable, the CC keys now also have the attributes sensitiveDataOrigin, fixedParent and fixedTPM, indicating that the key is only ever known to the TPM. The TPM2_PolicyAuthorize serves as a placeholder for OEM definable policies for key usage; such as checking the EVCC firmware integrity.

| Attribute | $SRK$ | $PC$ and $CC$ |
|---|---|---|
| type: | TPM2_ALG_ECC | TPM2_ALG_ECC |
| nameAlg: | TPM2_ALG_SHA1 | TPM2_ALG_SHA256 |
| objectAttributes: | fixedTPM, fixedParent, restricted, decrypt, noDA, sensitiveDataOrigin, userWithAuth | fixedTPM, fixedParent, sign, decrypt, sensitiveDataOrigin |
| authPolicy: | n/a | TPM2_PolicyAuthorize |
| curveID: | TPM2_ECC_NIST_P256 | TPM2_ECC_NIST_P256 |

Table 5.1: Public Area Attributes of the TPM 2.0 Keys [53]

The hierarchy of keys is the same as that of TrustEV depicted in Figure 5.3. If HIP is used as extension for current ISO 15118 then SHA1 needs to be used as the nameAlg. Otherwise the encrypted certificate exceeds the CertInstallResp maximum message size. If HIP is used as extension for the next generation ISO 15118-20 then SHA256 can be used as nameAlg, since the maximum message size has been increased. The use of SHA1 is however still considered secure since the nameAlg here does not rely on the collision resistance [12].

#### ISO 15118 certificate installation/update request message

The CertInstallReq and CertUpdateReq messages need to be extended by the public key of the TPM generated Contract Credential key pair $CC_{pub}$. This new field PublicContractKey contains the 64-octed Base64 string representing this public key. Alternatively, it is being

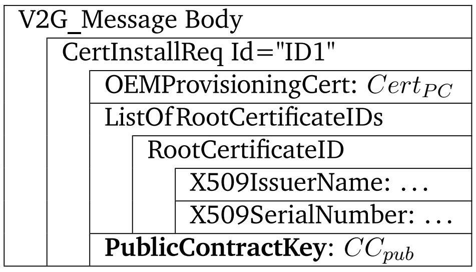


Figure 5.8: Extension of the ISO 15118 $CertInstallReq$ message [53]

discussed to use a Certificate Signing Request (CSR) in this field to carry the public key is a more PKI-CA friendly format. Figure 5.8 shows these extended messages.

Compatibility with regular ISO 15118 SECCs can be retained. ISO 15118 includes a protocol negotiation procedure where the major and minor version numbers are matched. In case the minor version number mismatches whilst the major version number matches, the SECC is supposed to ignore and forward unknown data elements sent by the EVCC ([80], Section 8.2.1).

#### Certificate Extension

The OEM Provisioning Credential certificate $Cert_{PC}$ is extended with the information needed by the MO to create the encrypted $Cert_{CC}$. Namely these information are used in a TPM2_ActivateCredential command. This information includes the public key $CC_{pub}$, the nameAlg as we define it, the SRK key and the policy digest of the policy to be associated with the $CC_{pr}$. The $SRK_{pub}$ is used for encrypting the Activate Credential data structure.

The remainder of these certificates are not altered and follow the certificate profiles specified in ISO 15118 [80], Annex F.

### 5.4.3 Integration of HIP in the ISO 15118 Protocol Flow

#### EV / EVCC manufacturing

The manufacturing process for HIP is exactly the same as with TrustEV, see Section 5.3.2.

#### Contract Credential Installation/Update

The Contract Credential provisioning process is depicted in Figure 5.10. The EVCC starts by generating a Contract Credential key pair in its TPM. It them constructs a CertInstallReq

| OEM Provisioning Certificate ($Cert_{PC}$) | | |
|---|---|---|
| Version: | | X.509v3 (0x2) |
| Serial Number: | | 12345 (0x3039) |
| Signature Algorithm: | | ecdsa-with-SHA256 |
| Issuer: | | CN=OEMSubCA2, O=ISO, C=US |
| Validity | Not Before: | May 7 08:40:32 2019 GMT |
| | Not After: | May 6 08:40:32 2021 GMT |
| Subject: | | CN=OEMProvCert, O=ISO, DC=OEM |
| Subject Public Key Info | Public Key: | OCTET STRING ($PC_{pub}$) |
| | Algorithm: | id-ecPublicKey |
| | Parameters: | namedCurve secp256r1 |
| X509v3 Extensions | Basic Constraints:[c] | CA:FALSE |
| | Key Usage:[c] | Digital Signature, Key Agreement |
| | **TPM 2.0 Extension:**[nc] | **TPM 2.0 EC Storage Root Key ($SRK_{pub}$) 512 bit OCTET STRING** |
| | | **TPM 2.0 SHA256 Policy Digest ($Pol$) 256 bit OCTET STRING** |
| Signature Algorithm: ecdsa-with-SHA256, Value: OCTET STRING | | |

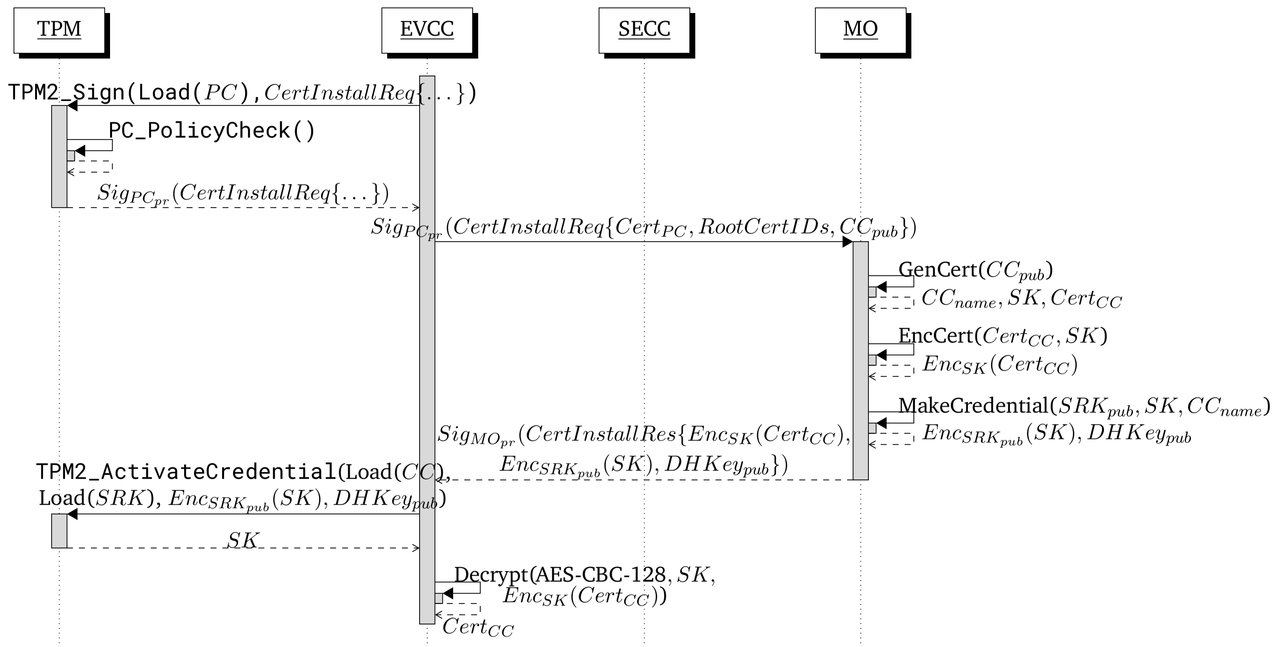
Figure 5.9: Provisioning Certificate with Custom Extension [53]

message containing the public key of this key pair. The CertInstallReq is then sent to the TPM for signing using the OEM Provisioning Key. The CertInstallReq message, together with the Provisioning Credential certificate, are send to the MO.

The MO verifies the CertInstallReq signature and extracts the $CC_{pub}$ public key for the to be created Contract Credential. It also extracts the $SRK_{pub}$ and $policyDigest$ from the $Cert_{PC}$. Using this data the MO creates a Contract Credential certificate $Cert_{CC}$. The MO then performs a TPM2_MakeCredential operation, where a Shared Secret $SK$ is generated that is used to encrypt the certificate. The SK is thereby embedded in a structure containing the identifier of the $CC_{pr}$ with all its TPM attributes (such as policyDigest) and is encryted using the $SRK_{pub}$. These two encrypted blobs are sent to the EVCC.

The EVCC passes the encrypted $SK$ to the TPM. The TPM will decrypt it using the $SRK_{pr}$ and validate that the included hash references $CC_{pr}$ alongside the $CC_{pr}$ object attributes. Only then will the TPM return $SK$ to the EVCC that in turn can decrypt $Cert_{CC}$. Now and only now has the Contract Credential been activated and is ready to use. If an adversary tried to send a software key to the MO, or a TPM key with different attributes (such as being exportable, or being generated outside the TPM) the TPM would deny the release of $SK$ and thus the activation of the $Cert_{CC}$.

If an MO would decide not to support the HIP extension, it would ignore the non-critical fields in the $Cert_{PC}$ and the additional field in the CertInstallReq and issue a regular CC



TPM
EVCC
SECC
MO
TPM2_Sign(Load($PC$),$CertInstallReq\{\dots\}$)
PC_PolicyCheck()
$Sig_{PC_{pr}}(CertInstallReq\{\dots\})$
$Sig_{PC_{pr}}(CertInstallReq\{Cert_{PC}, RootCertIDs, CC_{pub}\})$
GenCert($CC_{pub}$)
$CC_{name}, SK, Cert_{CC}$
EncCert($Cert_{CC}, SK$)
$Enc_{SK}(Cert_{CC})$
MakeCredential($SRK_{pub}, SK, CC_{name}$)
$Enc_{SRK_{pub}}(SK), DHKey_{pub}$
$Sig_{MO_{pr}}(CertInstallRes\{Enc_{SK}(Cert_{CC}), Enc_{SRK_{pub}}(SK), DHKey_{pub}\})$
TPM2_ActivateCredential(Load($CC$), Load($SRK$), $Enc_{SRK_{pub}}(SK), DHKey_{pub}$)
$SK$
Decrypt(AES-CBC-128, $SK$, $Enc_{SK}(Cert_{CC})$)
$Cert_{CC}$

Figure 5.10: Provisioning of Contract Credentials [53]

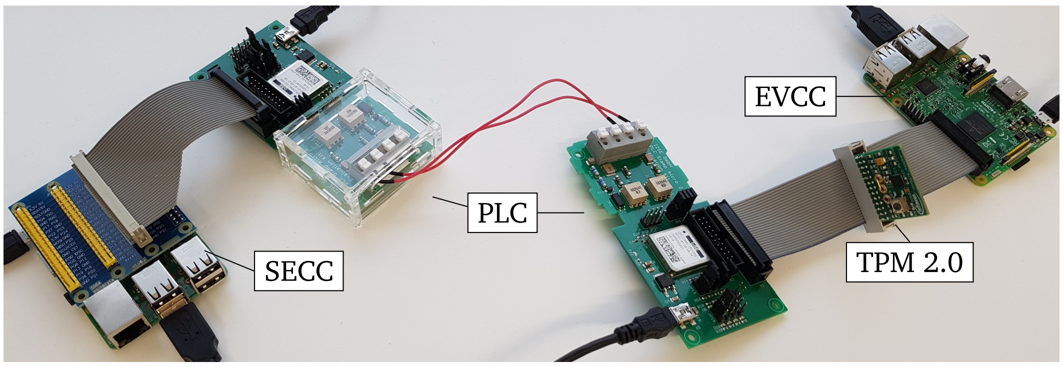


Figure 5.11: Test-bed Setup

key pair and certificate.

#### Contract Credential Usage

In order to use the Contract Credentials the EVCC sends the data to be signed to the TPM. It further needs to satisfy the E/A policy associated with the $CC_{pr}$. The returned signature can then be used in the ISO 15118 message as usual.

## 5.5 Implementation of TrustEV and HIP

Both TrustEV and HIP were implemented in [56, 53] using two Raspberry Pi 3B for EVCC and SECC. The EVCC was equipped with an Infineon Iridium 9670 TPM 2.0 chip and the PLC communication between the nodes was implemented using two PLC Stamp micro 2 EVBs. Figure 5.11 shows a picture of the setup.

For implementation, the ISO 15118 stack RISE V2G was used and the TPM2-TSS libraries were used for communication with the TPM. THe MO was also implemented on the SECC-Pi.

As policy for the $PC_{pr}$ and $CC_{pr}$ a TPM2_PolicyAuthorize was used with a signed policy containing a TPM2_PolicyPCR. The whole system could be demonstrated as feasible proof of concept implementation and the amount of adaptations, as described in the concept, could be verified.

## 5.6 Evaluation of HIP

### 5.6.1 Security Discussion

Given the usage of a TPM and the integration into ISO 15118 based on the HIP approach, the security of the overall system could be greatly enhanced. The following analysis of fulfilled security requirements stems from [53]:

1. *Secure key generation:* OEM provisioning key pair $PC$ and contract credential key pair $CC$ are both generated by the TPM (using the TPM's random number generator) and stored "under" the $SRK$ (cf. Section 5.4.3 and 5.4.3). Assuming the key generation of the used TPM is implemented correctly, security requirement 1 is fulfilled.
2. *Secure key storage:* As described in Section 5.4.2, $PC$ and $CC$ (and also $SRK$) carry the attributes fixedTPM and fixedParent preventing the export of private keys. Thus, only $PC_{pub}$ and $CC_{pub}$ can be exported and transmitted to other parties, e.g., backend systems. The private keys are securely stored in the TPM's shielded location. Thus, neither a backend hacker nor a car hacker can read out private keys and security requirement 2 is fulfilled.
3. *Secure cryptographic operations:* Security requirement 3 is also fulfilled, since the critical private keys are only used in the shielded area of the TPM and cannot be exported.
4. *Key usage authorization:* Security requirement 4 is fulfilled because the TPM checks the security policy every time a private key (OEM provisioning private key or contract certificate private keys) is accessed. By setting the policy *TPM2_PolicyPCR* (cf. Section 5.4.2), access to keys is only possible if the EVCC is in a trustworthy state.
5. *Cryptographic agility:* The TPM 2.0 is specified with a cryptographically agile interface. Modern TPMs already support multiple algorithms without any changes to the interaction model. Key lengths and algorithms can be chosen by simply altering the input structure. HIP works independently of the selected cryptographic algorithms. Insecure cryptographic algorithms can be easily exchanged. In rolled out systems, this would either require updating the firmware of the TPM or, if this is not possible, replacing the TPM. Security requirement 5 is fulfilled.
6. *Trustworthy credential enrollment:* The EVCC can only decrypt its $Cert_{CC}$ if the corresponding contract key was generated in the same TPM as the $SRK$ and sealed to the defined policy (cf. certificate extension in Figure 5.9) and also has the expected attributes (cf. Table 5.1). For a successful attack, an adversary would need to compromise the MO backend and to illegitimately control the TPM (e.g., by a run-time attack) for en-/decrypt $Cert_{CC}$ (e.g., with non-trustworthy parameters). This

attack vector is more complicated than a direct MO backend misuse and thus beyond the adversary model. Security requirement 6 is fulfilled.

### 5.6.2 Discussion of Functional Requirements

Furthermore, the following functional analysis of the HIP approach was presented in [53]:

1. *Minimal overhead:* To show 1 is fulfilled, the communication and computational overhead is analyzed. The goal was to stay within the limits defined by ISO 15118, which is fulfilled in all cases.

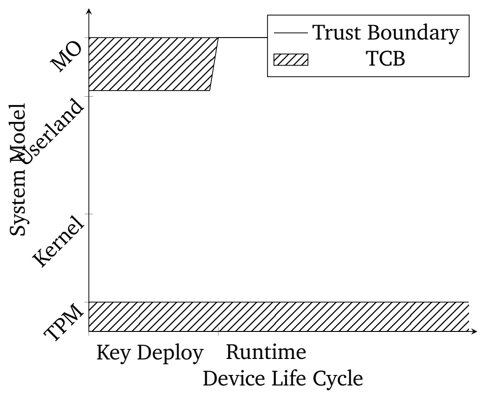
    - With respect to the communication overhead, the field for $Cert_{CC}$ is limited to 800 bytes and the field for encrypted contract keys is limited to 48 bytes. In this implementation, the encrypted $Cert_{CC}$ including the IV is transmitted in the certificate field. It requires 608 bytes. The encrypted AES session key $SK$, transmitted in the contract key field, requires 38 bytes[3]. Note that the upcoming edition ISO 15118-20 increases the limit for encrypted contract keys to 64 bytes. This would allow the use of SHA256 for the $SRK$ with an effective size of 50 bytes for the encrypted $SK$.
    - The computational overhead shall not result in exceeding timing parameters defined by ISO 15118 (cf. [80], Table 109). The maximum time between an EVCC's *CertInstallReq* message generation and the corresponding response is 4500 ms. Thus, the additional operations required by our protocol extensions at the MO should not be too time-consuming. The measurements were repeated 100 times using Java's System.nanoTime(). On average, it took 452.5 ms between *CertInstallReq* and the its response and never exceeded the maximum value. Note that, even though our measured times are arguably far higher than the expected values from a real MO, as our implementation is mostly in Java and runs on Raspberry Pis, our implementation still managed to stay far below the required performance time. Thus, any real-world implementation should not cause protocol incompatibilities, even assuming a much greater round trip times between the actors. Our protocol extension also introduces additional overhead by using the TPM from the EVCC, which falls within the EVCC sequence performance time of 40000 ms. The time for the entire certificate provisioning process is measured, between the start of signing the request and the end the decryption of the received $Cert_{CC}$ (cf. Figure 5.10) 100 times. The maximum value was never reached and the average value was

[3]20 byte HMAC and 18 byte encrypted ($SK_{size}||SK$), required by the TPM.

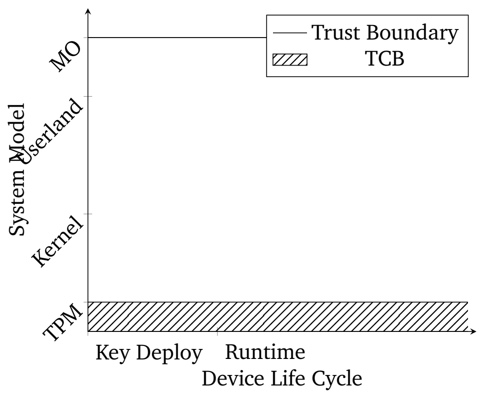
with 5385.1 ms – far below the maximum value. In addition, the time for signing a PnC authorization request with the TPM using $CC_{pr}$ (including the policy assertion) individually was measured, which was 753.3 ms.

2. *ISO 15118 conformance:* Our protocol extension slightly alters the ISO 15118 protocol to increase security. The request message is changed and a minor protocol version number introduced. The fields in the response message are refined. The protocol message flow remained unchanged. Since these minimal changes are required to substantially increase the security, the functional requirement 2 is also fulfilled.
3. *Backwards compatibility:* Since HIP does not change the general ISO 15118 message flow and the introduced certificate extension is marked as non-critical, intermediate nodes (e.g., the SECC) simply forward the messages ensuring backwards compatibility. Thus, functional requirement 3 is fulfilled.

## 5.7 Conclusion on Secure PnC



(a) TrustEV



(b) HIP/HIP-20

Figure 5.12: Trusted Computing Base for basic PnC

This chapter describes the concept, prototypical implementation, and evaluation of HIP, a protocol extension for ISO 15118 to enable secure key generation and usage in an HSM (in form of a TPM) within the vehicle. With our solution, the (private) keys of the OEM provisioning certificate and the contract certificates are generated and used only within the secured area of the TPM. It ensures backwards compatibility, i.e., entities which do not support the extension can still use the regular ISO 15118 mode. The introduced protocol changes are all not critical and the introduced additional overhead is still within the constraints defined by ISO 15118. The protocol extension could be integrated in future versions of the ISO 15118 to improve the protection of (private) keys.

As depicted in Figure 5.12, both TrustEV and HIP reduce the TCB by having only the TPM on the client device be part of the TCB. The difference between TrustEV and HIP becomes obvious on the inclusion of the Mobility Operator as part of the TCB in TrustEV whilst in HIP the MO only issues the certificate but does not create or ever know the key.

# 6 Time-based Unidirectional Attestation for Audit Logging

This chapter presents two approaches using hardware security modules to retrieve trusted measurements of devices. The second of those is proposed as a solution for time-based attestation statements using TPM 2.0. It consists of a synchronization phase between an TPM's internal clock and a global real-time clock and of attestation-tokens that are constructed against the TPM's internal clock. In comparison to earlier approaches (TUDA v1) that were designed for the hardware security module TPM 1.2, the approach in this chapter makes use of new feature-sets of TPM 2.0 which allow for a simpler and more efficient protocol and implementation. This work is an extension to my work in [43, 116] and the extensions in [55].

## 6.1 Background on TUDA based Audit Logs

Dedicated crypto-processors, i.e. hardware security modules (HSM), provide the basis for trusted measurements about the endpoint ("any computing device that can be connected to a network" [118]) they are installed in. This mechanism can be used to attest the identity of an endpoint and even the integrity of running software on an endpoint to a remote party. Typically, such remote attestations are conducted using protocols based on a challenge-response procedure in order to ensure freshness and/or recentness of the attestation. Unfortunately, the necessary bi-directional communication is not feasible in every usage scenario. The current trend in the IETF and W3C to employ REST interfaces in the IoT domain [31], or the goal to leverage established interfaces, such as SNMP, in the domain of network equipment would benefit from a protocol that requires only uni-directional communication since they do not offer parameterized Remote Procedure Calls.

For the analysis in this thesis the focus lies on the capabilities for audit logging of the integrity state of platforms, rather than the application to live peer-to-peer remote attestation. The security policy is that the integrity state of a platform shall be auditable for a long time even after the point in time that was attested. This is typically done by having a verifier performing peer-to-peer attestations and storing them in a secured long-term log; OpenCIT (formerly OpenAttestation [73]) is an example for this approach. This however makes the verifier part of the TCB for attestation statements as depicted in Figure 6.1. A relying party cannot revalidate any of the attestation statements afterwards or especially in the distant future. On the device itself, we see that the TCB consists only of the TPM's Roots of Trust, where the Core Root of Trust for Measurement (CRTM) is one of them, implemented as part of the early boot code of the CPU.

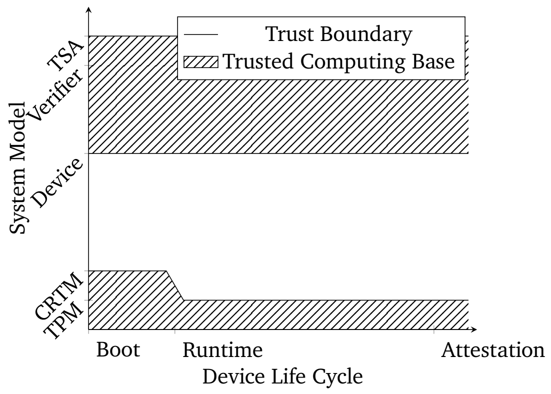


Figure 6.1: Trusted Computing Base for Verifier-driven Integrity Audit Logs

The time-based uni-directional attestation (TUDA) protocol is based on the general idea initially presented in [116] and further refinement of these concepts that are currently[1] evolving in the Internet Engineering Task Force (IETF) [55]. The essential principle of the TUDA protocol is built on time-based attestation statements about the integrity status of the attestee for a certain point in time. TUDA uses a Time Stamp Authority (TSA) that is available to both parties of the remote attestation – the attestee and the Verifier – to synchronize the TPM's internal clock with a UTC clock. Using regular regeneration of attestation statements, this approach derives almost equal recentness properties compared to regular challenge-response schemes.

While the approach presented in this chapter can leverage any HSM that satisfies the requirements of TUDA, currently the Trusted Platform Module specified by the Trusted Computing Group provides the required features. As a consequence, the term Hardware Security Module is used in this chapter when general characteristics and features are illustrated and the term Trusted Platform Module is used when specific functions and

[1]`https://datatracker.ietf.org/doc/draft-birkholz-rats-tuda/04/`

procedures are described, respectively.

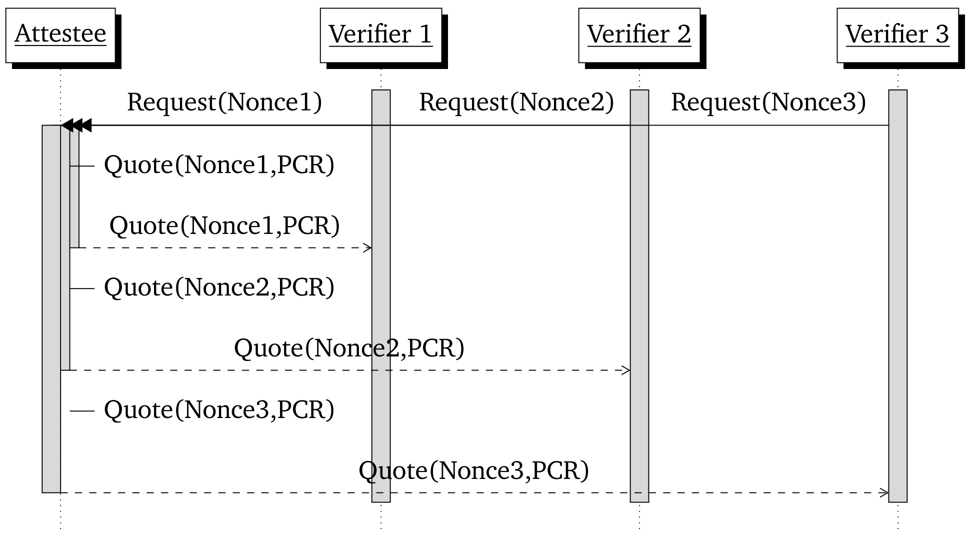


Figure 6.2: PTS-like Attestation

The capability to provide trusted measurements about an endpoint is an essential building block to enable automation of security decision [114],[14]. Existing solutions for endpoint assessment, such as the Network Endpoint Assessment protocol [118] or the use of the Open Vulnerability Assessment Language [140] focus on the acquisition and evaluation of collectable endpoint attributes [13]. It is vital to increase the trustworthiness of providers of information. Their assessments serves as a basis for security automation; especially if an end point's configuration and state is self-reported by the target endpoint (the "endpoint of interest" [13]). In combination with, for example, secure boot mechanisms [68], an attested source of target endpoint attributes can provide the basis for basic automated security decisions without the need of human interaction.

While the required building blocks for production solutions, e.g. appropriate HSM [61, 62, 63] and remote attestation protocols such as the Platform Trust Service (PTS) [119] already exist or start to emerge [126], [112], they focus on traditional enterprise usage scenarios [74]. Parallel execution of multiple invocations can only be executed in order, leading to time delay, see Figure 6.2. New architectures are being developed to deploy and manage constrained devices on a very large scale [124], some of which include large numbers of devices that are not owned by customers. This is the case in some usage scenarios of software updates, such as [14]. These devices require both more lightweight HSM and more lightweight attestation protocols due to the constrained resources available;

most prominently power consumption and network bandwidth.

A protocol that enables target endpoints to attest themselves without the need of a bi-directional communication channel also allows for the use of broadcast and multicast transports, or enable the use of new technologies such as the IEEE 802.11p [81] car-2-car or car-2-infrastructure communication.

A prominent example of a domain in which a challenge-response mechanic is not feasible, is the Web of Things in the context of the W3C [137]. Communication between devices in the Web of Things is typically facilitated via Representational State Transfer (REST) interfaces. The stateless nature of these interfaces is a guiding principle for corresponding protocols and data exchange procedures. While it is possible to map a challenge-response mechanism to REST interfaces, e.g. by creating separate REST endpoints for each RPC, this approach would conflict with the intent of the REST architecture specifically not to include RPCs in its architecture.

The work presented in this chapter is a significant improvement compared to the approach TUDA v1 presented in [116]. TUDA v1 for TPM 1.2 requires the use of a key hierarchy with two keys. The lower key is a restricted keys that needs to be regenerated for each change of PCR values. The TPM 2.0 allows for less overhead in regard to transferred data and reduction of computing time required from the HSM by merely requiring one key in total. Also note, that since TPM 2.0 is a backwards-incompatible re-engineered API, that TUDA v1 cannot be used for TPM 2.0 based systems.

At the same time, the advantages of TUDA v1 in comparison to challenge-response based attestation protocols are retained. A prominent example of a challenge-response based attestation procedure also utilizing a TPM is the PTS Attestation Protocol of the Trusted Computing Group (TCG) [119]. The PTS protocol allows to select which aspects of a remote endpoint a Verifier wants to evaluate. The specification defines a local service running on the Attestee (the Platform Trust Service) and corresponding interfaces to measure the target endpoint itself and to provide an attestation capability to remote Verifiers. The approach presented in this chapter does not focus on the acquisition of measurements on the Attestee. The same measurement procedures that are used in the Attestation PTS Protocol are also used with TUDA. The contribution of this chapter focusses on a data model that can be transported using several existing interface architectures, such as RESTCONF.

### 6.1.1 TUDA v1

The time-based uni-directional attestation (TUDA) presented in [55, 116] uses a trusted source of time — a Time Stamp Authority (TSA) that can produce trusted Time Stamp Tokens (TST) — to eliminate the need for a challenge-response. In a REST architecture, for example, if the same TSA is available to both parties — the Verifier and the Attestee — the remote attestation between them can be conducted using only GET but no POST operations (if the Verifier initiates the remote attestation) or only POST but no GET operations (if the Attestee initiates the remote attestation).

Additionally, the TUDA protocol relies on the capability of an HSM included in the Attestee to operate on restricted keys. Whenever a measurement stored in an HSM changes, a new restricted key is created by the HSM that can only be used as long as the current measurement value stored in the HSM is not changed by an updated measurement value. Restricted keys are temporal keys that are generated by the HSM internally and their validity is based on measurements stored inside the HSM. The TPM specified by the TCG satisfies these requirements.

TUDA transfers six chunks of information, called Information Elements (IE). The IEs are the content that is transported from the Attestee to the Verifier. The term freshness property in this context refers to the size of the time-window during that it is safe to assume that the values of the IEs are still valid. An IE that is still valid inside its specific time-window is referred to as fresh. The information contained in all IEs combined could be sent en-bloc with every remote attestation, but this is not necessary if the state of some of these IE is still fresh, see Figure 6.3.

Analogously, TUDA defines specific events at which the values of one or more IE change.

- Setup: TUDA is conducted the first time between Attestee and Verifier. Setup also occurs when the Identity Key Certificate of the Attestee or the TSA Certificate changes (see Figure 6.3).
- Initialization: the integrated HSM is reset after completing a boot-cycle (see Figure 6.3).
- Synchronization: due to clock drift, the relative time of the HSM integrated in the Attestee has to be anchored with the absolute time of the Verifier (see Figure 6.4).
- Measurement Change: the measurements stored in the HSM change and restricted keys have to be renewed (see Figure 6.4).

After each event a set of fresh IE has to be transmitted to the Verifier in order to successfully conduct an attestation.

In the following, each of the IEs are explained in more detail, and their core information is depicted in pseudocode. Note that the actual data structures are more complex than depicted here and may contain additional information that is not relevant for this use case.

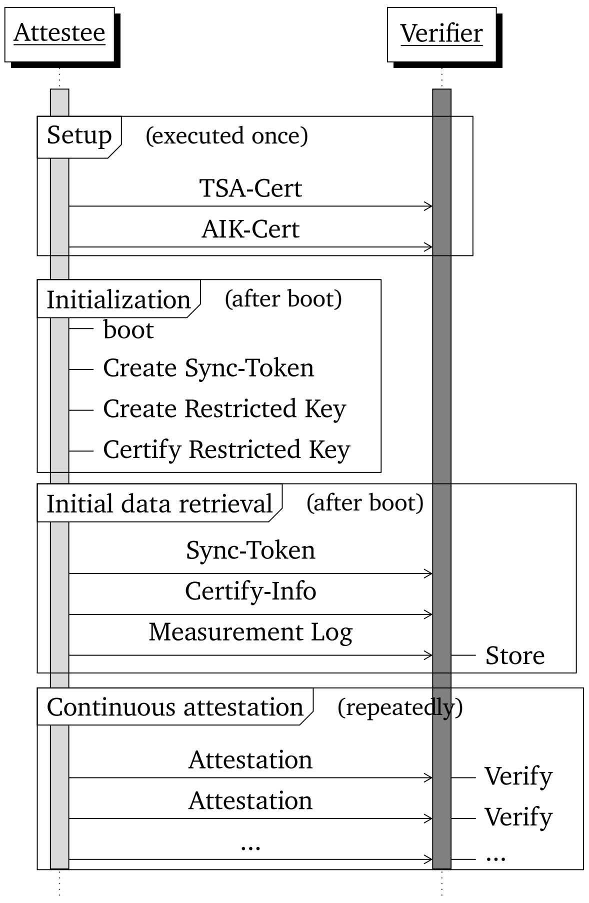


Figure 6.3: TUDA v1 initial attestation

**TSA Certificate** An IE that includes a digital certificate for the key used by the timestamp authority (TSA) [2] issued by the TSA's Certificate Authority. This certificate establishes the trust into the timestamps that are used to synchronized the TPM's clock with a global universal time clock (UTC). This IE only has to be transported with the first TUDA conducted between the Verifier and the Attestee or if the TSA changes.

```
TSA-Certificate :=
  X.509-Certificate(TSA-Key, TSA-Flag)
```

**Attestation Identity Key Certificate** An IE that includes a digital certificate about an identity key that is generated by the HSM internally (e.g. a TPM Attestation Identity Key – AIK) signed by a certificate authority. This key forms the identity basis for the attestations. Furthermore, the certificate must denote that the certified key is an AIK key. This means that the key cannot be used to sign arbitrary externally provided data. This allows the key to be used for signing special TPM-internal data structures, such as the *Restricted Key Token*. This Key is included in a certificate and may be a IEEE 802.1ar-like IDevID or LDevID attestation key [77], depending on the settings of the corresponding identity property.

```
AIK-Certificate :=
  X.509-Certificate(AIK-Key,
                    Identity-Flag)
```

**Synchronization Token** This IE contains the references for attestations based on relative time since boot provided by the HSM, called tick session. In order to associate the relative time provided by the Attestee with the absolute time available to a Verifier, a cryptographic synchronization between the tick session and the trusted absolute time provided by the TSA is transported via this IE.

```
SyncToken := {
  left_TickStampBlob :=
    sig(AIK-Key,
      currentTicks :=
        {tickValue, tickNonce})
  middle_TimeStamp :=
    sig(TSA-Key, h(left_TickStampBlob),
      UTC)
  right_TickStampBlob :=
    sig(AIK-Key, h(middle_TimeStamp),
      currentTicks :=
        {tickValue, tickNonce})
}
```

**Restricted Key Token** This IE contains a TPM-generated special data structure about a newly created restricted attestation key. This restricted key can only be used if the platform is in a certain state – i.e. if the PCRs of the TPM contain a predefined value. The data structure serves as a certificate that provides proof of this restriction on the key and is signed with the *Attestation Identity Key*.

```
Restriction-Token :=
  sig(AIK-Key,
    Restricted-PubKey,
    PCR-Info)
```

**Measurement Log** This IE contains the set of measurements associated with a specific Restricted Key. Based on these measurements the PCR values of the restricted key were derived. The verifier can use the measurement log in order to separately judge the trustworthiness of each software component that was loaded and to reconstruct the PCR values for the Restricted Key Token.

```
Measurement-Log :=
  List(EventName, PCR-Num, Event-Hash)
```

**Attestation Token** An IE that includes an attestation based on a PCR restricted key that provides evidence that a certain PCR configuration on the Attestee was given at a certain point in time. The relative time since boot used in this evidence can be associated with the absolute time available to the Verifier using a fresh Synchronization Token.

```
Attestation-Token :=
  sig(Restricted-Key,
    currentTicks :=
      {tickValue, tickNonce})
```

## 6.2 Approaches for TUDA on TPM 2.0

In order to provide a variant of TUDA for TPM 2.0, two different approaches are investigated; a one-to-one porting of the protocol to TPM 2.0 and a redesigned version that uses new features of TPM 2.0. This section gives an overview of the two approaches and explains the decision for proposing the second, redesigned approach for application as TUDA v2.

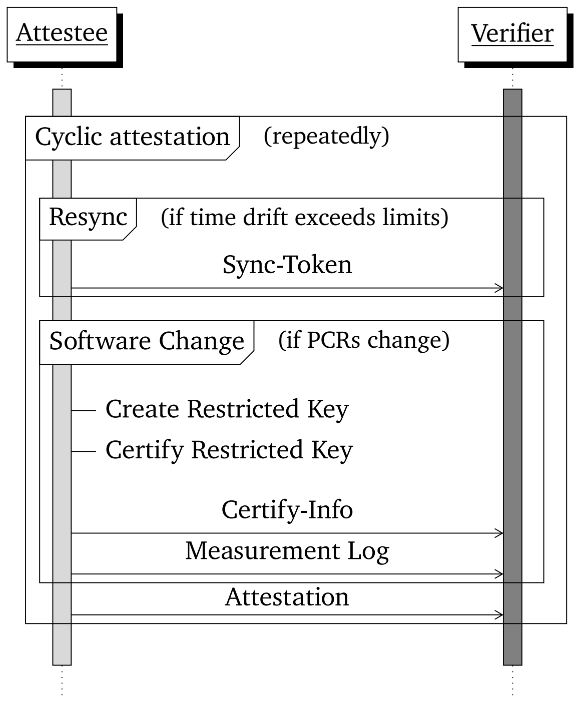


Figure 6.4: TUDA v1 continuous attestation

### 6.2.1 Approach 1 – TUDA for TPM 2.0 with Key Restriction

The first approach is the direct translation of TUDA v1 to TPM 2.0. The Informational Elements of this approach are the same as with the original TUDA, see Section 6.1.1. The data model however differs due to the new structures of TPM 2.0; specifically those that were signed by the TPM.

In the following, each of the IEs are explained in more detail, and their core information is depicted in pseudocode. Note that the actual data structures are more complex than depicted here and may contain additional information that is not relevant for this use case (see Section 2.4 for detailed descriptions).

**TSA Certificate** No changes from TUDA v1.

```
TSA-Certificate :=
  X.509-Certificate(TSA-Key, TSA-Flag)
```

**AIK Certificate** No changes from TUDA v1. In the terminology of TPM 2.0, the AIK is now called a “restricted key” instead of an “identity key”.

```
AIK-Certificate :=
  X.509-Certificate(AIK-Key,
                    Restricted-Flag)
```

**Synchronization Token** The synchronization token uses a different TPM command, TPM2_GetTime() instead of TPM_TickStampBlob(). The TPM2_GetTime() command contains the clock and time information of the TPM. The clock information is the equivalent of TUDA v1’s tickSession information.

```
SyncToken := {
  left_GetTime :=
    sig(AIK-Key,
      TimeInfo :=
        {time, resetCount, restartCount})
  middle_TimeStamp :=
    sig(TSA-Key, h(left_TickStampBlob),
      UTC)
  right_TickStampBlob :=
    sig(AIK-Key, h(middle_TimeStamp),
      TimeInfo :=
        {time, resetCount, restartCount})
}
```

**Restriction Info** The restriction information contains the same Informational Element as with TUDA v1. The difference is that with TPM 1.2 the restriction information was containd as part of the key’s public information. TPM 2.0 introduced Enhanced Authorization instead. The restriction to certain PCR values in this case is defined as a policy statement containing a TPM2_PolicyPCR element referencing the according PCR selection and values. The digest of this policy statement is registered in the public area of the key during key creation. In order to provide proof of this PCR restriction, the command TPM2_Certify() is used. The restriction information accordingly consists of PolicyPCR-information, KeyPublic-information and the certificate of this key.

```
Restriction-Token := {
  pcr-restriction :=
   {PCR-Selection, PCR-Values}
  key-certificate :=
    sig(AIK-Key,
      Restricted-PubKey,
      PolicyPCRdigest(pcr-restriction))
```

**Measurement Log** No changes to the contained information. The only difference is that due to the TPM 2.0's cryptographic agility, the hash algorithm must be provided and the length of hash values my be different.

```
Measurement-Log :=
  List(EventName, PCR-Num, Event-Hash)
```

**Attestation Token** The attestation token consists of the result of TPM2_GetTime(). It serves the same purpose as the TPM_TickStampBlob() of TUDA v1. It proves that at a certain point in time with repsect to the TPM's internal clock, a certain configuration of PCRs was present, as denoted in the key's restriction information.

```
Attestation-Token :=
  sig(Restricted-Key,
    TimeInfo :=
      {time, resetCount, restartCount})
```

### 6.2.2 Approach 2 – TUDA for TPM 2.0 with TPM-Time Quotes

The second approach describes a new concept that works differently from the concept of TUDA v1. Instead of relying on the TPM's time value this approach utilizes the TPM's clock value. In contrast to the time value, the clock value is embedded within every TPM2_Quote statement. This means that it can be utilized in explicit attestations as opposed to merely implicit attestations. The attestation token, which forms the core of the protocol, can use this in order to omit the creation of an intermediate key. Instead the AIK is used directly within the attestation tokens (see Figure 6.5). The TPM2_Quote includes a PCR-Selection and signature over its values alongside the clock information. This simplifies the approach drastically, since no intermediate restricted keys need to be generated for every change of PCRs. Furthermore, it is possible to create attestation tokens for several PCR selections in parallel, using the same key. This can be useful if not all information from PCRs shall be disclosed to all requesting parties. The respective update cycles and event are illustrated in Figure 6.6.

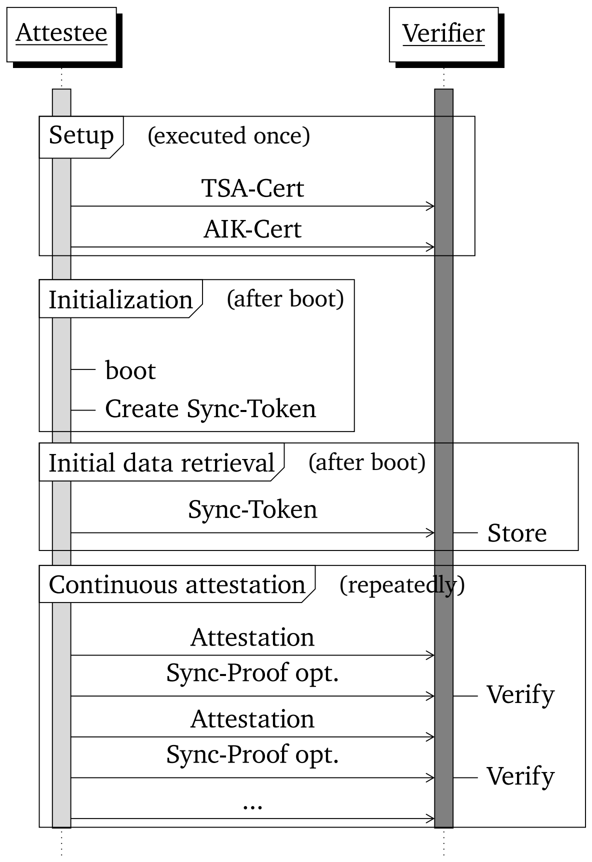


Figure 6.5: TUDA v2 initial attestation

However, the Clock value (as explained in Section 2.4) is not a constantly incrementing counter. Even though it is a strongly monotonic counter, it can be "fast-forwarded" using the TPM2_ClockSet command. Also, the current value of the Clock is stored persistently over reboot cycles.

There is a potential attack vector that could be exploited by "fast-forwarding" the Clock value. An attacker could try to gain access to the an Attestee in order to "fast-forward" the clock and produce attestation tokens for a point of time in the future. Delivering the precomputed attestation tokens instead of current tokens would provide falsified evidence of the platforms status.

Two possible alternatives exist in order to cope with this attack. It can either be known that the TPM2_ClockSet command is protected appropriately, such that only legitimate

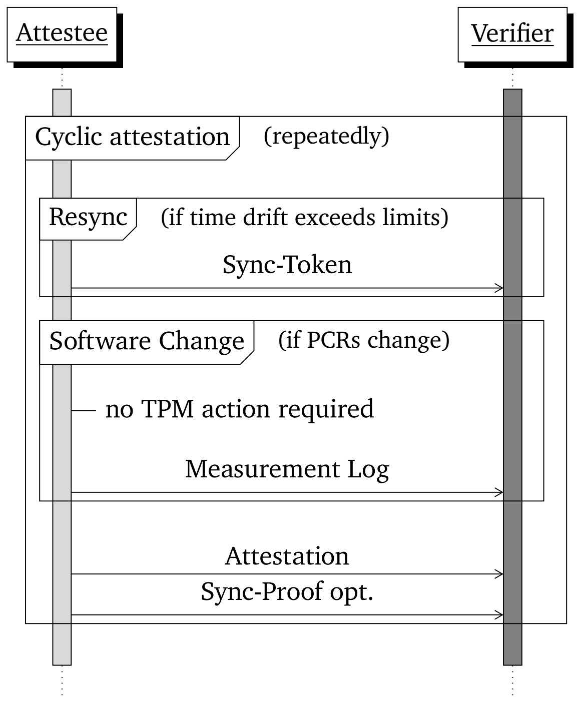


Figure 6.6: TUDA v2 continuous attestation

calls to “fast-forward” can be issued on the Clock value. Alternatively, a proof can be generated that provides evidence that the Clock value has not been “fast-forwarded” since the last synchronization token has been generated.

The latter approach requires a sync proof to be collected after the attestation token. The Verifier can then calculate the difference between the TPMs “non-forwardable” time value and the “fast-forwardable” Clock value and compare it to the difference between those values within the sync token. If the differences match, then no (effective) TPM2_ClockSet was called and the attack on fast forwarding can be excluded.

In the following, each of the IEs are explained in more detail, and their core information is depicted in pseudocode. Note that the actual data structures are more complex than depicted here and may contain additional information that is not relevant for this use case (see Section 2.4 for detailed descriptions).

**TSA Certificate** No Change compared to the previous approach.

```
TSA-Certificate :=
  X.509-Certificate(TSA-Key, TSA-Flag)
```

**AIK Certificate** No Change compared to the previous approach. However, the AIK is used to sign the attestation tokens directly in this approach instead of certifying the restriction properties of an intermediate key.

```
AIK-Certificate :=
  X.509-Certificate(AIK-Key,
                    Restricted-Flag)
```

**Sync Token** No Change in the transferred data. However, this approach now uses the clock value instead of the time value in order to calculate the point in time of the attestation token generation. If a sync proof is used, then the clock value is used for calculating the difference of clock and time in this token and compare it to the difference between clock and time in the sync proof.

```
SyncToken := {
  left_GetTime :=
    sig(AIK-Key,
    ClockTimeInfo :=
      {clock, time,
       resetCount, restartCount})
  middle_TimeStamp :=
    sig(TSA-Key, h(left_TickStampBlob),
      UTC)
  right_TickStampBlob :=
    sig(AIK-Key, h(middle_TimeStamp),
    ClockTimeInfo :=
      {clock, time,
       resetCount, restartCount})
}
```

**Measurement Log** No Change compared to TUDA v1.

```
Measurement-Log :=
  List(EventName, PCR-Num, Event-Hash)
```

**Attestation Token** This token consists of the output of a TPM2_Quote(). It contains a PCR selection and the values alongside the current value of the TPM's internal clock. The AIK is used to sign this structure.

```
Attestation-Token :=
  sig(AIK-Key,
    quote-info :=
      {PCR-Selection, PCR-Values}
    ClockInfo :=
      {clock, resetCount, restartCount})
```

**Sync Proof (optional)** This token contains the output of one TPM2_GetTime. It is signed using the AIK. Within this output, the difference for the values of clock and time is computed and compared to the difference for the values of clock and time inside the previous sync token. If the differences match, then the TPM's clock was not "fast-forwarded".

```
SyncToken :=
  sig(AIK-Key,
    ClockTimeInfo :=
      {clock, time,
       resetCount, restartCount})
```

### 6.2.3 Comparison and Choice

The second approach is much simpler and efficient to use in practical applications.

In the first approach each change in the platform status – i.e. each change in PCR values – requires the generation of a new key inside the TPM that contains a different restriction. Then the AIK must perform a TPM2_Certify on this key, before the key can be used to actually sign attestation tokens using TPM2_GetTime. In a dynamically changing system, this can occur very frequently, which introduces a huge overhead. Additionally, stateless transport interfaces such as REST and SNMP can provide the different IE independently in order to save network bandwidth. This can lead to cases where a change occurs between two requests – e.g. between retrieving a restriction info and an attestation token. This additional complexity can be partially solved within the transport interfaces – e.g. using update cycle counters – but can hardly be resolved completely. The Verifier must cope with this introduced complexity. The second approach on the other hand does not require any action for PCR changes except for updates to the measurement log, that the first approach requires as well. However, having the measurement log out of sync with the attestation token is easy to handle, since a Verifier can continuously recalculate the PCR candidates for the start of the measurement log and continuously append events and check if the resulting PCR candidate matches the attestation tokens PCR value.

Regarding the interaction with the TPM, the first approach requires the use of Policy Sessions in order to prove to the TPM the PCR restriction using TPM2_PolicyPCR. The attestation tokens in the second approach can be provided using no sessions at all, since the AIK does not need to be access protected. Only the TPM2_GetTime function requires the authentication of the TPM's privacy administrator (i.e. the Endorsement Hierarchy). This authentication can however also be set to zero, leading to the complete omission of sessions in this approach.

The only advantage of the first approach is that continuous attestations require only a single signature for the attestation token, whilst the second approach optionally requires a second signature for the sync proof.

In summary, the gained simplicity and network efficiency originating from the omission of a restricted, intermediate key and the omission of the session outweigh the optional overhead of the second approach, especially since the overhead only occurs optionally. For practical implementations the second approach with a non-zero owner authorization to prevent "fast-forwarding" and thereby omission of the sync proof is by far the most efficient approach. Therefore, the second approach is proposed as candidate for a TUDA v2.

## 6.3 TUDA Synchronization Protocol

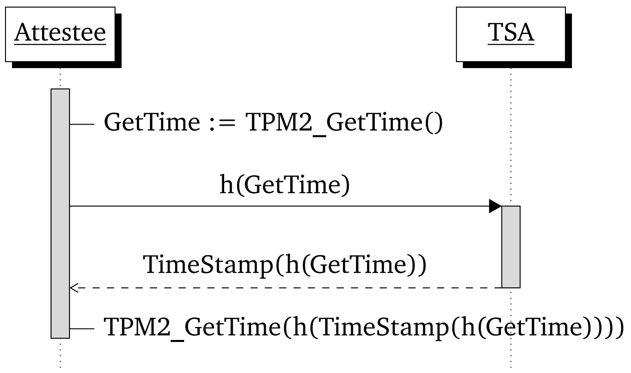

Attestee
TSA
GetTime := TPM2_GetTime()
h(GetTime)
TimeStamp(h(GetTime))
TPM2_GetTime(h(TimeStamp(h(GetTime))))

Figure 6.7: Synchronization Protocol

The synchronization protocol recommended for this application has similar properties to the protocol proposed in [116]. It differs however in order to allow the usage of a IETF RFC3161 compliant Timestamp Authority (TSA) [2]. In this protocol, the TPM generates a time token using TPM2_GetTime for the current point in time (called left). The device sends the hash of this token to the TSA, which signs it and returns the TimeStamp. The hash of this TimeStamp is then fed into another call to TPM2_GetTime (called right). The protocol is illustrated in Figure 6.7. The synchronization property of this protocol is then based on the fact that the TSA TimeStamp must have happened between the two executions of the TPM2_GetTime calls.

The synchronization token is then composed out of the three signed structures for TPM2_GetTime left, TSA-TimeStamp and TPM2_GetTime right.

The security of this protocol relies on the fact that the time basis of the first TPM2_-GetTime is unique. Since the TPM will advance its internal clock continuously and only reset via TPM2_Clear which would however invalidate the key from usage, the left structure, signature and resulting hash value are unique values. They denote that as this signature was generated at the denoted point in time, relative to the TPM's internal clock. This provides a fresh and unfakable nonce for the TSA TimeStamp and the TSA TimeStamp thereby provides a right-limited border for the TPM's clock. In order to prevent the replaying of the same hash to the TSA at a later point in time (i.e. to be able to ignore such cases) another TPM2_GetTime operation is performed which provides a left-limited border for the TPM's clock.

For an attacker with control over the device, two things are not possible. The attacker could re-execute the right TPM2_GetTime, which would however only lead to a decrease in accuracy of the synchronization token regarding its left limit. Alternatively, the attacker could re-execute steps 2 and 3 of the protocol, re-sending the hash-nonce of the left TPM2_GetTime to the TSA and providing an alternative right TPM2_GetTime for this new TimeStamp. This would however merely decrease the accuracy of the right limit. In summary, there is no means for an attacker to produce a false synchronization or to maliciously increase the accuracy beyond the given point. The attacker can only decrease the accuracy of the sync token.

Another problem with regards to accuracy originates for the TPM's internal clock itself, which can have a very high drift. This may require the periodic resynchronizations using this protocol or the adjustment of TPM2_ClockRateAdjust. See Section 6.4.1 for a more detailed elaboration.

## 6.4 Discussing TUDA based Audit Logs

In the following certain aspects of the presented approach for TUDA v2 are discussed.

### 6.4.1 Time Accuracy

The security of the presented approach is directly related to the accuracy of the point in time for the attestation tokens. This accuracy depends on two main factors; (i) the accuracy of the sync token and (ii) the drift of the TPM's internal clock.

The accuracy of the sync token depends on the round-trip times between the first TPM2_GetTime $t_{sl}$ and the second TPM2_GetTime $t_{sr}$. The timestamp of the TSA may have occurred at any point in time between those two TPM2_GetTime calls. A worst-case estimate for the accuracy of the sync token further needs to account for the TPM's clock's drift. The resulting equation is:

$$(t_{sr} - t_{sl}) \cdot (1 + drift_{TPM})$$

The drift of the TPM's clock value is vendor specific. The TPM 2.0 Library Specification Part 1 [128] defines that the maximum drift may be $drift_{TPM} = \pm 15\%$. The concrete value depends on the input frequency accuracy and operation temperature of the TPM which may be accessible to an attacker but these limits can still generally be expected to be much more precise. If a TPM is managed by a trustworthy party, then the TPM2_-ClockRateAdjust function can be used to counteract such drifts.

Nevertheless, a high drift and latency merely result in the requirement for more frequent resynchronization of the TPM's internal clock with the TSA clock. This means that new sync-tokens are retrieved regularly. For the verifier, this means that the maximum error of the time for an attestation token given the time since sync $t_{\Delta s}$ is:

$$err = (t_{sr} - t_{sl}) \cdot (1 + drift_{TPM}) + drift_{TPM} \cdot t_{\Delta s}$$

Given a certain $threshold$ for the accuracy of the time-based attestation, the following equation must hold:

$$t_{\Delta s} \leq \frac{threshold + (t_{sl} - t_{sr}) \cdot (1 + drift_{TPM})}{drift_{TPM}}$$

### 6.4.2 Security and Architecture Properties

Besides the ability to bind against transport interfaces that perform RPC-less state transfer such as REST or SNMP, the presented approach can also server other scenarios where bi-directional attestation protocols are not fit.

Most notably, the unidirectional aspect allows the use in multi-client scenarios, including REST with caches. In regular attestation protocols, each client request requires its own TPM signature, such that the TPM's throughput becomes the bottleneck. This was also the original motivation for the related work in [116]. Since the attestation tokens can stand on their own, the same token can be distributed to as many clients as necessary, even in multicast and broadcast environment.

In certain scenarios, the underlying secure requirement defines that data transfer may only occur from a secure area to an insecure area. Typical scenarios are nuclear power plants. The goal is here to prevent any interference with the highly security and safety critical operations of the actuators inside the security zone. Data transfer between secure and insecure area is regulated by data flow diodes, that allow data to only travel in one direction. In order to receive integrity information from the inside, a cyclic event stream from inside to outside must occur and time-based attestations are a very suitable solution.

## 6.5 Conclusions on TUDA based Audit Logs

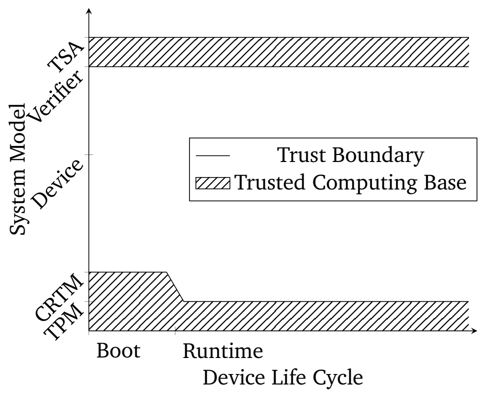


Figure 6.8: Trusted Computing Base for Time-driven Integrity Audit Logs

Due to their time-based nature, the attestation approach of this chapter can be used to create trusted audit logs. An audit report about the integrity state of the Attestee during a certain time span can be created by leveraging the time-based attestations created when a TUDA is conducted. If measurements are sent from the Attestee on a regular basis and stored appropriately on the Verifier or a corresponding audit server back-end, it is possible to generate a complete audit log about the Attestee. The transferred IE constitute forensic evidence and provide a proof of the state that can be reevaluated for any point of time in the past. The resulting TCB is depicted in Figure 6.8. The Verifier can be removed from the TCB and a relying party can reevaluate the attestation. Only the Time Stamp Authority remains part of the TCB besides the TPM's Roots of Trust.

# 7 Role based Access Management

Many embedded scenarios are composed of a variety of embedded systems that act as a compound system or trust domain. In order to grant users access to such a system, a trusted component (TC) is needed that authenticates users and assignes roles for these systems. One such example are cars – especially rental cars – and lorries that grant different users different permissions. The security policy for such a TC is to protect the different end-component credentials from unauthorized access and granting access to authorized users given their roles for said system. The works in this chapter are based upon my contributions to [101] which include most of the general approach and parts of the detailed realization concept.

## 7.1 Background on Role based Access Management

The automotive industry is undergoing a lot of changes. In the future, the need to own a car will become less important and robot taxis [135, 19, 33] an other ride-sharing [7] are replacing individual ownership. In the mean time, car sharing opens the bridge between now and then and also rental cars gain importance.

With the lack of individual ownership the current approach of physical key for accessing and igniting a vehicle does not suit anymore. Instead smartphones are turning into an omnipresent authentication device; even for individually owned cars. The association of rights between phones and cars is changing regularly.

Typically, a gateway for authenticating users will be implemented as a separate Electric Control Unit (ECU). This ECU will perform the authentication checks (in userspace) and grant access and authenticate to other components. As such, the TCB is constructed of the complete device as depicted in Figure 7.1. Opposed to the other examples presented in this work, this use case does not carry any dynamicity on the TCB. As such, a classical interpretation of TCB is sufficient for this solution.

The increase in connectivity and dynamicity introduced with such vehicles comes at the cost of larger attack surfaces and higher complexity. Nevertheless, they are subject to theft or "illegal renting" as well as burglary. Example of these are the attack on Tesla

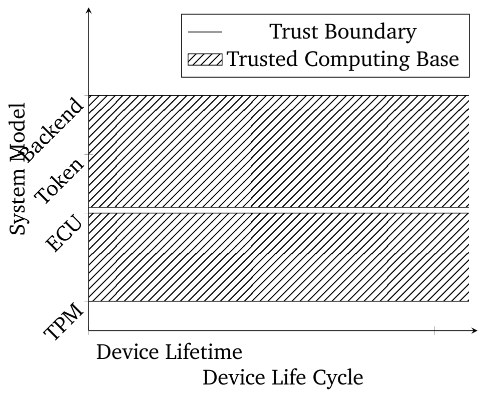


Figure 7.1: Trusted Computing Base for Role-based Trust Gateways without TPM

vehicles due to a weak cipher in the key signal [98].

Other attack vectors include access via the vehicle's sensors [100] followed by reconfiguration of ECU and other systems from this CAN-Bus access [87, 83]. Remotely executed attacks demonstrated that even today's cars can sometimes be altered, steered, unlocked or started without direct access [134, 59, 26, 90, 99, 117].

## 7.2 System and Requirements for Role Based Access Management

This section presents the system model used for this approach, then defines the corresponding adversary model and finally derives the requirements for the described scenario.

### 7.2.1 System Model for Role Based Access Management

The system model for Role Based Access Management is presented in Figure 7.2. It contains the following actors:

- $\mathcal{B}$ : The backend system managing and issuing rights and policies
- $\mathcal{C}$ : The vehicle that is subject to all policies.
- $\mathcal{U}$ : The driver of the vehicle using an authentication token $\mathcal{T}$ to authenticate against the $\mathcal{B}$ and $\mathcal{C}$ .
- $\mathcal{D}$ : Another driver that $\mathcal{U}$ may delegate rights to.

The user token $\mathcal{T}$ contains a signature key used for authentication. It shall be stored inside a shielded location of the token; e.g. a TEE environment inside a smartphone such as TrustZone[8] or other Secure Environment (SE) implementations as proposed in [37].

The car $\mathcal{C}$ contains a special purpose ECU, trusted component (TC) that acts as a gateway for cryptographic schemes and trust domains. This TC includes a TPM that controls the cryptographic keys and credentials for the in-vehicle domain. Without a TPM, then TC's main processor would need to perform then steps of authenticating the user, evaluating authorizations and access rights and communicates with the rest of the car. All this would need to be done on the same part of the system that also hosts the communication stack and a lot of other non-hardened software. With the TPM, these security relevant tasks can be performed by the TPM, whilst the main CPU performs the non-security related communication tasks, thus removing this part from the TCB.

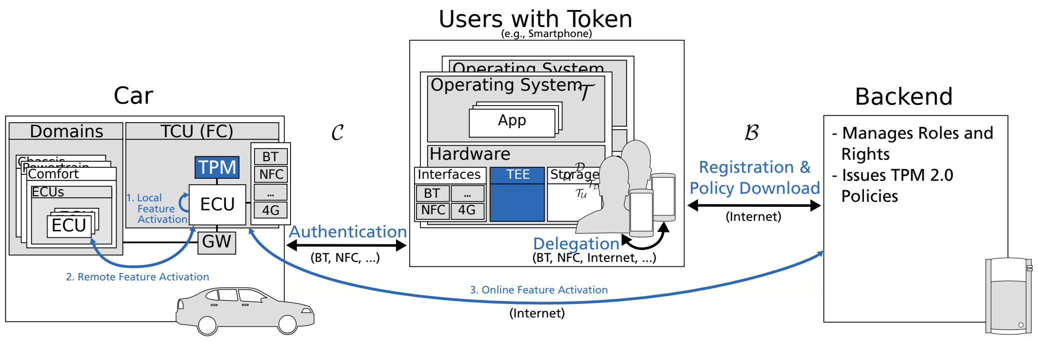


Figure 7.2: System Model [101]

As depicted by the arrows in Figure 7.2, a user $\mathcal{U}$ registers with a backend $\mathcal{B}$. This needs to be associated with a contract between $\mathcal{U}$ and $\mathcal{B}$, but this part is out of scope for this work. The backend $\mathcal{B}$ can then issue authorizations that are either transferred directly to the car $\mathcal{C}$ or indirectly to the car $\mathcal{C}$ via the users token $\mathcal{T}$. The user $\mathcal{U}$ can then authenticate against the car's TC in order to access services of the car $\mathcal{C}$.

### 7.2.2 Use Cases for Role Based Access Management

The system shall support three generic use cases taken from [101]:

- *UC1: Local feature activation* In this use case features of the TC are activated. The TC is the vehicle's Telematic Control Unit (TCU) which is responsible for all external communication and is equipped with a TPM. The TCU is typically more powerful than other ECU in terms of processing power and available memory. A specific instantiation of this use case is the user buying additional map data, e.g., maps of Europe. In this concept all map data would be stored encrypted in the memory of the TC and the user would get access to the respective feature, which in turn unlocks the decryption key inside the TC.

- *UC2: Remote feature activation* In this use case features in ECU are activated that are not the TC. These ECU are connected to the TC through the in-vehicle network such as a CAN bus or Automotive Ethernet. A specific instantiation of this use case could be activating the engine immobilizer and ignition system inside the engine management ECU. In this case, the user authorizes usage of an HMAC key secured by the TPM. The HMAC key is then used in a challenge-response mechanism to answer a challenge sent by the remote ECU. The remote ECU activates its feature when the challenge is answered correctly.
- *UC3: Online feature activation* In this use case a secure communication channel between the car and an external party such as the OEM backend is established. The TPM may store the authentication key that is used to establish the TLS connection. The channel to the OEM backend can be used for several purposes such as securely transmitting updates to the car or synchronizing music playlists.

### 7.2.3 Role Based Access Management Attack Model

The attack model for this solutions uses the cyber-physical Dolev-Yao attacker model [105] that is based on the classical Dolev-Yao model [38]. It assumes that an attacker has full access to all communication channels, being able to eavesdrop, intercept, inject, replay and modify all messages on all networks in the system. Furthermore an attacker has direct physical access to certain devices inside the system.

In this specific case the former especially includes the networks between car, user and backend, but also in in-vehicle networks between the TC and e.g. the engine immobilizer. The latter includes the application CPUs of the TC ECU and the application CPUs of other ECUs.

An attacker however cannot eavesdrop on encrypted or modify authenticated data unless the attacker has knowledge of the cryptographic keys. Furthermore, the attacker cannot access the shielded locations such as the TC's TPM or the token's TEE. The security of the engine immobilizer and backend are out of scope for this solution and thus assumed.

### 7.2.4 Functional Requirements for Role Based Access Management

The following functional requirements are defined in [101]:

- FR1 : Rights management Users in the system may obtain certain rights for a car, e.g., opening the car or use enhanced infotainment features that will be evaluated during authentication (FR3 ). Usage rights are bound to exactly one user. Users shall have the right to delegate their access rights to other registered users through the backend system. Especially delegated access rights may have further restrictions, such as time or usage constraints.
- FR2 : Revocation mechanism It shall be possible to actively revoke previously issued credentials, e.g., in case of a compromised cryptographic key or an authorization expires due to the end of a contract period.
- FR3 : Offline Authentication It shall be possible to access the vehicle and unlock personal features while the system is offline or little to no network access is available, e.g., underground car parks or in remote locations.
- FR4 : Offline rights delegation It shall be possible to delegate rights among different users offline without interaction with the backend. Online right delegation is already part of FR1.

## 7.3 Role Based Access Management Solution

### 7.3.1 Role Based Access Management Concept

Each of the use cases from Section 7.2.2 is based around the usage of a cryptographic key inside the TC. Those may be asymmetric decryption, signing or symmetric HMAC keys. The use case *UC1 local feature activation* typically requires a decryption key. *UC2 remote feature activation* typically refers to in-vehicle authentication e.g. towards the immobilizer. Inside the vehicle symmetric cryptography is dominant and in this concept covered by an HMAC key. *UC3 online feature activation* would typically be based on a TLS client authentication using asymmetric signature operations.

For each feature from any of these use cases that the TC shall govern a separate key is generated, referred to as $K^{\mathcal{C}}_{\mathrm{F_N}}$ . This key is created using the TPM policy TPM2_-PolicyAuthorize command. This policy binds the usage of this key to a second authorization entity for defining policies, the backend $\mathcal{B}$ , identified by the key $K^{\mathcal{B}}_{\mathrm{F_N}}$ . These policies can also be defined after the key has been created.

$$\boxed{K^{\mathcal{C}}_{\mathrm{F_N}}} - \boxed{\text{TPM2\_PolicyAuthorize} \propto K^{\mathcal{B}}_{\mathrm{F_N}}}$$

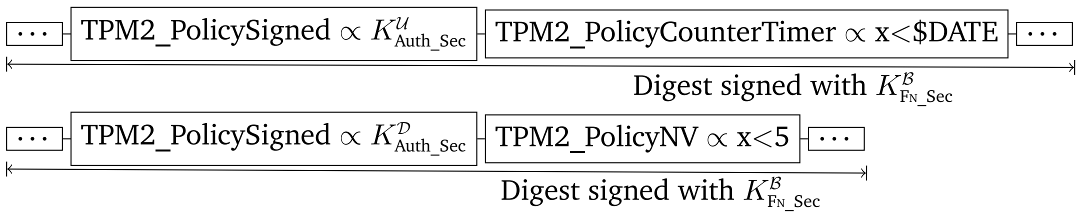
Figure 7.3: Feature key policy [101]

The actual functionalities from Section 7.2.4 to be represented with corresponding policies are signed statements by the backend $\mathcal{B}$ .

### FR1 : Rights management

The basic rights management tasks are implemented as follows:

- *Granting users rights*: In order to grant a user the right to access a feature key, a policy is issued by the backend, that contains a TPM2_PolicySigned command that references the key $K^{\mathcal{U}}_{\text{Auth\_Sec}}$ from the user token $\mathcal{T}$ . The TC's TPM generates a nonce that is signed by the token $\mathcal{T}$ and sent back to the TPM. Then the authorization of said backend issued policy is validated using the TPM2_PolicyAuthorize command.
- *Time constraining users rights* The TPM contains an internal clock. The command TPM2_PolicyCounterTimer can thus be integrated as part of a policy. This requires the TPM's clock to be synchronized with the backend (see e.g. Section 6.3) or the TPM's clock must be synchronized with the UTC time. The corresponding policy is presented at the top of Figure 7.4.
- *Restricting usage counts* Another possibility is to restrict the number of usages of a feature key by a user. The TPM not only supports NV memory for arbitrary data storage, but also certain specialized memory slots. One such way is the type NV_-PIN_PASS with an empty password. Whenever an authentication against such an NV index is performed, a counter is incremented until a threshold value is reached. Thus the backend can include a TPM2_PolicySecret referencing such an index. This of course requires the backend to open up a direct connection to the TPM in order to initialize and configure said NV index.
- *Delegation* The (online) delegation requirement can be accomplished by having the backend issue a policy for the second user's authentication key. Thus, the vehicle needs to be connected to the backend. Note FR4 for offline delegation.



$\cdots$
TPM2_PolicySigned $\propto K^{\mathcal{U}}_{\text{Auth\_Sec}}$
TPM2_PolicyCounterTimer $\propto$ x<\$DATE
$\cdots$
Digest signed with $K^{\mathcal{B}}_{\text{FN\_Sec}}$
$\cdots$
TPM2_PolicySigned $\propto K^{\mathcal{D}}_{\text{Auth\_Sec}}$
TPM2_PolicyNV $\propto$ x<5
$\cdots$
Digest signed with $K^{\mathcal{B}}_{\text{FN\_Sec}}$

Figure 7.4: Basic policies. (1st: Time restriction; 2nd: Revocation) [101]

### FR2 : Revocation mechanism

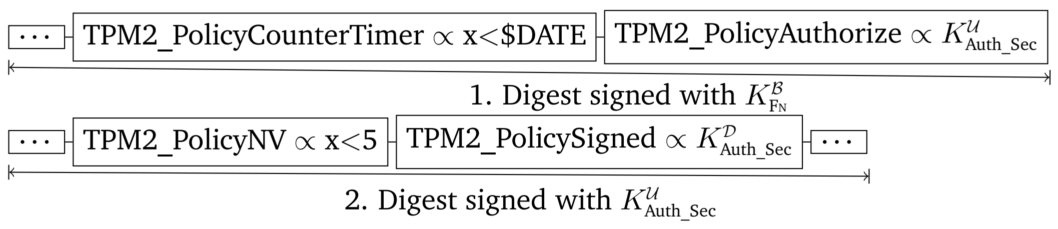
In order to revoke a user's rights, the corresponding policy needs to stop evaluating to true. The best way to do so is to use an NV index of type counter or bitmask on the TPM. A counter can only ever be incremented, a bitmask can bitwise be switched from 0 to 1 but not back. The policy itself can then reference such an index using TPM2_PolicyNV either with a parameter less-or-equal or bit-clear, as shown second in Figure 7.4. Whilst the counter is UINT64 and can be used virtually forever, the NV space for bitmasks is limited. To achieve more flexibility, the mechanisms can be combined and periodically the old bitmasks could be invalidated using a counter and new (compressed) bitmasks without the disabled policies recreated. This step will however require granted policies to be reissued.

### FR3 : Offline Authentication

The capability to perform offline authentication is inherent to the presented approach. The interaction for authentication between the TC's TPM and the user's token consists of a simple challenge-signing-reponse protocol. This can be implemented via WIFI, Bluetooth or NFC. The policies issued by the backend can even be cached and proxied via the user's token if no direct connection between the backend and the TC can be established. The limitation are those policies that require the establishment of NV indexes (of type counter or bitmask). The backend must either have created them previously or only use TPM2_PolicyCounterTimer .

### FR4 : Offline rights delegation

Support for offline delegation can be provided as separate authorization from mere authentication. In order to do so, the backend can authorize a policy containing a TPM2_-PolicyAuthorize element itself. This second TPM2_PolicyAuthorize will reference the key of user $\mathcal{U}$ 's token. The user can then issue any of the aforementioned policies (or also the right for delegation) by performing the same process as the backend usually does. Figure 7.5 provides an example.



...
TPM2_PolicyCounterTimer $\propto$ x<$DATE
TPM2_PolicyAuthorize $\propto K^{\mathcal{U}}_{\text{Auth\_Sec}}$
1. Digest signed with $K^{\mathcal{B}}_{\text{FN}}$
...
TPM2_PolicyNV $\propto$ x<5
TPM2_PolicySigned $\propto K^{\mathcal{D}}_{\text{Auth\_Sec}}$
...
2. Digest signed with $K^{\mathcal{U}}_{\text{Auth\_Sec}}$

Figure 7.5: Offline delegation sub policies for the delegated user. First: Offline delegation policy. Second: User-defined sub policy [101]

## 7.4 Role Based Access Management Prototype and Evaluation

### 7.4.1 Prototype Implementation

#### Hardware

The concept for Role-based Access Management was prototypically implemented in [101] in order to evaluate its performance. The hardware setup is depicted in Figure 7.6. It uses Raspberry PIs for TC and RE running the Rasbian operating system. The TC was connected with an Iridium SLB 9670 TPM 2.0. Two Android smartphones served as user tokens and the authentication key was saved using the Android Keystore that utilizes the phones' TrustZone [36].

Communication between the Android phones and the TC was established using Bluetooth. The in-vehicular network was represented with a physical Ethernet connection and the backend connection was implemented using WiFi.

#### Software

The TC was implemented in C and utilized the TSS2 Feature API (FAPI) [130] for communicating with the TPM. With FAPI, policies can be expressed in JSON. The policy for basic user authentication is depicted in Listing 7.1 and the offline delegation policy is depicted in Listing 7.2.

Listing 7.1: Basic User policy [101]

```
{
    "name": "uid:231432546464, features:3f25664de646a4221e3563f46",
    "policy": [
        {
            "type": "POLICYNV",
            "nvHandle": "/nv/Owner/Counter",
            "operandB": 1,
```

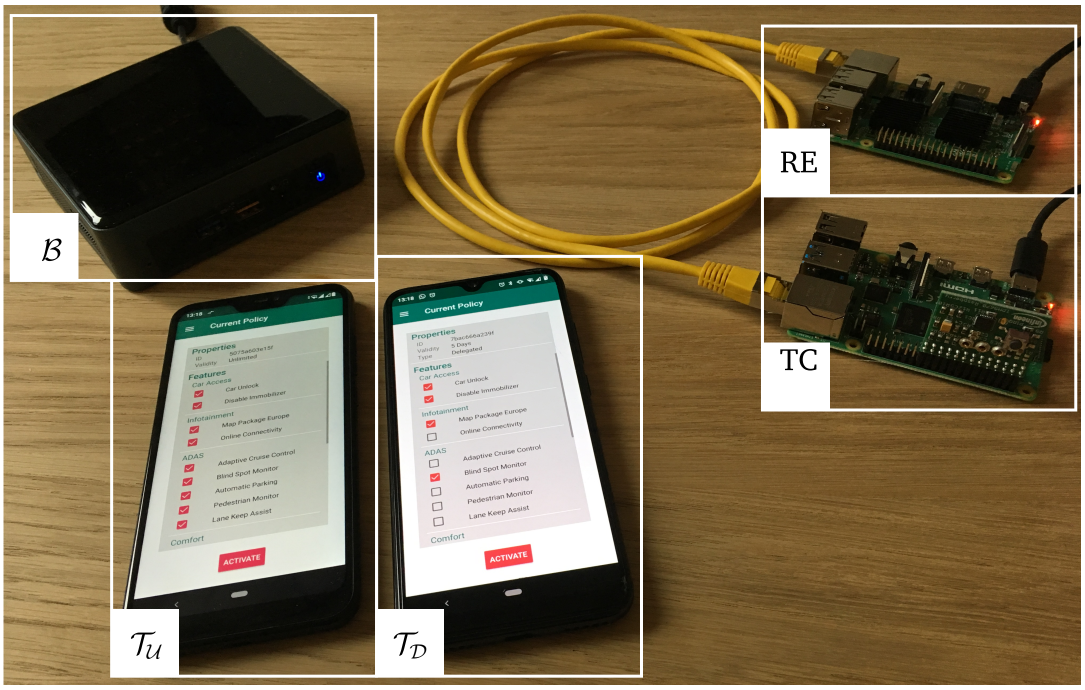


Figure 7.6: Prototype Environment [101]

```
            "offset": 0,
            "operation": "LT"
        },{
            "type": "POLICYCOUNTERTIMER",
            "operandB": 5000,
            "offset": "0",
            "operation": "LT"
        },{
            "type": "POLICYSIGNED",
            "keyPEM": "-----BEGIN PUBLIC KEY-----MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8
                AMIIBCgKCAQEAoX8ZsAAArspY3wmMpkgrviE8OMfxNypfMrWb6SJJSh/AXjAnwyTYswUT6
                T5sj1qoQkxjcWmEGHmsd3kOfEEcYg84I7eu0wY1N+tBfaFuCEKdQxkeIAmsf1xVt2WYi3
                WLn4VYIMC74SfDQXglhrSd1f9PlITq8RgLI933Bh9sUMjv9MIe+uWNkn5dtGCWZ4
                DzLpULF6WgcFyMlsAqqUOYmviwIRV7jrtvDrRerS7ChCS4dWRwNpD26Mw/XsDHQWnH8FL0
                dtZUcYXSyQQ/NZkq+GX4PRrh/YeScG80YbTr6GVSU/WFxpHVVXuk6EzqThThcm+J1
                GzsLVrfm9QSSh2rxQIDAQAB-----END PUBLIC KEY-----",
            "keyPEMhashAlg": "SHA256"
        },{
            "type": "POLICYAUTHORIZE",
            "keyPath": "/ext/LoadedBackendKey",
        }
```

```
    ]
}
```

Listing 7.2: Offline delegation policy and sub-policy [101]

```
{
    "name": "uid:231432546464, features:f425da4de224a4221e35eaf46",
    "policy": [
        {
            "type": "POLICYCOUNTERTIMER",
            "operandB": 5000,
            "offset": "0",
            "operation": "LT"
        },{
            "type": "POLICYSIGNED",
            "keyPEM": "-----BEGIN PUBLIC KEY-----MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8
                AMIIBCgKCAQEAoX8ZsAAArspY3wmMpkgrviE8OMfxNypfMrWb6SJJSh/AXjAnwyTYswUT6
                T5sj1qoQkxjcWmEGHmsd3kOfEEcYg84I7eu0wY1N+tBfaFuCEKdQxkeIAmsf1xVt2WYi3
                WLn4VYIMC74SfDQXglhrSd1f9PlITq8RgLI933Bh9sUMjv9MIe+uWNkn5dtGCWZ4
                DzLpULF6WgcFyMlsAqqUOYmviwIRV7jrtvDrRerS7ChCS4dWRwNpD26Mw/XsDHQWnH8FL0
                dtZUcYXSyQQ/NZkq+GX4PRrh/YeScG80YbTr6GVSU/WFxpHVVXuk6EzqThThcm+J1
                GzsLVrfm9QSSh2rxQIDAQAB-----END PUBLIC KEY-----",
            "keyPEMhashAlg": "SHA256"
        },{
            "type": "POLICYAUTHORIZE",
            "keyPath": "/ext/LoadedBackendKey",
        }
    ]
}
```

### Evaluation Setup

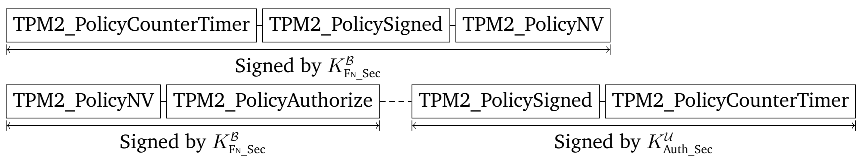

TPM2_PolicyCounterTimer
TPM2_PolicySigned
TPM2_PolicyNV
Signed by $K^{\mathcal{B}}_{\text{FN\_Sec}}$
TPM2_PolicyNV
TPM2_PolicyAuthorize
TPM2_PolicySigned
TPM2_PolicyCounterTimer
Signed by $K^{\mathcal{B}}_{\text{FN\_Sec}}$
Signed by $K^{\mathcal{U}}_{\text{Auth\_Sec}}$

Figure 7.7: Basic Policies used for performance evaluation. First is a user policy and second is a offline delegation policy consisting of two paths: the backend signed path and the user signed path. [101]

The performance evaluation of [101] used the policies shown in Figure 7.7. The first policy authenticates the user via signature, checks an NV counter for revocation and checks the policy for expiration. The second policy uses offline delegation with the delegated policy checking for the user authentication and expiration.

### 7.4.2 Network Utilization

| User Type | Data (Size [Byte]) | Total Size [Byte] |
|---|---|---|
| $\mathcal{U}$ | $p_N$ (1066), $\sigma_{p_N}$ (256), $K^{\mathcal{U}}_{\text{Auth\_Pub}}$ (451), $K^{\mathcal{B}}_{\text{FN\_Pub}}$ (451), $N_{\text{TPM}}$ (32), $\sigma_{N_{\text{TPM}}}$ (256) | 2512 |
| $\mathcal{D}$ | $p_{N_D}$ (371), $\sigma_{p_{N_D}}$ (256), $p_{N_{\text{SUB}}}$ (895), $\sigma_{p_{N_{\text{SUB}}}}$ (256), $K^{\mathcal{D}}_{\text{Auth\_Pub}}$ (451), $K^{\mathcal{U}}_{\text{Auth\_Pub}}$ (451), $K^{\mathcal{B}}_{\text{FN\_Pub}}$ (451), $N_{\text{TPM}}$ (32), $\sigma_{N_{\text{TPM}}}$ (256) | 3419 |

Table 7.1: Transmitted message sizes for authentication protocol [101]

In Table 7.1 the sizes of the different message elements are illustrated. The user authentication messages totaled in 2512 bytes. The delegation use case totaled in 3419 bytes. Given the bandwidth available in Bluetooth, these numbers seem insignificant.

### 7.4.3 Execution Time

The same policies were evaluated regarding their execution times, averaged over 1000 measurements. The values are presented in Table 7.2. Execution times of 3 to 4 seconds seem very high. They could be optimized by a more low-level implementation using TSS2 ESYS [132] instead of FAPI. Also upcoming versions of TPMs are expected to come with acceleration for Big Numbers and Hash algorithms, the two main components in this approach.

| User Type | BT Discovery [ms] | Policy Processing [ms] | Overall [ms] |
|---|---|---|---|
| $\mathcal{U}$ | 90,83 ($\pm$89,39) | 2979,56 ($\pm$155,63) | 3070,40 ($\pm$92,19) |
| $\mathcal{D}$ | 85,91 ($\pm$97,67) | 3957,83 ($\pm$136,60) | 4043,75 ($\pm$105,18) |

Table 7.2: Execution time of the basic user and a basic delegation policy [101]

## 7.5 Conclusion to Role Based Access Management

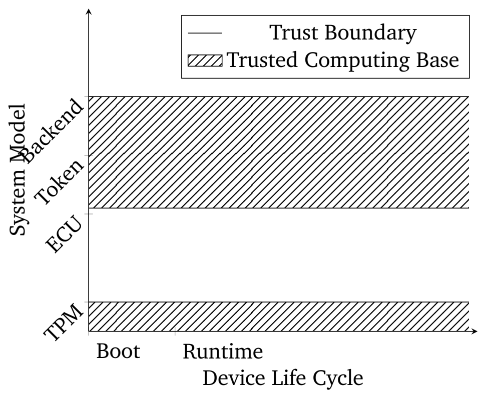


Figure 7.8: Trusted Computing Base for Role-based Trust Gateways with TPM

The proposed feature activation system for automotive scenarios utilizes TPM 2.0 inherent policy constructs to authorize key usage within the car. The system allows users to delegate their access rights both online and offline and even allows the setting of custom restrictions during both as well as actively revoked authorizations during online delegation. In this system, the TPM 2.0 is used as the endpoint of a secured communication channel between the backend and the user's authentication token, e.g., smartphone. The authentication of the user combined with the current policies grant access to symmetric keys of the TPM 2.0. These keys allow for the integration with existing in-vehicle networks and security solutions with minimal integration effort. The corresponding TCB is depicted in Figure 7.8. The application CPU of the ECU could be removed from the TCB. Since this part contains a lot of connectivity code in order to establish Bluetooth and Internet connections, this is a significant part of the codebase that could be removed from the TCB, making the system much more secure.

# 8 Engineering Systems

The example in Section 2.2 / Figure 2.3 shows how different security policies can be related to each other. This is especially useful during the security design phase of a system in order to minimize and determine the Trusted Computing Base (TCB) of a system.

This chapter introduces a consistent methodology for designing secure systems during this specification phase. The Security Modeling Framework SeMF serves as basis for its security vocabulary. The SeMF is extended by the concept of SeMF Building Blocks SeBBs as reasoning tool and provides a security design process utilizing them as refinement artifacts. This process guides the decision making during the system specification phase focused on the security aspects and integrates with refinement driven functional engineering processes in order to minimize and strengthen TCBs.

This methodology is applied to an example that illustrates its principle process. This example consists of a generalized use cases for displaying sensor readings, such as a vehicles speed, on a user-facing display, the tachometer. Though this use case appear more basic, it very nicely introduces the concepts of authenticity and confidentiality and how these can be related with each other through cryptographic operations. Thus, the results can then be applied to more complex scenarios alike.

This chapter builds upon and reuses text from my contributions in [107, 52] to which I was the main contributor. It is further based on my contributions to [71, 44, 45, 46, 39, 44].

## 8.1 Background on Engineering Systems

The aspect of security in the engineering process of information and communication technology systems gains increasing attention. The efforts of Microsoft's Security Development Lifecycle [89] or IBM's Secure Engineering Framework [76] are examples of industry's increased awareness. The necessity to improve development processes with respect to security at least partly results from the apparent growth in damage above the 100 million US dollar mark [102] as well as from the increase in attacker investment. Targeted attacks today can rely on funding of up to several million dollars [123].

Remarkably, the sole awareness for systems' security has led to an increase of quality of security in systems in recent years. Developer training, secure coding guidelines and best-practice patterns, code reviews and scanners or even code validation increase the code quality with regard to its security aspects. However, these approaches mostly target the correct implementation of a given specification and they are usually not well-structured and also do not support a more-or-less complete coverage and documentation of security-related issues.

Security would further strongly benefit from a proper secure engineering process extending various existing development processes. This process would need to start with security requirements elicitation at the earliest phase of the process and keep track of requirements and refine them throughout the different engineering steps. Security design decisions then also need to take into account possible attacks and threats and risks associated with them. In principal, such a process seems to be straightforward and some established techniques already implement parts of it.

SDL's STRIDE for example is one useful approach that is focused on a rather low level of abstraction and is oriented towards concrete (security) technologies. Other approaches address only particular security issues. One example is SecureUML [86] which focuses on access control modelling. In spite of the large amount of work on IT security, a generic basis for a more general approach towards secure engineering is still missing. Various formal models for security either concentrate on particular types of properties (e.g. information flow [34]) or on special areas (e.g. cryptographic protocols [1]). Other approaches require the construction of concrete adversary models [30].

In contrast, the Security Modeling Framework SeMF provides one generic framework for the specification and refinement of all security properties and will serve as a basis for the security design engineering process.

With increasingly complex and critical systems, it is however necessary to consider security from the very beginning throughout the whole engineering process in a precise, consistent and integrated manner, rather than a distinct engineering step or a separated set of functionalities.

Based on previous work on formal notions for security, combined in the Security Modeling Framework SeMF [66] and a requirements elicitation approach [50, 51] the contribution of this chapter consists of the extension of SeMF through so called SeMF Building Blocks (SeBB), that allow to encapsulate reasoning steps in a comprehensive manner, such that SeMF may be applied by non-experts.

The main contribution of this chapter lies within the provisioning of a security design engineering approach for the System Design/Specification phase in refinement driven engineering methodologies, as depicted in Figure 8.1. The purpose of this security design is to guide the functional engineering, prevent insecure designs at an early stage and

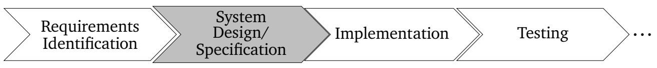


Figure 8.1: Focus within the Engineering Process

assist in design choices to be made. The resulting security concept documents security design decisions and the documented residual assumptions can serve as a basis for risk assessment and gives directions and testable conditions for the implementation or issue organizational security means.

## 8.2 Security Statements in SeMF

The Security Modelling Framework SeMF provides a formal language based semantics for expressing security statements in a comprehensible way. Through these means, it does not only provide an axiomatic framework, but rather an underlying reasoning framework that allows for unambiguous security statements. Further, SeMF distinguishes itself from other formal frameworks by providing some characteristics that make it especially suitable for application in the context of security engineering. The two most important aspects are carved out and motivated in this Section.

### 8.2.1 Constraint-Based Positive Security Statements

A typical way of expressing security statements is to formulate abuse-cases or anti-goals constructively. Formally or informally, the constructive expression is given through expressing attacker capabilities to e.g. deduce a secret from a message observation. The safety condition is, that the attacker's deduction action must not happen. These approaches attempt to describe the set of *Security Failures*. However, incompleteness of this set may lead to the implementations of possible security leaks into the system's specification.

Typically, *attacker centric* constructive reasoning seems more intuitive due to the ease of construction over constraining. Considering the (typical) case of incompleteness of Security Assumptions vs. Possible Attacks, a constraining approach based on assumptions is more robust: *It is better if a system meets its security requirements though some provided guarantees were not considered, than a system (supposedly) meets its security requirements though some possible attacks were not considered*. Similarly, the failure case for a constraining approach is the false statement of an assumptions whilst the failure case for constructive approaches is the false omission of an attack. Again, the constraining approach is more robust: *It* *is easier to identify a false statement that was written down, than to identify an omitted statement that was mistakenly not written down.*

One possibility to describe positive statements would be to take the desired functionality of the system as a basis. This however restricts the functional aspects of the engineering process too much to be practicable. Instead SeMF expresses the set of positive security statements as *constraints* against the system. This way, the security aspects of a system can be investigated separately from other aspects (functional, resources, usability, ...) which reduces overall complexity, whilst providing guidance to the functional engineering of the system.

### 8.2.2 Early and Expressive Statements

Another important factor for a comprehensive vocabulary for security statements is the expressiveness of the language. Often, reasoning about confidentiality is expressed as "Some malicious agent must not know asset X". Though this may appear simple at first sight, such a property implicitly requires statements about:

- What is the asset and where is it stored and/or processed?
- How can attackers observe, access or conclude it?
- Which are forbidden states, representing attacker knowledge of the asset?

As can be seen, the simple security goal expands to a more complicated set of questions when applied to an actual system. Typically, these questions are represented implicitly and constructively within the system behavior, rather than the property or action abstractions, which makes errors hard to detect.

SeMF on the other hand expresses confidentiality directly regarding the first two questions as part of the property itself. Further, it utilizes some special concepts in order to extend the system behavior explicitly.

- Actions to learn from
- Agents' Local View
- Agents' Initial Knowledge

With these the above questions are not addressed implicitly within the system behavior description, but explicitly expressed within the confidentiality property notation. Further, as this approach is capable of modeling agents' reasoning capabilities through their Initial

Knowledge, confidentiality can be expressed as constraint-based property and constructive statements of forbidden states are not necessary. Finally, it is also possible to specify more detailed properties, such as unauthorized agents' allowed knowledge about relations between the appearance of assets (e.g. "How often does a confidential asset change ?"). More details about this confidentiality property can be found in [67], [65].

Similarly, the notation for *authenticity* from [65] can be used to express explicitly *which agent* shall be *when* assured about *which* event – such as the sending or storing of data. In [46], there is also a notion of representing *Trust* which refers to the *trusting agent* and a *trusted property*. This notion is capable of expressing trust in the global system as well as explicit and specific properties of e.g. other trusted agents – such as the concepts of Trusted Third Parties and Trusted Computing.

### 8.2.3 Security Properties in the Security Modelling Framework SeMF

Security properties in SeMF follow these ideas of constraint based positive, early and expressive statements. The current syntactical representation is in the format of property predicates. A the subset of properties to be used within this chapter's example can informally described as follows: $precede(a, b)$ holds if whenever action $b$ occurs, action $a$ must have occurred (before) as well. $auth(a, b, P)$ holds if whenever action $b$ occurs, action $a$ must have happened authentically for agent $P$, according to $P$'s *local view* and *initial knowledge*. $conf(A, p, M, W)$ holds if in the occurrences of actions $A$ the parameter $p$ is indistinguishable (in terms of (L,M)-completeness [67]) from all elements in $M$ for every agent except those in the set $W$. $trust(P, prop)$ holds if agent $P$ has the trust that a property $prop$ holds according to its initial knowledge.

More information can be found in [67, 66, 64, 47].

## 8.3 Designing Security Policy Systems

Beyond the utilization of SeMF property predicates only as expressive and comprehensive vocabulary, their semantical definitions can be utilized for reasoning, refining and designing a secure system.

### 8.3.1 Reasoning about Security Design

During the design process, the engineer is challenged with the task of designing a system that fulfills a given set of requirements. For functional engineering a given requirement can often be fulfilled choosing and associating a mechanism that provides the required

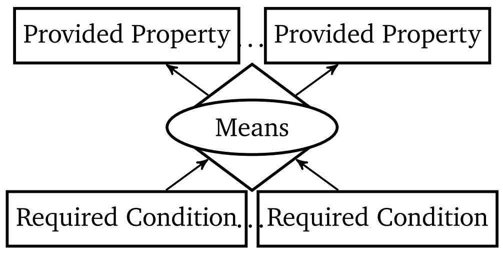


Figure 8.2: Structure of a Security Building Block

functionality. In the case of security however, it is usually not possible for a mechanism to directly fulfill a given requirement. Rather, security mechanisms introduce additional (internal) conditions, that need to be addressed, in order for the mechanism to provide its security properties. A similar but more generic understanding is used in Arguing Security [70].

An example for such conditions comes from RSA signatures that require the private key to be confidential to the signer in order to provide authenticity of a message. In case of HMAC, there are two internal conditions: The shared secret must be confidential to sender and receiver and the receiver must not have signed the message itself (e.g. in a different protocol step). *A security mechanism usually does not provide Security Properties unconditionally. Rather, it may only "transform" one property into another.* This circumstance is represented with the concept of Mechanism-based SeMF Building Blocks (M-SeBBs).

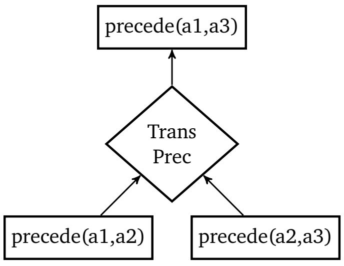
Sometimes however there may not be a mechanism that directly fulfills a given requirement. Instead an engineer has to refine a requirement to a set of less complex requirements, that each can be fulfilled by a mechanism. For these cases, the notion of F-SeBBs (or Formally based SeBBs) is used.

In order to support a comprehensible understanding SeBBs provide a graphical format that allows for easy comprehension even for non-experts. The details of included formal proofs are completely transparent to the user that only needs to understand the meaning of SeMF property predicates.

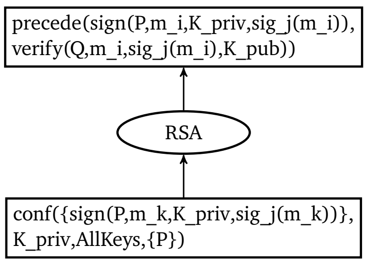
### 8.3.2 SeMF Building Blocks

Formally, SeMF Building Blocks (SeBBs) encapsulate theorems of implications within the underlying formal framework SeMF and its properties' semantics.

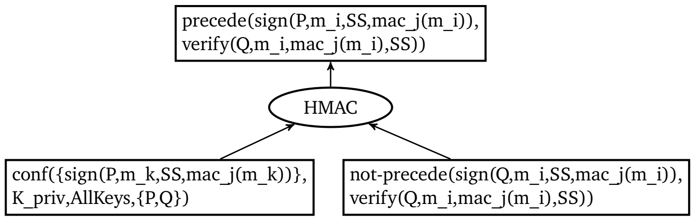
A SeMF Building Block consists of three layers, as illustrated in Figure 8.2:



(a) F-SeBB Precede Transitivity


precede(sign(P,m_i,K_priv,sig_j(m_i)), verify(Q,m_i,sig_j(m_i),K_pub))
RSA
conf({sign(P,m_k,K_priv,sig_j(m_k))}, K_priv,AllKeys,{P})

(b) M-SeBB: RSA-Signature


precede(sign(P,m_i,SS,mac_j(m_i)), verify(Q,m_i,mac_j(m_i),SS))
HMAC
conf({sign(P,m_k,SS,mac_j(m_k))}, K_priv,AllKeys,{P,Q})
not-precede(sign(Q,m_i,SS,mac_j(m_i)), verify(Q,m_i,mac_j(m_i),SS))

(c) M-SeBB: HMAC

Figure 8.3: Examples of SeMF Building Blocks

- The *provided properties* of a Security Building Block represent the security properties that the Security Building Block provides for the overall system.
- The *means* represents the mechanism or instrument by which the external properties can be achieved. These can be based on a proven theorem related to specific properties of the security requirements, or on expert knowledge related to the mechanism the SeBB addresses.
- The *required conditions* are those properties that must be fulfilled in order for the mechanism to provide the external properties.

These implications represented by a SeMF Building Block have to be formally proven, according to the means that they represent. These can be categorized into the two different classes of SeBBs mentioned before:

**F-SeBBs** or Formal SeBBs represent implications that originate from the semantics of security properties themselves as defined *within SeMF* allowing to relate them and proof these relations for all possible system instances. There are simple ones, such as that if something is confidential regarding (to be known only by) Bob, then this implies directly that this asset is confidential regarding Bob and Alice. But also more complex ones exist, such as the relating of trust assumptions with authenticity. The specialty with these properties is that they can be proven within SeMF independent of any specific system.

**M-SeBBs** or Mechanism SeBBs in contrast cannot be proven completely internally to SeMF. They represent implications originating from security mechanisms like protocols and cryptographic functions and require assumptions that originate from evidence or expert knowledge that is external to SeMF.

In order to visualize the differences between means originating from within SeMF and those that require external assumptions, the former are represented as diamonds, whilst the latter are represented as circles (compare Figure 8.3). Exemplarily, the theorem and proof for the F-SeBB Transitive Precede as presented in Figure 8.3a are shown:

**Theorem 1.** *Given system $S$ with properties $precede(a, b)$ and $precede(b, c)$ holding in $S$, the property $precede(a, c)$ holds in $S$ as well.*

Informally, this theorem can be argued as: If $b$ always precedes $c$ and $a$ always precedes $b$, then $a$ also always precedes $c$. In term of the SeMF semantics, the theorem can be proven as:

*Proof.* $precede(b, c)$ states that $\forall \omega \in B$ with $c \in alph(\omega)$ it holds that $b \in alph(\omega)$. Due to $precede(a, b)$ it is known that for these $\omega$ with $b \in alph(\omega)$, $a \in alph(\omega)$ as well, hence $precede(a, c)$ holds in $S$. $\square$

For M-SeBBs similar proofs have to be provided as well. They however include additional assumptions about the mechanisms. [46] gives an example on proving the example M-SeBB for RSA signatures in Figure 8.3b.

Note that all F-SeBBs can also be applied for trusted properties, as they hold for all possible systems, including an agent's knowledge system. M-SeBB usually will also be applicable regarding trusted properties, except when a certain agent does not trust a security mechanism. Within a given engineering environment, a global library can be established that provides engineers with a set of SeBBs authorized by the security experts of this domain or company.

## 8.4 The Security Design Process

Utilizing the concepts of SeMF property predicates and SeBBs, leads to a process for security engineering that interfaces with refinement driven functional engineering processes. Figure 8.4 illustrates the structure of concepts that are utilized in this approach. This approach integrates with the functional engineering process in the Design phase.

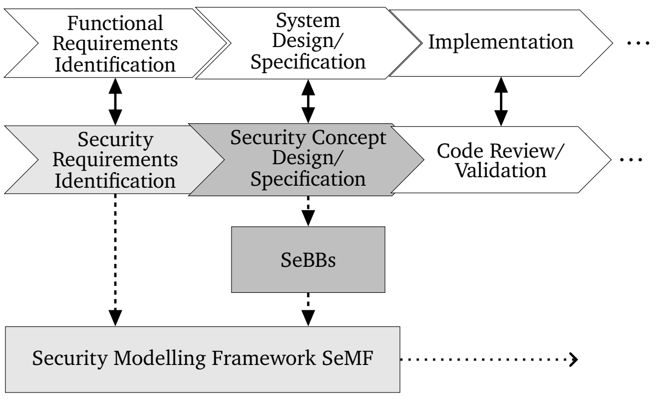

Functional Requirements Identification
System Design/ Specification
Implementation
...
Security Requirements Identification
Security Concept Design/ Specification
Code Review/ Validation
...
SeBBs
Security Modelling Framework SeMF

Figure 8.4: Structure of Progress Interaction

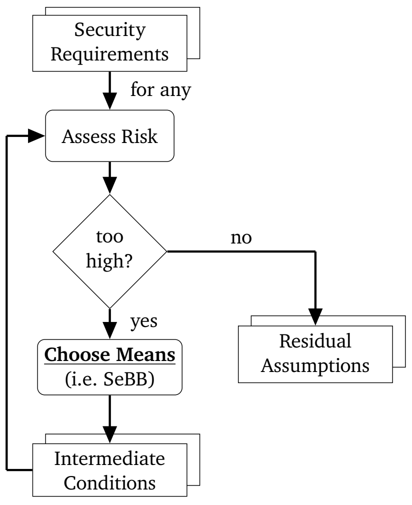
During the **requirements analysis** phase one may utilize approaches for systematic identification of formally defined security requirements such as [50]. The important aspect is that security requirements need to be expressed in the formally defined terminology of SeMF.

During the **design and specification** phase the functional engineering concept of refinement driven design decisions can be followed. The semantic clarity of SeMF properties and SeBBs can be leveraged. During all these steps, the focus lies within positive constrained-based security properties and follows the concept of conditional security. The well-understandable property predicates and SeBBs make the execution of these steps possible even for non-experts in formal methods and/or security. This provides an interface to a continuous risk assessment in order to judge on the fulfillment of conditions.

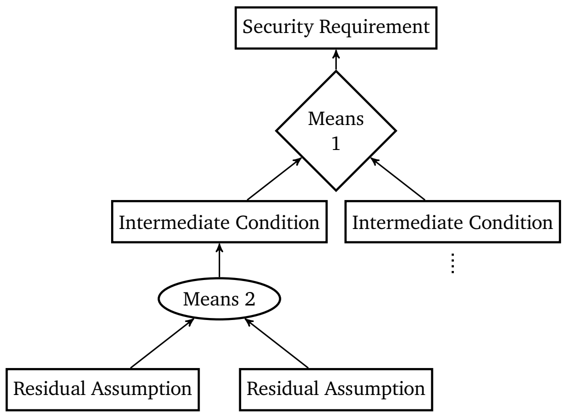
### 8.4.1 Support Security Design Decisions

The security design decisions should typically be performed in parallel to the functional design and specification phase. Here the concept of SeMF Building Blocks is used which encapsulate security mechanisms and requirement refinement rules in a comprehensible way.

For the security design aspects, the process presented in Figure 8.5a is used. Taking a security requirement from the set of collected requirements, in a first step the risk associated with the requirements is assessed (by some methodology). If it is low enough, the requirement does not need to be further addressed and may be added to the set of residual assumptions for documentation, traceability and residual risk assessment purposes.



(a) Security Design Decision Step



(b) SeBB-Trace of Design Decisions

Figure 8.5: Design Decisions Process and Documentation

The sets can further serve as guidance for implementation, code review, validation and testing tasks and cover organizational measures to be taken during deployment.

If the risk associated with a requirement is too high to disregard it, the engineer needs to decide on a mechanism that fulfills this requirement. As there are internal conditions associated with each SeBB, they serve as a direct guidance for the decision making. Once a SeBB is chosen, it can be integrated and its internal conditions will need processed in a later step as well.

Some M-SeBBs refer to specific actions in the system model. These actions may represent cryptographic operations related to an M-SeBB. If these actions do not exist, but the SeBB shall be applied, then the system model itself needs to be refined and the according agents extended to include these operations.

The repeated execution of these decision steps can form a graph of SeBBs and intermediate properties that lead from residual assumptions to the global security requirements as depicted in Figure 8.5b as well as a solution oriented refinement of the system model itself. The graph represents a formal proof of the fulfillment of the system's security requirements, given the set of residual assumptions to hold in the system.

### 8.4.2 Resulting Benefits

By following the above engineering method not only does the engineer have a guidance during the security design of the system, but gains further benefits from the structured approach and tools.

#### Traceability

SeBB graphs provide documentation that can be used during development, providing backtracing if design decisions need to be reverted. But they also serve as an input to the “design-rational” of a given system.

#### Support for Design Patterns

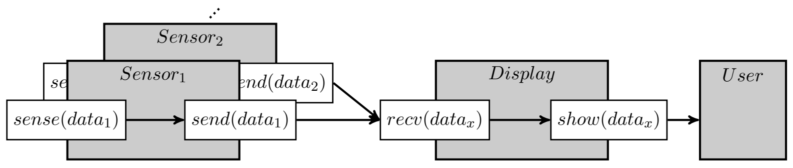
With this kind of traceability included into the method, the approach is suited for inclusion in pattern driven designs. A first work has been presented in [72], where such a traceability graph is provided alongside with patterns’ functional descriptions. Further, the existence of a repository of design and implementation patterns alongside with a library of SeBBs form a good combination to provide company-wide standards for specifications and implementations.

#### Complexity Reduction

As opposed to attack trees, the SeBB based traceability graphs do not necessarily have the format of trees. At some points, intermediate conditions can converge, e.g. a cryptographic key store may serve for several private keys at the same time. This allows for the convergence of security means in order to reduce system size and reuse already specified components (such as such a cryptographic key store).

#### Risk Assessment

The Traceability graphs can serve as a basis for risk assessment. Further, it is also possible to identify the most valuable intermediate conditions and residual assumptions based on their convergence. For the task of overall risk assessment, of course, attacker capabilities need to be estimated and the residual assumptions need to be mapped to attacker models. However, these attacker models are now more technically focused and less complex as in attack trees directly targeting global security requirements.



Requirement: $auth(sense(Sensor_x, data_x), show(Display, data_x), User)$

Figure 8.6: Example Requirement System

### Abstraction Level Flexibility

As opposed to other approaches that are focused on specific levels of abstraction, the approach is flexible enough to be applicable on abstract levels as well as concrete levels separately, as well as all the way from e.g. organizational to technical system design.

## 8.5 Exemplary Application of SeMF Engineering

In order to illustrate this approach an example of a display that shows a set of sensor values to a user is depicted in Figure 8.6. This could be the velocity and temperature sensors of a car, or the pressure and temperature sensors of a power-plant. For ease of readability, the graphical representation will not be used but the SeBBs will be represented in a textual description.

The input to the design phase of the engineering process is the model illustrated by Figure 8.6. Note that the arrows represent the functional relations and do not have any security implications. Regarding security the only information after the requirements elicitation phase is the requirement of the user to be assured that data on the display was authentically measured by the corresponding sensor:

$$auth(sense(Sensor_x, data_x), show(Display, data_x), User) \tag{Req}$$

First, this requirement can be transformed using a SeBB representing the relation of *authenticity* and *trust in precedence* to the authenticity of data shown on the display to actually be this display for the user and the user's trust in the precedence of the sensing of data by the respective sensor before showing it on the display:

$$trust(User, precede( \quad sense(Sensor_x, data_x), show(Display, data_x)))\wedge \tag{8.1}$$

$$auth(show(Display, data_x), show(Display, data_x), User) \tag{8.2}$$

Property (8.2) representing the correct recognition of the display may be solved through organizational means and is therefore added to the set of residual assumptions. Property (8.1) becomes the next requirement for this system.

Next, the engineer decides to solve the new requirement by adding a signature scheme. As the engineer has not yet decided to use HMAC, RSA or ECDSA, the engineer refines the system to add generic functions $sign(Sensor_x, data_x, sig(data_x))$ and $verify(Display, data_x, sig(data_x))$ to the system.

Now Property (8.1) can be refined to make use of these new actions from the signature scheme, using the *SeBB transitive precede* of Figure 8.3a twice. As mentioned before all F-SeBBs and most M-SeBBs can be applied to the trusted properties as well. This results in the division of the user's trust into three properties, where the first one refers to the sensors internal behavior, the second one represents the exact requirement against to to be decided signature scheme, and the third describes the requirements against the display:

$$\begin{aligned} trust(User,&precede( \\ &sense(Sensor_x, data_x), \\ &sign(Sensor_x, data_x, sig(data_x)))\wedge \end{aligned} \tag{8.3}$$

$$\begin{aligned} trust(User,&precede(, \\ &sign(Sensor_x, data_x, sig(data_x)) \\ &verify(Display, data_x, sig(data_x))))\wedge \end{aligned} \tag{8.4}$$

$$\begin{aligned} trust(User,&precede( \\ &verify(Display, data_x, sig(data_x)), \\ &show(Display, data_x))) \end{aligned} \tag{8.5}$$

Property (8.4), a precedence between a sign and a verify can be solved with both, the *SeBB RSA* and the *SeBB HMAC* of Figures 8.3b and 8.3c. The engineer now has to decide which direction to take for the next refinement.

From *SeBB HMAC* the engineer can see, that he would need to provide a way to distribute a shared secret $SS$ confidentially to the Sensors and the Display. Instead, he decides to choose RSA. He refines the system by enriching the $sign$ and $verify$ actions according to the actions in the *SeBB RSA*'s provided property.

The application of *SeBB RSA* to Property (8.4) leads to the new requirement of the user in the confidentiality of each of the sensors' private keys respectively.

$$\begin{aligned}&trust(User, conf(\{sign(Sensor_x, data_x, K_x^{priv},\\&sig_x(data_x))\}, K_x^{priv}, AllKeys, \{Sensor_x\}))\end{aligned} \tag{8.6}$$

The engineer could also choose to use the same private key for both sensors. This however would have raised the issue that $data_1$ could be mistaken as $data_2$ leading to attack of providing temperature information as velocity in a car, unless data was tagged. Note that this approach captures these kinds of mistakes.

The set of open security requirements after this step consists of Property (8.3), Property (8.5) and Property (8.6).

In order to guarantee the confidentiality of the sensors' private keys as is Property (8.6), the engineer decides to include a smartcard or TPM within the sensors. Accordingly, he has to split the sensor further into two parts, one for the $sensing$ and sending, and one for $signing$ operations. As a preparation, the engineer adds the corresponding interface action $cmd\_sign$ and $return\_sign$ to the sensor agent. Then, the engineer can refine Property (8.3) with the *F-SeBB transitive precede* into the user's trust in the precedence in the future $sensing$ part as well as the $signing$ part:

$$trust(User, precede(sense(Sensor_x, data_x), cmd\_sign(Sensor_x, data_x)))\wedge \tag{8.7}$$

$$trust(User, precede(cmd\_sign(Sensor_x, data_x), sign(Sensor_x, data_x, K_x^{priv}, sig_x(data_x)))) \tag{8.8}$$

After performing this split operation, each of the sensors' actions is assigned to either $Sensor_x$-$Sensing$ or $Sensor_x$-$Signing$, Property (8.6) contains both of them in place of the former $Sensor$ agent. This set can be further reduced, using a *F-SeBB confidential agent inclusion* because if two agents are allowed to know a secret, then this is trivially fulfilled, if only one of them (the $Signing$ part) does know the secret:

$$\begin{aligned}&trust(User, conf(\{sign(Sensor_x\text{-}Signing, data_x, K_x^{priv},\\&sig_x(data_x))\}, K_x^{priv}, AllKeys, \{Sensor_x\text{-}Signing\}))\end{aligned} \tag{8.9}$$

At this point, the engineer could decide to use a Trusted Platform Module (TPM) as Signing component to solve Property (8.9) and Property (8.8) with TPM-specific conditions

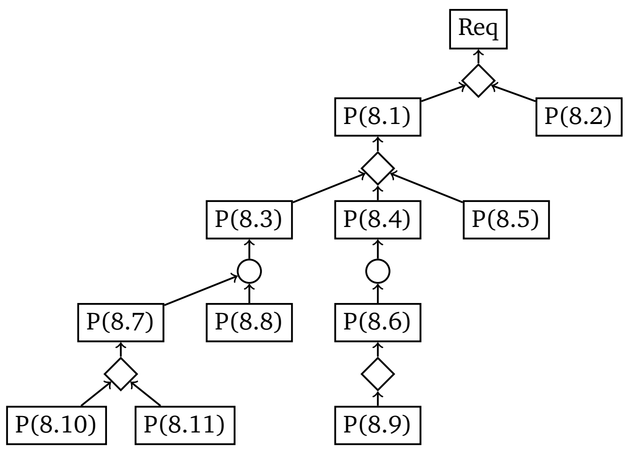


Figure 8.7: SeBB-graph of example refinement

about the certificate management and trust relations to realize. Further, a TPM could be used to fulfill Property (8.7), using TPMs' capability to restrict the usage of keys, to a certain software configurations through PCRs [61, 127]. Using the presented approach, this can be modeled by a *SeBB PCR-Restriction* that would satisfy Property (8.7) given the properties:

$$\begin{aligned}
&trust(User, precede\text{wi}phase( && (8.10)\\
&\quad sense(Sensor_x, data_x),\\
&\quad cmd\_sign(Sensor_x, data_x),\\
&\quad changeconfig(Sensor_x, config_{good}),\\
&\quad changeconfig(Sensor_x, config_y)))\wedge\\
&trust(User, nothappens\text{wi}phase( && (8.11)\\
&\quad sign(Sensor_x, data_x, K_x^{priv}, sig_x(data_x)),\\
&\quad changeconfig(Sensor_x, config_{\neg good}),\\
&\quad changeconfig(Sensor_x, config_y)))
\end{aligned}$$

The resulting SeBB-graph that can be used for the Security Rational, Traceability and Risk Assessment is presented in Figure 8.7.

## 8.6 Conclusions to Engineering Systems

The presented approach provides a usable security design engineering process based on formal models with a formal semantics and utilizes the new notion of SeMF Building Blocks for supporting refinement of requirements and security design decisions. It can be applied during any kind of refinement driven engineering of systems and provides a variety of additional benefits (compare Section 8.4.2).

The example shows how this methodology can be applied and demonstrates that by following this approach even complex scenarios can be handled. Moreover, it also shows that the process is capable of representing non-trivial combinations of requirements, technical solutions and underlying assumptions as they occur in the context of Trusted Computing Base minimization. An earlier version of this approach has already been successfully applied to a real-life case study for Car-2-Car an in-vehicle security architecture design during project EVITA [82].

# 9 Conclusion

This thesis shows that it is possible to reduce the TCB for non-trivial security policies down to a TPM or “mostly a TPM”. With the reduction of the TCB and its physical separation the complexity of the TCB is decreased and thereby the security of the system is increased. Furthermore, given that the TPM was purposefully designed for security tasks and is typically evaluated for Common Criteria, the TCB is adequately strengthened.

In order to evaluate the strength of the TCB, especially for the “mostly a TPM” solutions the notion of how to describe a TCB was extended by a second dimension that represents the dynamic aspects of the TCB boundary over the lifecycle of a system. With Figure 2.4 this difference is already made obvious, since it allows to differentiate between the firmware being part of the TCB during a given authentication process, whilst prior firmwares are not part of the TCB during a present authentication process. This differentiation is also of great importance in Rollback protection mechanisms in Figure 3.3 or in the difference between a trivial PnC credential provisioning with HSMs versus the TrustEV and HIP approachs as depicted in Figure 5.1b versus Figure 5.12.

The implementation of security policies with a TPM as TCB was demonstrated by the implementation of systems with the policies for *Selective Confidentiality of Data and Code based on Device Attributes*, *Protection of Device Resident Data bound to Firmware enabling Updates*, *Secure Credential usage in Plug and Charge*, *Offline Integrity Audit Logs* and *Role-based access management*. These policies are examples of non-trivial security policies that are typically implemented on rich application processors. In this thesis, it was highlighted that a TCB can be implemented based on a TPM as sole or majority of a TCB, even though a TPM contains only a finite set of predefined functionalities. By utilizing the TPM’s usage attributes and Enhanced Authorization policy framework, its NV index storage type (such as counters), import capabilities for confidential key transfer and credential activation means, it is possible to implement a wide variety of security policies. The limited set of functionalities of the TPM are its strength since they provide the basis for highly secure implementations and security evaluation of the TPM chips. It can be seen that it is possible to construct a TCB out of this predefined set of functionalities and that a turing-complete computational basis is oftentimes not required.

Finally, a framework for engineering systems based on security policy refinement and partitioning of properties into several smaller sets has been introduced. It offers a first step towards a structure formal reasoning process that enables the systematic creation of TCBs and the division of a system into the TCB and non-TCB domains.

## 9.1 Future Work

Future works include the implementation of further security policies based on TPMs and to make the process of using TPMs as TCBs more stream-line. One step is to demonstrate how different TPM functionalities serve as building blocks for these TCBs.

In this effort the notion of TCBs extended to the dynamic aspects of the device lifecycle can be of great benefit. In the future the different importance of components and lifecycle aspects should be refined by weighting their contributions to the graphs in order to better represent actual security strengths. Thus this approach could provide a suitable assessment strategy.

By extending the formal based engineering process for deducing security policies and TCBs from global security goals a more systematic approach could be achieved. As a first step the support by graphical tools could benefit a wider adoption and provide a more convenient reasoning framework.

# Bibliography

[1] Martin Abadi and Veronique Cortier. "Deciding knowledge in security protocols under (many more) equational theories". In: *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. IEEE. 2005, pp. 62–76. DOI: 10.1109/CSFW.2005.14.

[2] Carlisle Adams et al. *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)*. RFC 3161 (Proposed Standard). Internet Engineering Task Force, 2001. URL: http://www.ietf.org/rfc/rfc3161.txt.

[3] Open Charge Alliance. *Open Charge Point Protocol 1.6*. Version 1.6. 2016. URL: http://www.openchargealliance.org/protocols/ocpp/ocpp-16/.

[4] Open Charge Alliance. *Open Charge Point Protocol 2.0 - Part 0 - Introduction*. Arnhem, Netherlands, Apr. 2018. URL: https://www.openchargealliance.org/protocols/ocpp-201/.

[5] Open Charge Alliance. *Open Charge Point Protocol 2.0 - Part 2 - Specification*. Arnhem, Netherlands, Apr. 2018. URL: https://www.openchargealliance.org/protocols/ocpp-201/.

[6] Sven Apel et al. "Software Product Lines". In: *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 3–15. DOI: 10.1007/978-3-642-37521-7_1.

[7] James Arbib and Tony Seba. *Rethinking Transportation 2020-2030*. RethinkX, 2017, p. 162. ISBN: 978-0-9994-0160-6.

[8] ARM Limited. *Introducing Arm TrustZone*. Mar. 2019. URL: https://developer.arm.com/technologies/trustzone.

[9] Colin Atkinson, Joachim Bayer, and Dirk Muthig. "Component-Based Product Line Development: The KobrA Approach". In: *Software Product Lines: Experience and Research Directions*. Ed. by Patrick Donohoe. Boston, MA: Springer US, 2000, pp. 289–309. DOI: 10.1007/978-1-4615-4339-8_16.

[10] AUTOSAR. *Specification of Secure Onboard Communication*. 2017. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf.

[11] Kaibin Bao et al. "A threat analysis of the vehicle-to-grid charging protocol ISO 15118". In: *Computer Science - Research and Development* 33.1 (Feb. 2018), pp. 3–12. DOI: 10.1007/s00450-017-0342-y.

[12] Elaine Barker. *NIST Special Publication 800-57 Part 1, Recommendation for Key Management: General*. NIST. 2020. URL: https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final.

[13] H. Birkholz, J. Lu, and N. Cam-Winget. *Security Automation and Continuous Monitoring (SACM) Terminology*. Internet-Draft draft-ietf-sacm-terminology-10. The Internet Engineering Task Force (IETF), July 2016. URL: https://datatracker.ietf.org/doc/draft-ietf-sacm-terminology/.

[14] Henk Birkholz, Nancy Cam-Winget, and Carsten Bormann. "IoT Software Updates need Security Automation". In: *Internet of Things Software Update Workshop*. Dublin, 2016. URL: https://down.dsg.cs.tcd.ie/iotsu/subs/IoTSU_2016_paper_23.txt.

[15] Eric Bodden et al. "Spl lift: statically analyzing software product lines in minutes instead of years". In: *ACM SIGPLAN Notices*. Vol. 48. 6. ACM. 2013, pp. 355–364. DOI: 10.1145/2499370.2491976.

[16] Neil Brown. *Overlay Filesystem*. URL: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[17] Milan Broz. *dm-crypt: Linux kernel device-mapper crypto target*. Version ee336a2c. 2020. URL: https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt.

[18] Milan Broz and Vashek Matyas. "The TrueCrypt On-Disk Format–An Independent View". In: *IEEE security & privacy* 12.3 (2014), pp. 74–77. DOI: 10.1109/MSP.2014.60.

[19] Sarah Buhr and Megan Rose Dickey. "Self-driving Ubers are now picking people up in Arizona". In: *TechCrunch* (2017). URL: https://techcrunch.com/2017/02/21/self-driving-ubers-are-now-picking-people-up-in-arizona/.

[20] Lucas Buschlinger, Markus Springer, and Maria Zhdanova. "Plug-and-Patch: Secure Value Added Services for Electric Vehicle Charging". In: *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*. Vol. 2. 2019, pp. 1–10. DOI: 10.1145/3339252.3339269.

[21] John Calcote. *Autotools: a practitioner's guide to GNU autoconf, automake, and libtool*. No Starch Press, 2019. ISBN: 978-1-5932-7972-1.

[22] CAR 2 CAR Communication Consortium. *Protection Profile V2X Hardware Security Module, Release 1.3.0*. 2018. URL: https://www.car-2-car.org/fileadmin/documents/Basic_System_Profile/Release_1.3.0/C2CCC_PP_2056_HSM.pdf.

[23] Carnegie Mellon Software Engineering Institute Web Site. *Software Product Lines*. 2016. URL: http://www.sei.cmu.edu/productlines/.

[24] Marco Carvalho et al. "Heartbleed 101". In: *IEEE security & privacy* 12.4 (2014), pp. 63–67. DOI: 10.1109/MSP.2014.66.

[25] Stephen Checkoway et al. "Comprehensive Experimental Analyses of Automotive Attack Surfaces". In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011, p. 6. DOI: 10.5555/2028067.2028073.

[26] Stephen Checkoway et al. "Comprehensive Experimental Analyses of Automotive Attack Surfaces". In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011, p. 6. DOI: 10.5555/2028067.2028073.

[27] Paul Clements and Linda Northrop. *Software Product Lines : Practices and Patterns*. 6th ed. Addison Wesley, 2007. ISBN: 978-0201703320.

[28] Jonathan Corbet. "dm-verity". In: *Linux Weekly News* (Sept. 2011). URL: https://lwn.net/Articles/459420/.

[29] Jonathan Corbet. "The Integrity Measurement Architecture". In: *Linux Weekly News* (2005). URL: https://lwn.net/Articles/137306/.

[30] Anupam Datta et al. "On Adversary Models and Compositional Security". In: *IEEE Security & Privacy* 9.3 (2011), pp. 26–32. DOI: 10.1109/MSP.2010.203.

[31] Soumya Kanti Datta. "W3C Web of Things Interest Group Face-to-Face Meeting at EURECOM [Future Directions]". In: *IEEE Consumer Electronics Magazine* 5.3 (2016), pp. 40–43. DOI: 10.1109/MCE.2016.2556820.

[32] David Challener and Will Arthur and Kenneth Goldman. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Springer, 2015. ISBN: 978-1-4302-6583-2.

[33] Alex Davies. “GM’s Robocar Service Drives Employees Around SF for Free”. In: *Wired* (Sept. 2017). URL: `https://www.wired.com/story/gm-cruise-anywhere-self-driving-san-francisco/`.

[34] Dorothy E. Denning. “A Lattice Model of Secure Information Flow”. In: *Communications of the ACM* 19.5 (May 1976), pp. 236–243. DOI: `10.1145/360051.360056`.

[35] Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*. 1986.

[36] Google Developers. *Android keystore system*. 2019. URL: `https://developer.android.com/training/articles/keystore.html`.

[37] Alexandra Dmitrienko and Christian Plappert. “Secure Free-Floating Car Sharing for Offline Cars”. In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. CODASPY ’17. Scottsdale, Arizona, USA: ACM, 2017. DOI: `10.1145/3029806.3029807`.

[38] D. Dolev and A. Yao. “On the security of public key protocols”. In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: `10.1109/TIT.1983.1056650`.

[39] Jörn Eichler, Andreas Fuchs, and Nico Lincke. “Supporting Security Engineering at Design Time with Adequate Tooling”. In: *2012 IEEE 15th International Conference on Computational Science and Engineering*. 2012, pp. 194–201. DOI: `10.1109/ICCSE.2012.34`.

[40] Elizabeth Flanagan. “The Yocto Project”. In: *The Architecture of Open Source Applications* 2 (2012), pp. 347–358.

[41] Thomas Fountian. *Secure feature activation*. US Patent App. 10/526,252. Aug. 2003.

[42] Clemens Fruhwirth. *LUKS On-Disk Format Specification Version 1.2*. 2018. URL: `http://cdn.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/on-disk-format.pdf`.

[43] Andreas Fuchs. “Improving scalability of remote attestation”. Diplomarbeit. Technische Universität Darmstadt, 2008.

[44] Andreas Fuchs and Sigrid Gürgens. “Preserving Confidentiality in Component Compositions”. In: *Software Composition - 12th International Conference, SC 2013, Budapest, Hungary, June 19, 2013. Proceedings*. Springer. 2013, pp. 33–48. DOI: 10.1007/978-3-642-39614-4_3.

[45] Andreas Fuchs, Sigrid Gürgens, and Carsten Rudolph. “A Formal Notion of Trust – Enabling Reasoning about Security Properties”. In: *Trust Management IV*. Ed. by Masakatsu Nishigaki et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 200–215. DOI: 10.1007/978-3-642-13446-3_14.

[46] Andreas Fuchs, Sigrid Gürgens, and Carsten Rudolph. “Formal Notions of Trust and Confidentiality - Enabling Reasoning about System Security”. In: *Journal of Information Processing* 19 (2011), pp. 274–291. DOI: 10.2197/ipsjjip.19.274.

[47] Andreas Fuchs, Sigrid Gürgens, and Carsten Rudolph. “On the Security Validation of Integrated Security Solutions”. In: *Emerging Challenges for Security, Privacy and Trust*. Springer, 2009, pp. 190–201. DOI: 10.1007/978-3-642-01244-0_17.

[48] Andreas Fuchs, Christoph Krauß, and Jürgen Repp. “Advanced Remote Firmware Upgrades Using TPM 2.0”. In: *ICT Systems Security and Privacy Protection*. Ed. by Jaap-Henk Hoepman and Stefan Katzenbeisser. Springer. 2016, pp. 276–289. DOI: 10.1007/978-3-319-33630-5_19.

[49] Andreas Fuchs, Christoph Krauß, and Jürgen Repp. “Runtime Firmware Product Lines Using TPM 2.0”. In: *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2017, pp. 248–261. DOI: 10.1007/978-3-319-58469-0_17.

[50] Andreas Fuchs and Roland Rieke. “Identification of authenticity requirements in systems of systems by functional security analysis”. In: *Workshop on Architecting Dependable Systems (WADS 2009), in Proceedings of the 2009 IEEE/IFIP Conference on Dependable Systems and Networks, Supplementary Volume*. 2009.

[51] Andreas Fuchs and Roland Rieke. “Identification of Security Requirements in Systems of Systems by Functional Security Analysis”. In: *Architecting Dependable Systems VII*. Ed. by Antonio Casimiro, Rogério de Lemos, and Cristina Gacek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 74–96. DOI: 10.1007/978-3-642-17245-8_4.

[52] Andreas Fuchs and Carsten Rudolph. “Security Engineering Based on Structured Formal Reasoning”. In: *2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom)*. IEEE. 2012, pp. 145–152. DOI: 10.1109/BioMedCom.2012.30.

[53] Andreas Fuchs et al. “HIP-20: Integration of Vehicle-HSM-Generated Credentials into Plug-and-Charge Infrastructure”. In: *Computer Science in Cars Symposium*. CSCS ’20. Feldkirchen, Germany: Association for Computing Machinery, 2020. DOI: 10.1145/3385958.3430483.

[54] Andreas Fuchs et al. “HIP: HSM-based Identities for Plug-and-Charge”. In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. ARES ’20. Virtual Event, Ireland: Association for Computing Machinery, 2020, pp. 1–6. DOI: 10.1145/3407023.3407066.

[55] Andreas Fuchs et al. *Time-Based Uni-Directional Attestation*. Internet-Draft draft-birkholz-tuda-02. The Internet Engineering Task Force (IETF), July 2016. URL: https://datatracker.ietf.org/doc/draft-birkholz-tuda/.

[56] Andreas Fuchs et al. “TrustEV: Trustworthy Electric Vehicle Charging and Billing”. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC ’20. Brno, Czech Republic: Association for Computing Machinery, 2020, pp. 1706–1715. DOI: 10.1145/3341105.3373879.

[57] Dipayan P. Ghosh, Robert J. Thomas, and Stephen B. Wicker. “A Privacy-Aware Design for the Vehicle-to-Grid Framework”. In: *2013 46th Hawaii International Conference on System Sciences*. IEEE. 2013, pp. 2283–2291. DOI: 10.1109/HICSS.2013.54.

[58] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. 2004. ISBN: 978-0-2017-7595-2.

[59] Andy Greenberg. “Hackers Remotely Kill a Jeep on the Highway – With Me in It”. In: *Wired* (July 2015). URL: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[60] Trusted Computing Group. *Algorithm Registry*. Revision 01.32. 2015. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG-_Algorithm_Registry_r1p32_pub.pdf.

[61] Trusted Computing Group. *TPM 1.2 Main Specification Part 1: Design Principles*. 2011. URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf.

[62] Trusted Computing Group. *TPM 1.2 Main Specification Part 2: Structures*. 2011. URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-2-TPM-Structures_v1.2_rev116_01032011.pdf.

[63] Trusted Computing Group. *TPM 1.2 Main Specification Part 3: Commands*. 2011. URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-3-Commands_v1.2_rev116_01032011.pdf.

[64] Sigrid Gürgens, Peter Ochsenschläger, and Carsten Rudolph. "Abstractions Preserving Parameter Confidentiality". In: *Computer Security – ESORICS 2005*. Ed. by Sabrina de Capitani di Vimercati, Paul Syverson, and Dieter Gollmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 418–437. DOI: 10.1007/11555827_24.

[65] Sigrid Gürgens, Peter Ochsenschläger, and Carsten Rudolph. "Authenticity and Provability - A Formal Framework". In: *Proceedings of the International Conference on Infrastructure Security*. InfraSec '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 227–245. ISBN: 3540443096.

[66] Sigrid Gürgens, Peter Ochsenschläger, and Carsten Rudolph. "On a Formal Framework for Security Properties". In: *Comput. Stand. Interfaces* 27.5 (2005), pp. 457–466. DOI: 10.1016/j.csi.2005.01.004.

[67] Sigrid Gürgens, Peter Ochsenschläger, and Carsten Rudolph. "Parameter confidentiality". In: *Informatik 2003 – Teiltagung Sicherheit*. Ed. by Rüdiger Grimm, Hubert B. Keller, and Kai Rannenberg. Gesellschaft für Informatik, 2003. URL: http://dl.gi.de/handle/20.500.12116/29597.

[68] Sigrid Gürgens et al. "Security Evaluation of Scenarios Based on the TCG's TPM Specification". In: *Computer Security – ESORICS 2007*. Ed. by Joachim Biskup and Javier López. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 438–453. DOI: 10.1007/978-3-540-74835-9_29.

[69] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. "On the notion of variability in software product lines". In: *Proceedings Working IEEE/IFIP Conference on Software Architecture*. IEEE. 2001, pp. 45–54. DOI: 10.1109/WICSA.2001.948406.

[70] Charles B. Haley et al. "Arguing security: Validating security requirements using structured argumentation". In: *Third Symposium on Requirements Engineering for Information Security (SREIS'05) held in conjunction with the 13th International Requirements Engineering Conference (RE'05)*. 2005.

[71] Brahim Hamid, Sigrid Gürgens, and Andreas Fuchs. “Security patterns modeling and formalization for pattern-based development of secure software systems”. In: *Innovations in Systems and Software Engineering* (2015). DOI: 10.1007/s11334-015-0259-1.

[72] Brahim Hamid et al. “Enforcing S&D Pattern Design in RCES with Modeling and Formal Approaches”. In: *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ed. by Jon Whittle. Vol. 6981. Springer, 2011, pp. 319–333. DOI: 10.1007/978-3-642-24485-8_23.

[73] Ai-qin HAN and Jian-lu LIU. “Discussion on Open Attestation”. In: *Journal of Tianjin Administrative Cadre College of Politics and Law* 3 (2006), p. 13.

[74] David Harrington and David Waltermire. *Endpoint Security Posture Assessment: Enterprise Use Case*. RFC 7632. The Internet Engineering Task Force (IETF), 2015. URL: https://tools.ietf.org/html/rfc7632.

[75] Michael Henson and Stephen Taylor. “Memory Encryption: A Survey of Existing Techniques”. In: *ACM Comput. Surv.* 46.4 (2014), pp. 1–26. DOI: 10.1145/2566673.

[76] IBM. *Secure Engineering Framework*. URL: http://www.redbooks.ibm.com/abstracts/redp4641.html.

[77] *IEEE 802.1AR - Secure Device Identity*. IEEE. 2009. URL: http://www.ieee802.org/1/pages/802.1ar.html.

[78] Atsushi Ishii and Narasimhan Parthasarathy. *SIM-based automatic feature activation for mobile phones*. US Patent App. 10/830,755. Apr. 2004.

[79] ISO/IEC. *Road vehicles – Vehicle to grid communication interface – Part 1: General information and use-case definition*. ISO Standard ISO 15118-1:2019. Geneva, Switzerland: International Organization for Standardization, Apr. 2019.

[80] ISO/IEC. *Road vehicles – Vehicle-to-Grid Communication Interface – Part 2: Network and application protocol requirements*. ISO Standard 15118-2:2018. Geneva, Switzerland: International Organization for Standardization, Dec. 2018.

[81] Daniel Jiang and Luca Delgrossi. “IEEE 802.11p: Towards an International Standard for Wireless Access in Vehicular Environments”. In: *VTC Spring 2008 - IEEE Vehicular Technology Conference*. IEEE. 2008, pp. 2036–2040. DOI: 10.1109/VETECS.2008.458.

[82] Christoph Jouvray et al. *Deliverable D3.1: Security and Trust Model v2*. Tech. rep. Project Evita, 2011.

[83] Karl Koscher et al. “Experimental Security Analysis of a Modern Automobile”. In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 447–462. DOI: 10.1109/SP.2010.34.

[84] Nitin Kumar and Vipin Kumar. *Bitlocker and Windows Vista*. 2008.

[85] Michael Lauer. “Building embedded linux distributions with bitbake and openembedded”. In: *Free and Open Source Software Developers’ European Meeting*. 2005.

[86] Torsten Lodderstedt, David Basin, and Jürgen Doser. “SecureUML: A UML-Based Modeling Language for Model-Driven Security”. In: *UML 2002 – The Unified Modeling Language*. Ed. by Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 426–441. DOI: 10.1007/3-540-45800-X_33.

[87] Federico Maggi. *The Crisis of Connected Cars: When Vulnerabilities Affect the CAN Standard*. Aug. 2017. URL: https://blog.trendmicro.com/trendlabs-security-intelligence/connected-car-hack/.

[88] Microsoft. *Overlay for Unified Write Filter (UWF)*. URL: https://msdn.microsoft.com/en-us/library/windows/hardware/mt571992(v=vs.85).aspx.

[89] Microsoft. *Security Development Lifecycle*. URL: http://www.microsoft.com/security/sdl/default.aspx.

[90] Charlie Miller and Chris Valasek. *A Survey of Remote Automotive Attack Surfaces*. 2014. URL: https://ioactive.com/a-survey-of-remote-automotive-attack-surfaces/.

[91] Charlie Miller and Chris Valasek. *Adventures in Automotive Networks and Control Units*. 2014. URL: https://ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf.

[92] Chris Mitchell. *Trusted Computing*. Institution of Engineering and Technology, 2005. ISBN: 978-0-86341-525-8.

[93] Jeff Nickoloff. *Docker in action*. Manning, 2016. ISBN: 978-1-6334-3023-5.

[94] *OCHPdirect extension to the Open Clearing House Protocol*. Version 0.2. smartlab. Aug. 2016. URL: http://www.ochp.eu.

[95] *Open Charge Point Interface*. Version 2.2. NKL. Sept. 2019. URL: https://ocpi-protocol.org.

[96] *Open Clearing House Protocol*. Version 1.4. smartlab. Aug. 2016. URL: http://www.ochp.eu.

[97] *Open InterCharge Protocol for Emobility Service Provider / Charge Point Operator*. Version 2.2. Hubject GmbH. Oct. 2017. URL: https://www.hubject.com.

[98] Charlie Osborne. *How to steal a Tesla Model S in seconds*. Sept. 2018. URL: https://www.zdnet.com/article/how-to-steal-a-tesla-model-s-in-seconds/.

[99] Charlie Osborne. *OwnStar: Unlock and track any GM OnStar connected car for $100*. July 2015. URL: https://www.zdnet.com/article/how-to-steal-a-tesla-model-s-in-seconds/.

[100] Jonathan Petit et al. "Remote attacks on automated vehicles sensors: Experiments on camera and lidar". In: *Black Hat Europe* 11 (2015). URL: https://api.semanticscholar.org/CorpusID:39608826.

[101] Christian Plappert, Lukas Jäger, and Andreas Fuchs. "Secure Role and Rights Management for Automotive Access and Feature Activation". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 227–241. ISBN: 9781450382878. DOI: 10.1145/3433210.3437521.

[102] "PlayStation Network breach has cost Sony $171 million". In: *cbsnews* (). URL: http://www.cbsnews.com/8301-504083_162-20065621-504083.html.

[103] David Quigley et al. "Unionfs: User-and community-oriented development of a unification filesystem". In: *Proceedings of the 2006 Linux Symposium*. Vol. 2. 2006, pp. 349–362.

[104] P. Rademakers and P. Klapwijk. *EV Related Protocol Study*. Jan. 2017. URL: https://www.elaad.nl/uploads/downloads/downloads_download/EV_related_protocol_study_v1.1.pdf.

[105] Marco Rocchetto and Nils Ole Tippenhauer. "CPDY: extending the Dolev-Yao attacker with physical-layer interactions". In: *International Conference on Formal Engineering Methods*. Springer. 2016. DOI: 10.1007/978-3-319-47846-3_12.

[106] Ishtiaq Rouf et al. "Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study". In: *19th USENIX Security Symposium (USENIX Security 10)*. Washington, DC: USENIX Association, 2010. URL: https://www.usenix.org/conference/usenixsecurity10/security-and-privacy-vulnerabilities-car-wireless-networks-tire-pressure.

[107] Carsten Rudolph and Andreas Fuchs. "Redefining Security Engineering". In: *2012 5th International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE. 2012, pp. 1–6. DOI: 10.1109/NTMS.2012.6208773.

[108] John Rushby. "A trusted computing base for embedded systems". In: *Proceedings 7th DoD/NBS Computer Security Conference*. Citeseer. 1984, pp. 294–311.

[109] Neetesh Saxena et al. "Network Security and Privacy Challenges in Smart Vehicle-to-Grid". In: *IEEE Wireless Communications* 24.4 (2017), pp. 88–98. DOI: 10.1109/MWC.2016.1600039WC.

[110] Sunil Saxena and HV Kwon. "Tizen architecture". In: *Tizen Developer Conference, San Francisco, California*. 2012.

[111] Kai Schramm and Marko Wolf. "Secure Feature Activation". In: *SAE International Journal of Passenger Cars-Electronic and Electrical Systems* 2 (2009), pp. 62–67. DOI: 10.4271/2009-01-0262.

[112] Steffen Schulz, Ahmad-Reza Sadeghi, and Christian Wachsmann. "Short Paper: Lightweight Remote Attestation Using Physical Functions". In: *Proceedings of the Fourth ACM Conference on Wireless Network Security*. WiSec '11. Hamburg, Germany: Association for Computing Machinery, 2011, pp. 109–114. DOI: 10.1145/1998412.1998432.

[113] Hendrik Schweppe et al. *Deliverable D3.2: Secure on-board protocols specification*. Tech. rep. 2011.

[114] *Security Automation and Continuous Monitoring Charter*. Charter charter-ietf-sacm-02. The Internet Engineering Task Force (IETF), Aug. 2016. URL: https://datatracker.ietf.org/doc/charter-ietf-sacm/.

[115] HIS – HerstellerInitiative Software. *SHE – Secure Hardware Extension Functional Specification*. 2009.

[116] Frederic Stumpf et al. "Improving the Scalability of Platform Attestation". In: *Proceedings of the Third ACM Workshop on Scalable Trusted Computing (ACM STC 2008)*. Alexandria, VA, USA: ACM Press, 2008, pp. 1–10. DOI: 10.1145/1456455.1456457.

[117] Vangelis Stykas and George Lavdanis. *Incorrect privilege assignment could lead to full user and vehicle compromise*. May 2018. URL: https://seclists.org/fulldisclosure/2018/May/37.

[118] Joseph Tardo et al. *Network Endpoint Assessment (NEA): Overview and Requirements*. RFC. 2008. URL: https://datatracker.ietf.org/doc/rfc5209/.

[119] *TCG Attestation PTS Protocol: Binding to TNC IF-M*. Version 1.0, Revision 28. Trusted Computing Group. Aug. 2011.

[120] *TCG GLossary*. Version 1.1, Revision 1.00. Trusted Computing Group. May 2017.

[121] *TCG TPM 2.0 Library Profile for Automotive-Thin*. Version 1.0. Trusted Computing Group. Mar. 2015.

[122] *TCG TSS 2.0 TPM Command Transmission Interface (TCTI) API Specification*. Version 1.0 Revision 12. Trusted Computing Group. Apr. 2019.

[123] technewsdaily. *Why We Won't Soon See Another Stuxnet Attack*. URL: `http://www.technewsdaily.com/7012-stuxnet-anniversary-look-ahead.html`.

[124] *Thing-to-Thing Research Group Charter*. Charter charter-irtf-t2trg-01. The Internet Research Task Force (IRTF), 2015. URL: `https://datatracker.ietf.org/doc/charter-irtf-t2trg/`.

[125] Trusted Computing Group. *TCG TPM v2.0 Provisioning Guidance*. Guidance Version 1.0 - Revision 1.0. Mar. 2017.

[126] Trusted Computing Group. *Trusted Computing Group TPM 2.0 Library Specification Approved as an ISO/IEC International Standard*. June 2015.

[127] Trusted Computing Group. *Trusted Platform Module Library - Part 1: Architecture*. Specification Family 2.0 - Rev. 01.38. Sept. 2016.

[128] *Trusted Platform Module Library Specification*. Family 2.0, Level 00, Revision 01.16. Trusted Computing Group. Oct. 2014.

[129] *TSS 2.0 JSON Data Types and Policy Language Specification*. Version 0.7, Revision 08. Trusted Computing Group. June 2020.

[130] *TSS Feature API Specification*. Family 2.0, Level 00,Revision 00.12. Trusted Computing Group. Nov. 2014.

[131] *TSS System Level API and TPM Command Transmission Interface Specification*. Family 2.0, Revision 01.00. Trusted Computing Group. Jan. 2015.

[132] *TSS2.0 Enhanced SystemAPI (ESAPI) Specification*. Version 1.00, Revision 08. Trusted Computing Group. May 2020.

[133] *TSS2.0 Overview and Common Structures Specification*. Version 0.90, Revision 03. Trusted Computing Group. Oct. 2019.

[134] Liam Tung. *VW-Audi security: Multiple infotainment flaws could give attackers remote access*. May 2018. URL: https://www.zdnet.com/article/vw-audi-security-multiple-infotainment-flaws-could-give-attackers-remote-access/.

[135] James Vincent. "World's first self-driving taxi trial begins in Singapore". In: *The Verge* (Aug. 2016). URL: https://www.theverge.com/2016/8/25/12637822/self-driving-taxi-first-public-trial-singapore-nutonomy.

[136] Bernhard Walzel et al. "Robot-Based Fast Charging of Electric Vehicles". In: *WCX SAE World Congress Experience*. Apr. 2019. DOI: 10.4271/2019-01-0869.

[137] *Web of Things Interest Group Charter*. Charter. The World Wide Web Consortium (W3C), Aug. 2016. URL: https://www.w3.org/2016/07/wot-ig-charter.html.

[138] Benjamin Weyl et al. "Secure on-board architecture specification". In: *Evita Deliverable D3.2* 3 (2010), p. 2.

[139] Wikihow. *How to Downgrade a PSP*. URL: http://www.wikihow.com/Downgrade-a-PSP.

[140] Matthew Wojcik et al. "Introduction to OVAL". In: *The MITRE Corporation* (2005).

[141] Cheng Xu et al. "A Remote Attestation Security Model Based on Privacy-Preserving Blockchain for V2X". In: *IEEE Access* 6 (2018), pp. 67809–67818. DOI: 10.1109/ACCESS.2018.2878995.

[142] Daniel Zelle et al. "Anonymous Charging and Billing of Electric Vehicles". In: *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ARES 2018. Hamburg, Germany: Association for Computing Machinery, 2018, pp. 1–10. DOI: 10.1145/3230833.3230850.

[143] Daniel Zelle et al. "On Using TLS to Secure In-Vehicle Networks". In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ARES '17. Reggio Calabria, Italy: Association for Computing Machinery, 2017. DOI: 10.1145/3098954.3105824.

[144] Kim Zetter. "Sony Got Hacked Hard: What We Know and Don't Know So Far". In: *wired* (2014). URL: https://www.wired.com/2014/12/sony-hack-what-we-know/.

[145] Tianyu Zhao et al. "A secure and privacy-preserving payment system for Electric vehicles". In: *2015 IEEE International Conference on Communications (ICC)*. IEEE. 2015, pp. 7280–7285. DOI: 10.1109/ICC.2015.7249489.