Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Lari Lehtomäki

Realizing eID scheme on TPM 2.0 hardware

Master's Thesis
Espoo, April 12th, 2016

Supervisor:	Professor N. Asokan
Instructor:	M.Sc. Thomas Nyman

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author: Lari Lehtomäki

Title:
Realizing eID scheme on TPM 2.0 hardware

Date:	April 12th, 2016	**Pages:**	52
Major:	Data Communication Software	**Code:**	T-110
Supervisor:	Professor N. Asokan		
Instructor:	M.Sc. Thomas Nyman		

Most of the financial, healthcare, and governmental services are available on Internet, where traditional identification methods used on face-to-face identification are not possible. Identification with username and password is a mediocre solution and therefore some services require strong authentication. Finland has three approved strong authentication methods: smart cards, bank credentials, and mobile ID. Out of the three authentication methods, only the government issued smart card is available to everyone who police can identify reliably. Bank credentials require identification with an identity document from Finland or other European Economic Area (EEA) country. Mobile ID explicitly require identification with Finnish identity document. The problem with smart cards is the requirement for a reader, slow functioning, and requirement for custom driver. A TPM could function as a replacement for a smart card with accompanying software library.

In this thesis, I created a PKCS #11 software library that allows TPM to be used for browser based authentication according to draft specification by Finnish population registry. The keys used for authentication are created, stored and used securely inside the TPM. TPMs are deemed viable replacement for smart cards. The implemented system is faster to use than smart cards and has similar security properties as smart cards have. The created library contains implementations for 30% of all TPM 2.0 functions and could be used as a base for further TPM 2.0 based software.

Keywords:	security, trusted computing, eID, PKCS#11
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Lari Lehtomäki		
Työn nimi: Sähköisen identiteetin toteuttaminen TPM 2.0 -laitteistolla			
Päiväys:	12. huhtikuu 2016	**Sivumäärä:**	52
Pääaine:	Tietoliikenneohjelmistot	**Koodi:**	T-110
Valvoja:	Professori N. Asokan		
Ohjaaja:	FM Thomas Nyman		

Pankki-, terveys- ja julkiset palvelut ovat suureksi osin saatavilla internetin välityksellä. Tunnistautuminen käyttäjätunnuksella ja salasanalla ei takaa riittävää luotettavuutta, vaan joissain palveluissa on käytettävä vahvaa tunnistautumista. Suomessa on tällä hetkellä käytössä kolme vahvaa tunnistautumisvälinettä: pankkien käyttämät verkkopankkitunnukset, Väestörekisterikeskuksen kansalaisvarmenne ja teleyritysten mobiilivarmenteet. Näistä kolmesta kansalaisvarmenne on ainoa, joka ei vaadi asiakkuutta ja on täten kaikille saatavilla, jotka poliisi voi luotettavasti tunnistaa. Verkkopankkitunnukset vaativat tunnistautumisen suomalaisella tai Euroopan talousalueen (ETA) valtion myöntämällä henkilötodistus. Mobiilivarmenne myönnetään vain henkilölle, joka voidaan tunnistaa suomalaisella henkilötodistuksella. Kansalaisvarmenne on kuitenkin älykortti kaikkine älykortin ongelmineen: sen käyttämiseen tarvitaan erillinen lukija, sen toiminta on hidasta ja se vaatii erillisen laiteajurin. Tämän työn tavoitteena on luoda ratkaisu, jolla älykorttipohjainen tunnistautuminen voidaan toteuttaa tietokoneissa olevan TPM-piirin avulla.

Tässä diplomityössä luotiin PKCS #11 -rajapinnan täyttävä ohjelmistokirjasto, joka mahdollistaa TPM-piirin käyttämisen tunnistautumiseen selaimessa Väestörekisterikeskuksen laatiman määritelmän luonnoksen mukaan. Tunnistautumisavaimet luodaan, tallennetaan ja niitä käytetään TPM:ssa, mikä varmistaa avainten luottamuksellisuuden. Älykortin toiminnallisuudet todettiin mahdolliseksi toteuttaa TPM-piirillä. Toteutettu järjestelmä on nopeampi käyttää kuin älykortti ja se takaa älykortteja vastaavan tietoturvatason. Työn tuloksena tehty kirjasto toteuttaa 30 % kaikista TPM 2.0 -ohjelmistorajapinnoista, ja kirjastoa voidaan käyttää osana tulevia TPM 2.0 -ohjelmistoja.

Asiasanat:	tietoturva, vahva tunnistautuminen, sähköinen henkilökortti
Kieli:	Englanti

Acknowledgements

Foremost, I would like to express my gratitude to Professor Asokan for offering an interesting topic and a position on the Secure Systems group at Aalto University. I am grateful to M.Sc. Thomas Nyman, my instructor, for guidance on the topic, help on academic writing and patience reading draft versions. I also wish to thank D.Sc. Jan-Erik Ekberg for providing valuable help on understanding the TPM.

Last but not least, I would like to thank my family for supporting me on this journey. Special thanks for Riku who helped multiple times with structure and secrets of scientific writing. Finally, thank you Elli for the motivation and support you have provided.

[illegible]

Espoo, April 12th, 2016

Lari Lehtomäki

Abbreviations and Acronyms

API	Application Programming Interface
CSR	Certificate Signing Request
EA	Enhanced Authorization
eID	Electronic Identification
EK	Endorsement Key
GP	GlobalPlatform
HMAC	Hash-based Message Authentication Code
HSM	Hardware Security Module
ME	Management Engine
MMU	Memory Management Unit
MPU	Memory Protection Unit
OS	Operating System
PCR	Platform Configuration Register
PKCS	Public-Key Cryptography Standards
REE	Rich Execution Environment
RTM	Root of Trust for Measurement
SE	Secure Element

SMM	System Management Mode
SoC	System on a Chip
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCTI	TPM command transmission interface
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TSS	TPM Software Stack
TPM	Trusted Platform Module

Contents

1 Introduction 9
1.1 Problem statement . . . 10
1.2 Research Goals . . . 10
1.3 Structure . . . 10

2 Background 12
2.1 Public-key cryptography . . . 13
2.2 Standardization . . . 13
2.3 Trusted Platform Module . . . 14
2.3.1 Platform Configuration Register . . . 16
2.3.2 Enhanced Authorization . . . 17
2.3.3 TPM Software Stack . . . 18
2.3.4 TPM Form Factor . . . 19
2.4 Trusted Execution Environment . . . 20
2.5 Intel Management Engine . . . 21
2.6 ARM TrustZone . . . 21
2.7 On-Board Credentials . . . 22
2.8 Smart cards . . . 22

3 Design 23
3.1 Requirements . . . 25
3.1.1 Functionality . . . 25
3.1.2 Algorithms . . . 25
3.2 Mapping TPM and PKCS #11 interfaces . . . 26
3.2.1 Sessions . . . 26
3.2.2 Authentication . . . 26
3.2.3 Multi-part command flow . . . 27

4 Implementation **28**
4.1 Architecture . . . 28
4.2 TPM communication module . . . 29
4.2.1 TPM communication interface . . . 30
4.2.2 Implemented TPM functions . . . 31
4.2.2.1 Functions used for eID . . . 31
4.2.2.2 Additional implemented functions . . . 33
4.3 PKCS #11 module . . . 34
4.3.1 PKCS #11 interface . . . 35

5 Evaluation **36**
5.1 Implementation coverage . . . 36
5.2 Performance metrics . . . 37
5.3 Results . . . 38

6 Discussion **41**
6.1 Security compared to smart card . . . 41
6.2 Usability compared to smart card . . . 42
6.3 Improvements . . . 43

A Introduction to common TPM2 commands **50**

Chapter 1

Introduction

Internet has revolutionized services that previously required face-to-face contact. Brick-and-mortar services are superseded with Internet-based services in the banking industry, governmental services, and local take-away restaurants. Lack of face-to-face contact and traditional methods of identifying people, Internet services need new kinds of authentication methods. Passwords are widely used, but have proven to be inefficient and insecure solution on critical systems. User security is further weakened with typical end user habits of reusing the passwords on multiple sites, not using adequately long passwords, and choosing passwords that are in some way predictable [9]. These systems require strong authentication to protect users from malicious entities.

The Finnish *Act on Strong Electronic Identification and Electronic Signatures* [1] defines three orthogonal methods to authenticate a user. Authentication can be done with something the user knows (e.g. passwords), something the user possesses (e.g. one time codes, tokens), and some physically unique property of the user. The act states that strong user authentication is based on two methods out of the three possible. Generally this is known as multi-factor authentication.

Currently citizens in Finland have three strong authentication methods available: the government issued identity card with smart card (eID), banking credentials with one time codes, and mobile ID [7]. The Finnish citizen card (FINeID) requires a separate hardware reader and driver software, which limit the utility to only committed users [21, 22]. Banking credentials can be issued only to customers of banks, which can be a problem, for example, for foreigners [4, 5]. A mobile ID requires mobile phone to have charge, a working network connection, and is dependent on mobile carriers [6].

1.1 Problem statement

There is a need for a public key based user authentication system that can be made widely available. Using public-key cryptography as a strong authentication method has proven to be a secure solution which has been used on Transport Layer Security (TLS) protected websites as well as on physical smart cards. As previously mentioned, smart card -based systems are not easy to use. To have a viable replacement technology for passwords, public-key cryptography needs a functional and easy-to-use method for key management. Combining this key management method with a solution to integrate public-key authentication to web browsers would enhance the security of web authentication.

1.2 Research Goals

The aim of this thesis is to implement a hardware based solution for public-key authentication, which is widely deployable. Keeping in mind the previously stated problems on the current system, the goal is to implement a replacement method for smart card based eIDs. The solution should be a strong authentication method and provide the same functionality as the current smart cards. Strong authentication is achieved with public-key cryptography and these keys are stored using TPM 2.0 as a key-storage. The solution is available to use on web browsers, and therefore the implemented solution is packaged as a software library. The solution uses TPM 2.0 hardware, which is expected to be widely available on PC platform and has not been used on similar systems previously.

1.3 Structure

This thesis is divided into seven chapters. The work is motivated by describing the need for new authentication methods that provide an assurance required on financial and governmental services. Chapter 2 - Background will introduce hardware security mechanisms used in this thesis and comparable technologies. Chapters 3 - Design and 4 - Implementation explains the logical design and implementation details used to combine TPM 2.0 functionality behind PKCS #11 interface commonly used in authentication. Chapter 5 - Evaluation reports implementation coverage and compares the performance of implemented model to a smart card based eID solution. Chapter 5.3 - Results aggregates the work done and the evaluation of solution. Chapter 6 - Discussion compares TPM based solution to smart card based eID solution on security and functionality.

Chapter 2

Background

Trusted Computing is a general term describing hardware assisted security technologies used to secure software. The need for security technologies arise as the systems have a growing number of components from multiple sources and not all the sources can be verified or trusted equally. Similarly, single large software could be divided to different security levels.

The security of a system is based on a set of hardware and software components which is called the Trusted Computing Base (TCB). A vulnerability in some part of TCB would compromise the security of whole system. On the other hand, misbehaving hardware or vulnerability in software outside of TCB must not affect the security of the whole system.

Hardware based solutions to trusted computing vary greatly as they have different objectives and constraints. A Trusted Platform Module (TPM) is a secure crypto-processor, which is used as key storage, to authenticate the platform, and to ensure platform integrity. A Trusted Execution Environment (TEE) is a complete execution environment running custom software protected from Operating System (OS) and applications in a normal environment. A Secure Element (SE) is a tamper-resistant platform capable of running confidential applications and storing their data securely. Memory Management Unit (MMU) and Memory Protection Unit (MPU) are hardware modules used to segregate memory regions for different tasks. Memory allocation can be used as a building block for security architectures [14]. Virtualization technologies (Intel VT-x etc.) can be used to create isolated environments, which is similar to the concept of TEE [32]. These solutions require at least partial trust on the hardware.

2.1 Public-key cryptography

In symmetric cryptography, both the sender and the receiver use the same shared key to encrypt and decrypt the communication. Public-key cryptography uses two keys, a public key and a private key. Anyone with a public key can encrypt a message, which can be decrypted only with a matching private key. Public-key cryptography can equally be used to create a digital signatures. A private key can be used to create a signature, which can be verified with a matching public key.

2.2 Standardization

Multiple approaches to trusted computing has created also various separate standardization efforts to enable trusted computing. The most prominent ones are carried out by the Trusted Computing Group (TCG) and GlobalPlatform (GP).

The Trusted Computing Group (TCG) is non-profit organization developing trusted computing standards. The most active standard has been the TPM standards, which define a security module for the PC platform. TPM module functions as a Root of Trust for Measurement (RTM) and includes functionality for public-key cryptography and key storage. A new version of TPM standard, the version 2.0, was published in 2014 and it contains new features including new algorithms and an authorization model, which are used in this thesis.

GlobalPlatform (GP) is a non-profit industry consortium combining experts from multiple businesses to collaborate and standardize specifications on issues in trusted computing. Their standardization effort has focused on Secure Elements (SEs) and TEEs. Standards published by GlobalPlatform include a Card Specification and a multi-part TEE device specification. A card specification standardizes smart cards, payment cards, and transportation cards. TEE device specifications create a common Application Programming Interface (API) for multiple hardware based TEEs.

A group of Public-Key Cryptography Standards (PKCS), originally published by RSA Security, define techniques to use on public-key cryptography. Four RSA operations: encrypt, decrypt, sign, and verify are defined on first published standard PKCS #1 [25]. Widely used format for Certificate Signing Request (CSR) is defined on PKCS #10 [20]. The PKCS #11 standard defines a *Cryptoki* API to access cryptographic operations on particular token. [24]

2.3 Trusted Platform Module

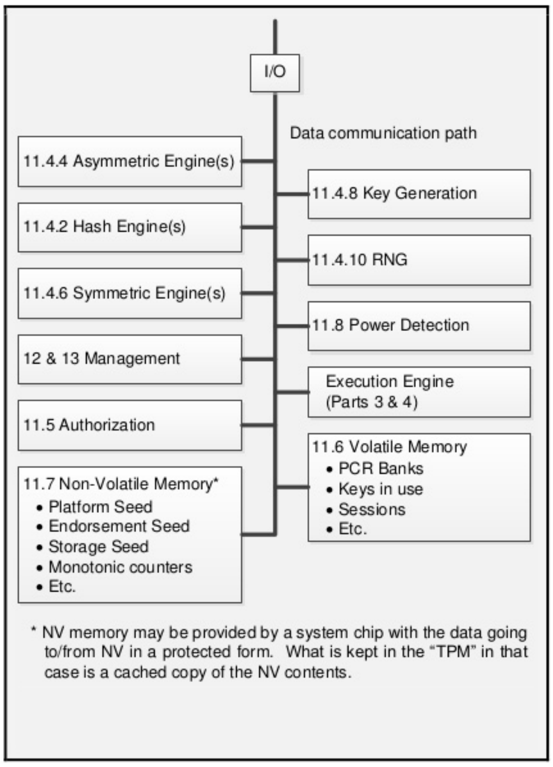
I/O
Data communication path
11.4.4 Asymmetric Engine(s)
11.4.2 Hash Engine(s)
11.4.6 Symmetric Engine(s)
12 & 13 Management
11.5 Authorization
11.7 Non-Volatile Memory*
• Platform Seed
• Endorsement Seed
• Storage Seed
• Monotonic counters
• Etc.
11.4.8 Key Generation
11.4.10 RNG
11.8 Power Detection
Execution Engine (Parts 3 & 4)
11.6 Volatile Memory
• PCR Banks
• Keys in use
• Sessions
• Etc.
* NV memory may be provided by a system chip with the data going to/from NV in a protected form. What is kept in the "TPM" in that case is a cached copy of the NV contents.

Figure 2.1: TPM Architecture Overview [29]

A Trusted Platform Module (TPM) is a security module with predefined set of cryptographic functions. These functions include symmetric and asymmetric cryptography, hash calculation, key creation, and random number creation. In addition, the module contains a memory to store keys and special registers to store platform state as seen by attestation measurements. The TPM is an isolated execution environment in which the computation is performed. Communication with the module is done on two-way I/O bus, which is the only communication channel between the host operating system and the TPM. These components are shown on Figure 2.1.

TPM specifications are defined by the Trusted Computing Group (TCG), and the first version was published by the group in 2006. Version 1.2 compatible modules have received wide acceptance among business class computer manufacturers, and have been included in virtually all enterprise-class desktops and laptops and are therefore available in tens of millions of devices [3]. Version 2.0 of the TPM specification was published in 2014 containing new features and updating the short comings of previous version. New TPM 2.0 modules are not compatible with chips conforming to the previous version of the specification. Therefore, software communicating with the TPM has to first identify the version of the module.

The TPM 2.0 contains several new features, which include support for new cryptographic algorithms, Enhanced Authorization (EA), and possibility to implement TPM functionality on software. Support for multiple cryptographic algorithms is included as previous version supported only one algorithm for each cryptographic operation. EA allows more complex object authorization schemes than the previously used password and Platform Configuration Register (PCR) based scheme.

TPM 2.0 has separated one storage hierarchy used in TPM 1.2 to four different storage hierarchies. New hierarchies separate the authorization of different usage levels and improve the management of objects. Four new hierarchies are *storage hierarchy*, *platform hierarchy*, *endorsement hierarchy*, and *null hierarchy*. The *storage hierarchy* is meant to be used by the platform owner and is equivalent in functionality to the TPM 1.2 storage hierarchy. The *platform hierarchy* is meant to be used by the BIOS and the System Management Mode (SMM). The *endorsement hierarchy* is for attestation, and the *null hierarchy* is for temporal storage.

One of the improvements is to use cryptographically bound names to identify resources in the TPM. Objects can be identified by using hash value of the object. This is an improvement to TPM 1.2 that used runtime issued identifiers, or handles, to identify objects.

TPM can store cryptographic keys and other data commonly referred as objects. These stored objects can be accessible to all or the access can be restricted. Sessions are used to authorize access to restricted objects. Objects on TPM have authentication value, which is used to compare against authorization provided by the session. Authorization can be for example a password string. Authorization is not required for all types of commands on TPM. Commands not requiring access to restricted objects inside TPM can be called without authenticated session. For example, hash and time functions do not require authentication.

Authentication to TPM 2.0 can be done with a plaintext password, with a Hash-based Message Authentication Code (HMAC) value calculated from the password or with an Enhanced Authorization (EA) policy. Plaintext password is the simplest method to use, as no session is created and the password is sent in clear-text with every command. A HMAC session protects the password by transferring only a pre-calculated HMAC value to TPM, which validates the access. With an EA session, authorization is bound to the state of TPM. EA is described in more detail on section 2.3.2. HMAC and EA sessions are started with `TPM2_StartAuthSession`[1] command. [2]

2.3.1 Platform Configuration Register

Platform Configuration Registers (PCRs) are special registers containing a hash value. They are used to store cryptographically produced measurements of platform state, verifying the software and configuration. Storing a value in a PCR is called an *extend* operation, which combines the previous hash value and new measurement with a one-way hash function.

A TPM contains 24 PCRs, where measurements from different platform components are stored in distinct registers. The platform can be shown to be in some verified state by verifying the value of each register. As the TPM does not know the expected state of platform, the register values need to be compared against trusted measurements to validate the state. To verify that expected OS version is booted the PC, each part of the boot process needs to be measured and verified. This is called *Secure Boot*.

PCRs can be modified with two operations. Register extending and register reset. Extend operation updates the register value by combining the current value with an input value. Extend is defined in Equation 2.1. Values of PCR can be described as an unbalanced binary hash tree, where new value is added as a new root node and left child is the old value and the right child is the input parameters [16, 17]. A Figure 2.2 contains a visualization of PCR values interpreted as binary hash tree where the hash function is SHA-256.

$$PCR_{new} := Hash_{alg}(PCR_{current}||digest) \tag{2.1}$$

Register reset sets the register value back to the default initial state of the register. The default initial state can be all zeros or all ones depending on the register. The platform specification can also define some other default value to present the initial platform state.

[1]All TPM2 commands are briefly introduced on Appendix A.

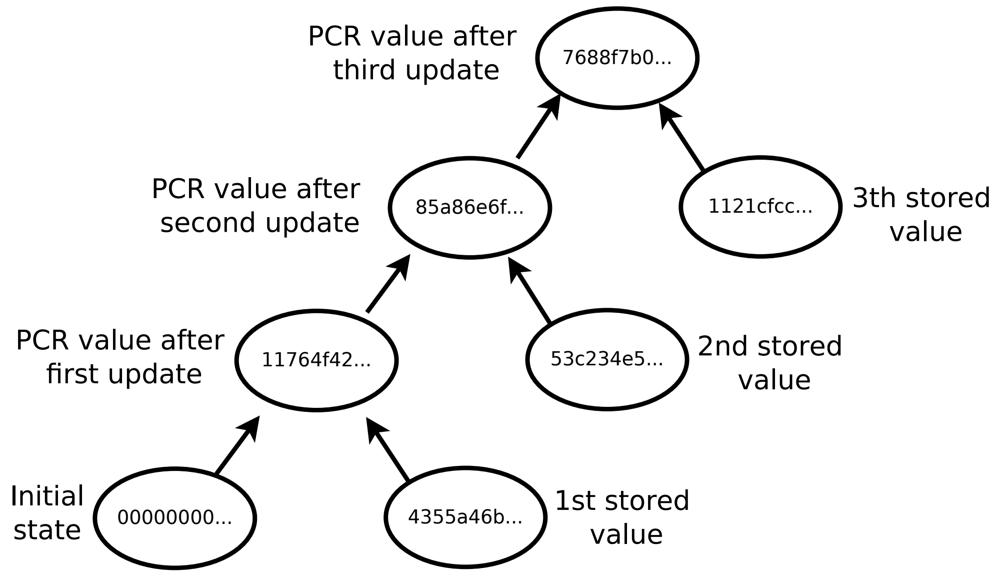

Figure 2.2: PCR interpreted as binary hash tree

TPM 2.0 can support more than one hash function and therefore multiple PCR register banks are supported. Each register bank contains 24 PCR registers for some particular hash function. Therefore a TPM can have PCR values extended with both SHA-1 and SHA-256 hash functions.

2.3.2 Enhanced Authorization

Enhanced Authorization (EA) is a new authorization method available in the TPM 2.0. An authorization can be a password, an HMAC value or a collection of EA policies. An EA policy is a condition for the state of TPM which must be satisfied to gain access. Multiple EA policies can be chained to create complex policy hierarchies. A session stores the state of the EA policy in a *policyDigest* field. EA policy grants access to an object if the *policyDigest* of session equals to the `authPolicy` field of the object.

The EA session is started with the command `TPM2_StartAuthSession` with proper parameters. This starts a session with *zero authorization* and the expected authorization is accomplished with a series of policy commands. Policy commands are used to update the authorization of the session. With policy commands, the authorization can be linked to user entered passwords, current values of PCR registers, or testing content of non-volatile memory.

These policies can be seen as tests that can pass or fail and which can be combined with logical AND and OR connectives to formulate complex policy statements. A brief introduction to used TPM commands are presented on Appendix A, which contains also policy commands.

The state of policy session, the authorization, is stored on *policyDigest* field of the session. Policy commands update the *policyDigest*, and each updated value is tied to the previous value by concatenating the previous value and the name of new command and then hashing this as the new *policyDigest* value. This is similar to PCR *extend* shown on Equation 2.1. Exception to previous update mechanism are policy commands (such as `TPM2_PolicyOR` or `TPM2_PolicyAuthorize`) which reset the *policyDigest* value to allow multiple valid *policyDigest*s to be included in authorization session. Authorization with policy session can be linked to specific state of the system by combining values in the PCR registers to the *policyDigest*.

2.3.3 TPM Software Stack

The function of the TPM Software Stack (TSS) is to facilitate application access to TPM without letting processes to disturb each other. This is accomplished by allowing only one application to access TPM at a time. Before an application can access the TPM, TSS swaps out the context of previous applications and loads the context of current application to TPM. This is done to mimic exclusive access to TPM for application. This helps further to counter the very limited memory of TPM by allowing applications to use all of the available resources on TPM, as well as limiting the access to only the current applications resources.

TSS has predefined APIs for multiple levels of access to TPM. The main goal is to provide an interoperable interface for application developers creating applications to multiple platforms. These APIs are shown on the Figure 2.3, which also shows the relationship between the TPM 1.2 and TPM 2.0 TSS implementations.

TrouSerS is an open-source project creating a GNU/Linux implementation of TSS. At the time of writing, TrouSerS only has support for older versions of the TPM/TSS specification.

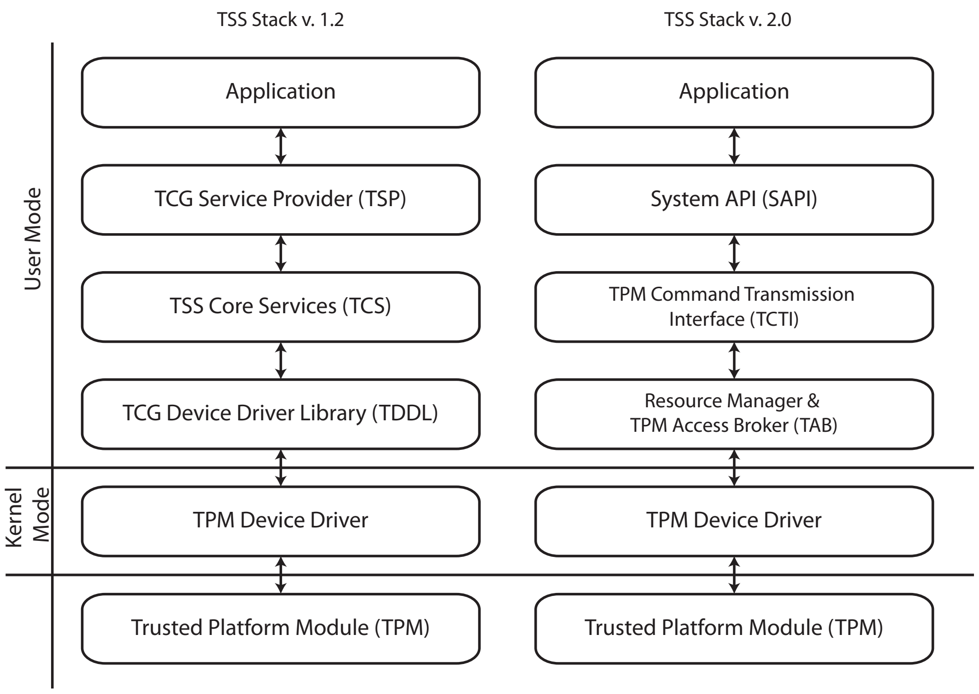
TSS Stack v. 1.2
TSS Stack v. 2.0
User Mode
Kernel Mode
Application
TCG Service Provider (TSP)
TSS Core Services (TCS)
TCG Device Driver Library (TDDL)
TPM Device Driver
Trusted Platform Module (TPM)
Application
System API (SAPI)
TPM Command Transmission Interface (TCTI)
Resource Manager & TPM Access Broker (TAB)
TPM Device Driver
Trusted Platform Module (TPM)

Figure 2.3: TSS subsystems [27, 28, 31]

2.3.4 TPM Form Factor

Discrete Chip TPM

The first version of the standard defined TPM as an integrated circuit (IC) with two possible predefined form factors. Figures 2.4 and 2.5 show the two possible form factors of TPM chips. TSSOP28 package is on the left and VQFN32 package is on the right. TPM chips are interchangeable between different manufacturers, as both the functionality and the form are fixed in specifications [29, 30].

Software TPM

The version 2.0 of TPM standard allows implementing TPM functionality on software instead of discrete hardware modules. This allows TPM functionality to be embedded on systems where a host processor supports special execution mode separating TPM from the rest of the system. This can allow

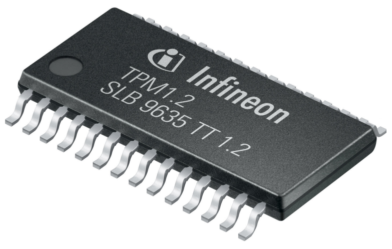

Figure 2.4: TPM 1.2 chip in TSSOP28 package [11]

Figure 2.5: TPM 2.0 chip in VQFN32 package [12]

TPMs to be included on smaller systems as no extra physical components are required. Possible systems include platforms where multiple separate execution environments are available, System on a Chip (SoC), and inside other TEE solutions. Intel has leveraged this possibility in their NUC platforms, which have implemented TPM inside Management Engine (ME). Another interesting possibility is to include TPM functionality inside GlobalPlatform Device TEE as a trusted application. Microsoft has created a design and an implementation for firmware TPM leveraging ARM TrustZone [23].

2.4 Trusted Execution Environment

A Trusted Execution Environment (TEE) is an isolated execution environment for running secure tasks. The objective of TEE is to protect assets from software attacks originating outside of TEE. Characteristics of the TEE are isolation, integrity and confidentiality for the applications running on it [10]. The application running on TEE can provide security service to regular environment. The regular execution environment runs concurrently with the TEE.

GlobalPlatform has created hardware and software architectures for Trusted Execution Environments (TEEs). The GlobalPlatform model forms a split-world environment where a trusted OS and a rich OS run side by side, one on Trusted Execution Environment (TEE), and another on Rich Execution Environment (REE). The communication between these two environments occurs through APIs defined in the GlobalPlatform device specification.

2.5 Intel Management Engine

Intel Management Engine (ME) is a separate execution environment inside Intel system chipset. This environment contains a separate OS and is running applications that are used to manage the computer. ME is used to provide remote management features called Intel Active Management Technology (AMT). As the ME is separated from main system, it is operational as long as the computer is connected to mains and can be used to control the computer even if it is turned off. Intel NUC platform uses the ME to implement a software based TPM. From ME point of view, the TPM is just one application running on ME environment.

Management Engine should not be mixed up with System Management Mode (SMM), which is an operating mode on Intel x86 processors. In this mode all normal execution is halted and a software, usually part of platform firmware, is executed.

2.6 ARM TrustZone

The ARM TrustZone is a security technology for ARM based processors to split one processor into two isolated domains. A secure and a normal domain both run their own OS, and access from a normal domain to a secure domain is possible only through the secure monitor. This monitor acts as a gatekeeper between the TrustZone domains. The separate execution domains have equal capabilities except that they have a separate memory space. A separate monitor mode controls the change from domain to domain. Applications from a normal domain can access functionality of secure part through API.

The most common scenario is to run small embedded OS on the secure domain with only very limited set of applications. This forms the small TCB which is verified against anomalies. The normal domain then runs the full sized OS, for example Android. Most of the current smart phones and tablets contain an ARM processor with TrustZone capabilities. Therefore the previously mentioned GlobalPlatform TEE is modeled after TrustZones split-world architecture.

2.7 On-Board Credentials

Kostiainen et al. present an On-Board Credentials (ObC) as a general credential storage method utilizing secure hardware to protect stored credentials. They state that the ObC could be implemented on any secure hardware and they give three example environments: TI M-Shield, TPM, and hypervisor. The security of the system is stated to be as secure as the underlying hardware. They implemented ObC on Texas Instruments' M-Shield secure hardware, used on the Nokia mobile phones. [15] Later versions of the ObC were implemented on ARM TrustZone.

The ObC system is modular and can be used to store credentials from multiple sources. The system is also designed to have no technical or policy restriction for the use. This is further emphasized with secure remote provisioning which is accessible to any party willing to setup the required systems.

ObC can also be used to store credential programs that generate the credentials at runtime, for example, a program for generating one-time passwords from protected secret seed. As some algorithms might not be publicly known, the programs can be stored also securely.

2.8 Smart cards

Smart cards are identification cards containing an embedded electronic chip for electronic identification. Smart cards have the same form-factor as used on ISO standardized identity cards. Smart cards also contain an embedded electronic chip containing a microprocessor and memory. The processor and memory are very limited compared to desktop computers and therefore the processing times could span multiple seconds for single operations. Electronics used on smart cards form a Secure Element (SE) meaning that they are packaged on a tamper-proof chip. The chip usually holds a RSA private key and identification is done by signing a challenge with the private key. Private key can only be accessed by the microprocessor on card, which guarantees the confidentiality. Smart cards are standardized by the ISO on *ISO/IEC 7816 Series - Smart Card Standards.*

Chapter 3

Design

Electronic identification with public-key encryption is based on presenting a signature, which is made with a private key. This signature acts as the proof of possession of a private key. Typically on RSA cryptosystems the signed data is a hash function of the original data. This sets four functional requirements for the implementation of Electronic Identification (eID) on TPM. Firstly, the design must be able to calculate RSA signature, which requires a public-private key-pair. Secondly, the private key should have access control in place to allow only legitimate use of key. Thirdly, this key-pair should be generated inside TPM to stay confidential. Fourthly, a hash function is needed for the signature.

Algorithmic requirements are based on most widely used secure authentication algorithms. Previously identified functional requirements lead to a set of algorithmic requirements. Firstly, a signature creation algorithm is needed conforming to *HTML5 and Digital Signatures* specification issued by Finnish population registry [8]. Secondly, a public-private key-pair generation algorithm is required. Thirdly, multiple hash functions are required to support both previously widely used algorithms and also newest algorithms.

The PKCS #11 library format is a widely accepted API for cryptographic functions. This library format is widely used to access hardware based key-stores like Hardware Security Modules (HSMs) and smart cards. For example, Firefox browser supports PKCS #11 to access keys and certificates available on external hardware. Previously identified functionalities are adjusted to fit into a format defined in PKCS #11 standard. Implemented PKCS #11 library supports multiple algorithms for each function. [24]

HTML5 and Digital Signatures specification describes an authentication flow done inside a web browser [8]. The eID solution described is going to be a link between a TPM hardware and a web browser as required by the specification. The architecture of the eID solution is presented on the right side of Figure 3.1. The eID solution is called the *PKCS #11 TPM library*. The library acts as a middleware between the browser and the OS TPM driver. The left side depicts a typical architecture used on TPM based programs.

To complete RSA functionality, encryption, decryption, signing, and signature verification are implemented. Encryption does not benefit from secure hardware, as the operation does not use private key, and it can therefore be done in software if possible.

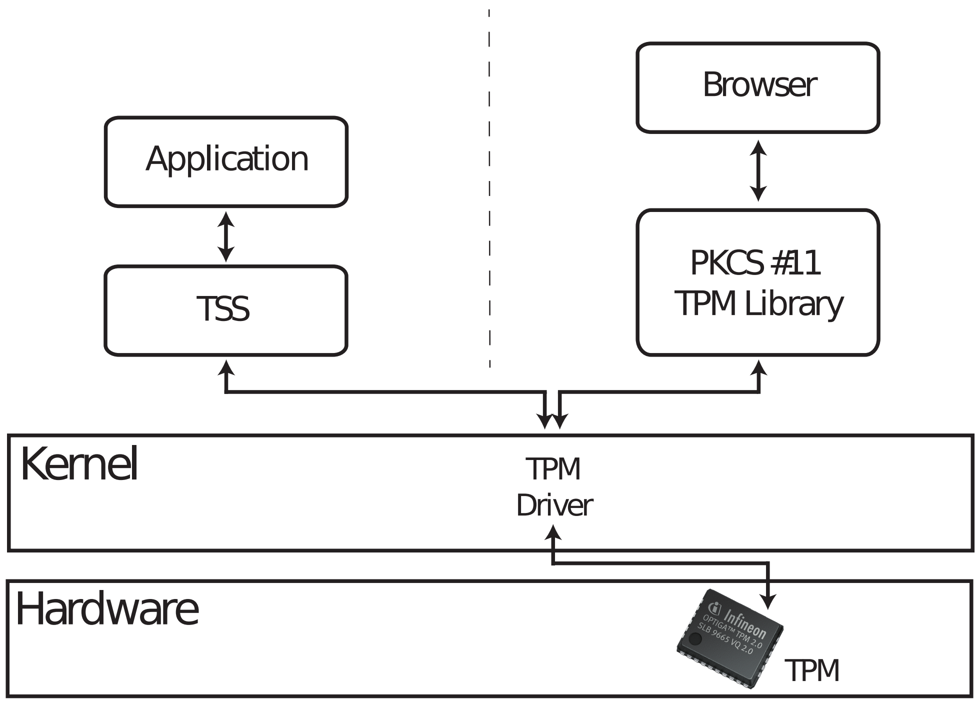

Figure 3.1: Use case

3.1 Requirements

3.1.1 Functionality

The previously identified three functional requirements can be implemented on TPM hardware securely. TPM stored keys are always a part of a key hierarchy which starts from a primary key. A key is protected by the key above it on key hierarchy. The primary key is special key type, which starts the hierarchy and therefore it has extra implied properties. The primary key is always created with attributes prohibiting the extraction of private key, prohibiting moving the private key to other hierarchy, and enabling decryption with the primary key.

Creating a primary key-pair and creating a normal public-private key-pair have their distinct commands on TPM. Both commands have same syntax and keys can be used interchangeably if the TPM key protection hierarchy is ignored.

TPM implements two separate command flows to calculate a hash function. Differences between these two flows are described in detail on section 3.2.3.

A signature in public-key cryptography is created with the private key by signing a hash value of original data. Right after the hash value is calculated on TPM the same value can be returned to TPM for signature calculation. Calculating a RSA signature on TPM is done with `TPM2_Sign` TPM command.

3.1.2 Algorithms

RSA algorithm is used to generate a signature with public-private key-pair. RSA signing scheme can only sign data with length less than or equal to key size. Therefore a hash algorithm is used to compact the data and identify it uniquely. This unique fingerprint calculated with the hash is then signed using RSA algorithm.

Used hash functions are first generation SHA for backward compatibility and second generation SHA-256 as current recommended hash function. These hash functions are implemented on the PKCS #11 library to be used on signing. Both *RSASSA-PSS* and *RSASSA-PKCS1-v1_5 RSA* signature schemes defined on RFC 3447 (PKCS #1) are supported [13].

3.2 Mapping TPM and PKCS #11 interfaces

PKCS #11 library specification is designed to be modular framework which can be extended with new algorithms. TPM interface is on the other hand an interface to very limited hardware, which implements the predefined TPM functions. Combining these two interfaces requires planning to match equal function on both sides.

3.2.1 Sessions

Sessions is the first example of dissimilarities between these two frameworks, which needs adaptation between PKCS #11 and TPM. PKCS #11 sessions are shared context for nearly all commands queried through the interface. These sessions can be read-only or read-write session where the former type is the default type for a new session. An open session can be upgraded to a more capable one by logging in. Sessions on TPM are only used as an authentication for accessing privileged objects. Commands such as hashing do not require a session on authentication at all.

3.2.2 Authentication

Authentication is the second example. PKCS #11 has `C_Login` command for authentication which requires a PIN code as a parameter. This PIN code is the secret token granting access. A PIN code is used as a legacy term in this context and the parameter can handle any kind of password string as long as the underlying device supports this.

TPM on the other hand supports three kind of authentication: password, HMAC, and Enhanced Authorization (EA). Password authentication is a plain static password communicated to TPM on every command requiring authorization. HMAC authentication is static password with HMAC value calculated before entering to TPM.[1] This can protect the used password of eavesdropping if the HMAC value is calculated beforehand. These two can be right directly combined with PKCS #11 authentication. EA bounds the particular state of the TPM to allow access to protected resources. A password can be one of the measured states on EA.

[1]The key used on HMAC is a combination of session parameters, salt, and nonces.

3.2.3 Multi-part command flow

The third example is the command flow for multi-part commands, commands that need to transfer more data than is possible in one operation. Hash functions are an example of this function class as the length of input data can be arbitrary. PKCS #11 has two alternatives for multi-part command flows.

`C_`*`Command`*`Init` ⇒ `C_`*`Command`* or
`C_`*`Command`*`Init` ⇒ `C_`*`Command`*`Update` ⇒ `C_`*`Command`*`Final`

On former case, the `C_`*`Command`*`Init` has arguments for the algorithms or keys. Second part of the command then contains all the data and terminates the command. On latter case, `C_`*`Command`*`Init` is equal to previous example, `C_`*`Command`*`Update` contains the data partitioned to multiple fragments, and `C_`*`Command`*`Final` terminates the process with last fragment.

As previously mentioned, TPM has two separate command flows to calculate a hash function. If the hashed data can fit in one TPM command, then `TPM2_Hash` can be used.[2] If the data is larger than approximately 2 kB then multi-part hashing is required. Multi-part hashing is divided into three phases, start of the sequence where the algorithm is defined, update sequence where more data is fed to hash function and complete phase where the last increment of data can be given and which returns the calculated hash value. Commands for multi-part hash are `TPM2_HashSequenceStart` command following with zero or more `TPM2_SequenceUpdate` commands and finally `TPM2_SequenceComplete` command.

As PKCS #11 always requires at least two commands to issue a multi-part command, e.g. sign, this forces the TPM as well use multi-part commands. `C_`*`Command`*`Init` on PKCS #11 interface maps to start command on TPM and `C_`*`Command`* to update and complete commands. The three-part PKCS #11 commands are mapped as they are to equivalent TPM commands.

[2] In *TCG PC Client Platform TPM Profile (PTP) Specification* a maximum command size is defined to be "large enough to support the largest implemented command". Largest mandatory command is `TPM2_ContextLoad` with 2074 bytes. [30, p. 54]

Chapter 4

Implementation

The previous chapter described a design for a software library communicating between the web browser and a TPM. On this chapter a detailed description of the library is provided.

This library communicates directly with the Linux kernel TPM device driver. This is contrary to typical applications using TPM through TSS interfaces as shown on Figure 2.3 on page 19. The decision was forced by the lack of TSS implementation on Linux for TPM 2.0 at the time of writing. This restricts other applications from accessing TPM as long as this library is running. Similarly multiple instances of this library cannot run simultaneously.

Implementation is programmed with C language and source code conforms to C89 standard. To use TPM 2.0 hardware on Linux, at least a Linux kernel version 4.0.0 is required. This is the first version to include TPM 2.0 device driver.

4.1 Architecture

Implementation of the library is divided into two modules, each with their distinct operations. The TPM communication module handles the communication with the TPM hardware and provides an easy to use interface for TPM 2.0 commands. The API provided by the communication module mimics closely the command codes used by TPM 2.0. Communication module can be used to build other applications that need to access TPM functionality. PKCS #11 module implements *Cryptoki* API defined in PKCS #11 standard. Logical structure of implementation is visualized on Figure 4.1.

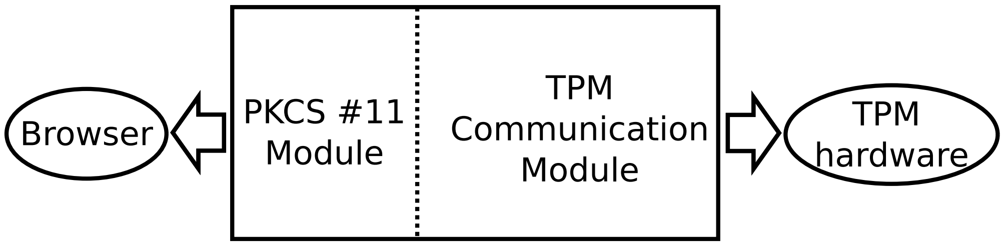

Figure 4.1: Structure of implementation

4.2 TPM communication module

The TPM communication module is a non-compatible replacement for TSS to handle low-level communication with the TPM. The module has a wrapper function for each implemented TPM2 command and handles all the details on communication.

The module communicates with the TPM through Linux character device `/dev/tpm0`. This character device is made by the kernel device driver for TPM. A typical Linux application using TPM would use TSS as a middleware.

Communication with TPM chip consists of command and response pairs. This module is a synchronous as each command execution blocks until a response is available from the TPM. Synchronous operation mode was chosen over an asynchronous as this would have increased the complexity of both the TPM communication module and applications using it.

Marshaling a buffer means transferring only the used size of buffer and not the maximum size. Variable length TPM2B types consist of a size field and a buffer field. An unmarshaled type could be 2 bytes size field and 4096 bytes buffer field. If this type is used to transfer a 2048 byte public key, then the 16-bit size field would containing value 2048, and the buffer would be 2048 bytes long. Marshaling is done to minimize communicated data between a host system and TPM. As the previous example shows, this can halve the communicated data. Buffer marshaling is done without action from developer.

The TPM specification define strictly the return value for each kind of failed operation. This helps debugging as a wrong value on input parameter has a meaningful error message. The communication module can be compiled with debug options to print return values, error messages, and even full bytecode of commands and responses.

4.2.1 TPM communication interface

The TPM communicates through a serial interface, exposes by TPM device driver.[1] Communication through this serial interface is done with a binary communication protocol. The protocol defines how TPM command and response structures are serialized for transportation.

Each TPM command has its distinct response message and no other command can be issued before a response to previous command is received. This limitation is due on the single threaded nature of TPM.

The communication interface uses a big-endian byte order, also called the network byte order. This is reversed compared to a byte order used on Intel x86 processors. This affects all fixed length parameters, i.e. `uint16_t`, `uint32_t`, and `uint64_t` and their aliases.

Keys, buffers, and strings, which do not have fixed length are transferred as variable length buffers. Variable length types consist of 16-bit size field and the data field, which size is defined by the size field. Types with variable length are named with a `TPM2B_`-prefix, e.g. `TPM2B_DIGEST`.

The TPM command structure is defined on Specification Part 1, Section 18 [29]. Commands and responses have similar structures, where they both have 4 components that are not all mandatory. Each command must start with a header containing a tag, a size of command and a TPM command code. The command code is followed by up to three handles, which are used to indicate protected objects for TPM command, e.g. signing keys. If a command is accessing protected resources on TPM, an authorization area is present after the handles. Commands having input parameters to the TPM contain also a parameter component. All commands can be divided into these four components, header, handles, authorization, and parameters.

A response starts equally with header containing the same three field as commands. On the response, the last field of header contains a response code indicating the return value of the command. Equally to commands, the second component of response is handles. If the response contains an authorization area, a ”size of parameter” field is added before the parameter component. The response data from TPM operation is passed in the

[1]TPM platform profiles describe how data from a serial like interface is transferred on platform specific hardware bus. PC computers follow the *TCG PC Client Platform TPM Profile (PTP) Specification* [30].

Field	Tag	Size of command	Command Code
Value	`TPM_ST_NO_SESSIONS`		`TPM_CC_ReadClock`
	`0x80 0x01`	`0x00 0x00 0x00 0x0a`	`0x00 0x00 0x01 0x81`

Table 4.1: TPM command header as bytecode

parameter component. The authorization area is the last component if it is included on the response.

Commands have a few variations on which components they have and what kind of information is stored in any given fields. Each command is defined in details on Specification Part 3 [29]. A `TPM2_Sign` command, having all the four components, is visualized on Table 4.2. The table contains as well information on the order, type, and size of fields. The variable length `TPM2B` fields on table are sized to fit SHA-256 hash.

The simplest commands contain only three fields required in header, tag, size, and command code. `TPM2_ReadClock` is such a command, and byte stream for this command is listed on Table 4.1. First two bytes are tag, next four indicate the size of command, and the last four are command code. Tag field indicates if the command contains one or more authorizations (sessions) and thus `authorizationSize`/`parameterSize` fields.

4.2.2 Implemented TPM functions

TPM communication module contains implementations for 31 TPM 2.0 commands. TPM 2.0 has a total of 108 defined commands. The API exposed by the communication module mimics closely the underlying API of TPM. Each implemented TPM 2.0 command has equally named function on communication module.

4.2.2.1 Functions used for eID

The implementation of electronic identification requires three functionalities. As previously stated, hash calculation is done as multi-part operation. Commands `TPM2_HashSequenceStart`, `TPM2_SequenceUpdate`, and `TPM2_SequenceComplete` are used to calculate hash functions as required. Without the multi-part requirement from PKCS #11 interface, the `TPM2_Hash` command could have been used. The signature is calculated with the `TPM2_Sign` command. The key-pair is created with `TPM2_CreatePrimary`

Component	Field	Type	Size	Offset
Header	Tag	`uint16_t`	2	0
	Size of Command	`uint32_t`	4	2
	Command code	`uint32_t`	4	6
Handles	Key handle	`TPMI_DH_OBJECT`	4	10
Authorization	Authorization size	`TPM_AUTHORIZATION_SIZE`	4	14
	Authorization Area	`TPMS_AUTH_COMMAND`	10	24
	│ sessionHandle	`TPMI_SH_AUTH_SESSION`	4	24
	│ nonce	`TPM2B_DIGEST`	22	28
	│ │ size	`uint16_t`	2	28
	│ │ buffer	`char[]`	20	30
	│ sessionAttributes	`TPMA_SESSION`	1	50
	│ hmac	`TPM2B_AUTH`	22	51
	│ │ size	`uint16_t`	2	51
	│ │ buffer	`char[]`	20	53
Parameters	digest	`TPM2B_DIGEST`	22	73
	│ size	`uint16_t`	2	73
	│ buffer	`char[]`	20	75
	inScheme	`TPMT_SIG_SCHEME`	4	95
	│ scheme	`TPMI_ALG_SIG_SCHEME`	2	95
	│ details	`TPMU_SIG_SCHEME`	2	97
	│ │ sha256	`TPMS_SIG_SCHEME_SHA256`	2	97
	│ │ │ hashAlg	`TPMI_ALG_HASH`	2	97
	validation	`TPMT_TK_HASHCHECK`	28	99
	│ tag	`TPM_ST`	2	99
	│ hierarchy	`TPMI_RH_HIERARCHY`	4	101
	│ digest	`TPM2B_DIGEST`	22	105
	│ │ size	`uint16_t`	2	105
	│ │ buffer	`char[]`	20	107

Table 4.2: `TPM2_Sign` command layout

command. This commands is identical to `TPM2_Create` command, except that it creates primary keys and does not require authorization.

4.2.2.2 Additional implemented functions

Non-volatile memory commands TPM has non-volatile memory which is usable by applications. The memory can be used for example to store certificates for the keys stored in TPM. The non-volatile memory is allocated with `TPM2_NV_DefineSpace` command, which associates specified authorization requirements to a memory block. The value of NV-memory is read with `TPM2_NV_Read` command. Similarly `TPM2_NV_Write` command writes values to the NV-memory. If a memory region is defined as a counter, then counter is updated with command `TPM2_NV_Increment` and cannot be modified with the *write* command. The non-volatile memory region is freed with the `TPM2_NV_UnDefineSpace` command.

Enhanced Authorization (EA) Commands EA commands are used to construct a policy allowing access if certain conditions are fulfilled. With policy commands the conditions can be chained into complex authorization requirements. Implemented commands are `TPM2_PolicyAuthorize`, `TPM2_PolicyAuthValue`, `TPM2_PolicyCommandCode`, `TPM2_PolicyCounterTimer`, `TPM2_policyNv`, `TPM2_PolicyOR`, `TPM2_PolicyPassword`, `TPM2_PolicyPCR`, `TPM2_PolicySecret`, and `TPM2_PolicyGetDigest`.

RSA cryptosystem The RSA public-key cryptosystem consists of sign, verify, encrypt, and decrypt operations. The sign command was introduced previously as it was used on eID implementation. The signature verification is done with `TPM2_VerifySignature` command. This command could also be implemented on software as only the public key is required. Similarly encrypt with `TPM2_RSA_Encrypt` command is possible to optimize away from slow TPM to be performed on main CPU. RSA decrypt uses private key for operation and therefore `TPM2_RSA_Decrypt` command is important to perform on TPM.

Time TPM has a monotonic counter to keep track of time. This counter advances only when the TPM is powered and therefore needs adjustment if it is needed to be aligned with real time. Additionally the hardware keeps count of number resets and restarts. `TPM2_ReadClock` command is implemented, which returns the value of three counters.

Random Number Generation TPM can be used as a hardware random number generator. Output from random number generator can be requested with `TPM2_GetRandom` command. Random number generator can accept an additional input with `TPM2_StirRandom` command to further differentiate the output.

General-Purpose Commands TPM module is initialized with `TPM2_Startup` command. This command should be called before operating with TPM for the first time. Usually TPM is initialized by the TSS framework library.

TPM specification leaves most of functions as optional and lots of implementation details to manufacturers to decide. These details can be dynamically queried with `TPM2_GetCapability` command. Capability information contains, for example, implemented commands, algorithms, and memory constrains.

The memory on TPM is very limited, which constrains simultaneously available keys or sessions on TPM memory to only a few objects. Therefore, cleaning memory of unused objects is a frequent operation which is done with `TPM2_FlushContext` command.

Password based authorization is not the only available authorization model and command `TPM2_StartAuthSession` is used to start HMAC and EA based authorization sessions. When session is no longer required command `TPM2_FlushContext` is used to remove session from TPM memory.

An external public key can be loaded to TPM with command `TPM2_LoadExternal`. This is required if a signature must be verified or a policy based authorization is extended with pre-authorized values signed by trusted third party.

4.3 PKCS #11 module

PKCS #11 module is a wrapper for TPM functionality. This module contains only the functionality to pass request and responses between the PKCS #11 interface and TPM command module. Detailed list of implemented PKCS #11 functions are listed on Table 5.3 on the next chapter.

4.3.1 PKCS #11 interface

PKCS #11 is one of the PKCS, which defines an API for public-key operations in cryptography.

Interface specification is extensive, including key generation, signing, encryption, hashing, message authentication codes, and random number generation. In each implementation one can choose which subset of the specification to implement as not all of the functionality is required by the specification. For example, a cryptographic token containing pre-issued public and private keys could be used only for signing, and expose only public key signing part of the programming interface.

A minimal implementation of PKCS #11 library would consist of only `C_GetFunctionList` function. Fully functional PKCS #11 library needs at least parts from four functions groups defined in PKCS #11 standard. Groups defined on standard are *General-purpose Functions*, *Slot and Token Management Functions*, *Session Management Functions*, and *Object Management Functions.* [24]

PKCS #11 module is commonly implemented as a shared library, which allows multiple applications to access cryptographic tokens through same interface. Each implementation of PKCS #11 shared library is tailored to a particular type of cryptographic token.

Chapter 5

Evaluation

5.1 Implementation coverage

The implemented PKCS #11 library contains only a subset of functionality presented on PKCS #11 standard. Required functionality identified in Chapter 3 are implemented and additionally calculation of hash function and random number generation are included. List of implemented PKCS #11 functions are presented on Table 5.3 at the end of chapter.

Validity of the implementations is tested using two test methods. TPM communication module is tested using unit tests for each implemented TPM command. Functionalities tested using unit tests, are listed on Table 5.3, column *TPM unit tests*. All unit tests are self-contained, meaning that they do not expect the TPM to be in any particular state. Tests are written in a manner which keeps the TPM state unchanged. This includes removal of created keys after the test, ending all active sessions, and flushing possible active sequences.

The PKCS #11 module is tested using `pkcs11-tool` from OpenSC project[1]. This tool is a command line application used to manage and use PKCS #11 security tokens. The `pkcs11-tool` has a built in capability to perform tests on the token and report if operation was carried out successfully. The fourth column on Table 5.3 contains a list of tested PKCS #11 functionalities.

These PKCS #11 tests further validate the functionality of TPM communication module. The `pkcs11-tool` command contains a predefined test set containing the expected outputs for each cryptographic operation. Tests are run through PKCS #11 interface using TPM for cryptographic oper-

[1] `https://github.com/OpenSC/OpenSC/`

ations. Running `pkcs11-tool` command against implemented PKCS #11 library uses the same interfaces as a browser would calling the library.

Size of implementation was measured using lines of code metrics which are presented on Table 5.1. These statistics were measured using CLOC utility[2].

Language	files	blank	comment	code
C	23	1244	639	5298
C Header	8	799	1006	2916
make	1	31	3	95
	32	2074	1648	8309

Table 5.1: Lines of code

5.2 Performance metrics

Performance of the TPM 2.0 and implemented PKCS #11 library was evaluated against a TPM 1.2 and a smart card. All metrics were collected in a similar manner running `pkcs11-tool` command repeatedly for each PKCS #11 module. TPM 2.0 metrics were collected from Intel NUC computer[3] where TPM is implemented as a software running probably on Intel ME. TPM 1.2 metrics were run on Infineon TPM chip[4] with Opencryptoki[5] as PKCS #11 library. EID metrics were measured from Finnish electronic identity card with OpenSC provided PKCS #11 library.

Used hash algorithms were SHA-1 on TPMs and SHA-256 on TPM 2.0. Signing was measured by creating RSA signatures with 2048 bit keys. The signature tests were not run at all on TPM 1.2. All measurements were an average of 1000 runs. Performance metrics were shown on Table 5.2.

The hash calculation was faster on TPM 1.2 chip compared to TPM 2.0, which runs on software. This leads to the conclusion that the TPM runs on Intel ME, which has a separate extremely limited execution environment compared to the main processor. The big difference on the signature creation between eID and TPM is probably due to smart card technology used on eID.

[2]`https://github.com/AlDanial/cloc`
[3]Intel NUC D34010 with software TPM
[4]Infineon SLB 9635 TT 1.2
[5]`http://opencryptoki.sourceforge.net/`

	Hash		RSA signature	
	SHA-1	SHA-256	SHA-1	SHA-256
TPM 1.2	$0.02s \pm 0.00$	-	-	-
TPM 2.0	$0.06s \pm 0.00$	$0.06s \pm 0.00$	$1.36s \pm 0.00$	$1.36s \pm 0.00$
eID	-	-	$4.25s \pm 0.00$	$4.25s \pm 0.00$

Table 5.2: Performance of hash and RSA signature tested through PKCS #11 interface

5.3 Results

The goal was to implement a TPM based eID solution usable from web browsers. Chapter 3 identified the required functionalities of TPM and algorithms used for web based authentication. Based on these, the implementation was described in chapter 4 and evaluated in chapter 5.

In the light of measurements provided in previous chapter, the TPM based authentication is evidently faster than the compared smart card. The mean time to compute 2048 bit RSA signature on TPM is 1.36 seconds. This is almost three times faster than the 4.25 seconds required on smart card.

Table 5.3: Implemented PKCS #11 functions

PKCS #11 function	Implemented	TPM Unit tests	PKCS #11 interface tests
C_Initialize	✓		✓
C_Finalize	✓		✓
C_GetInfo	✓		✓
C_GetFunctionList	✓		✓
C_GetSlotList	✓		✓
C_GetSlotInfo	✓		✓
C_GetTokenInfo	✓		✓
C_WaitForSlotEvent			
C_GetMechanismList	✓		✓
C_GetMechanismInfo	✓		✓
C_InitToken			
C_InitPIN			
C_SetPIN			
C_OpenSession	✓		✓
C_CloseSession	✓		✓
C_CloseAllSessions	✓		✓
C_GetSessionInfo	✓		✓
C_GetOperationState			
C_SetOperationState			
C_Login	✓		✓
C_Logout	✓		✓
C_CreateObject			
C_CopyObject			
C_DestroyObject	✓	✓	
C_GetObjectSize			
C_GetAttributeValue	partial		
C_SetAttributeValue			
C_FindObjectsInit	✓		✓
C_FindObjects	✓		✓
C_FindObjectsFinal	✓		✓
C_EncryptInit			
C_Encrypt			
C_EncryptUpdate			
C_EncryptFinal			

PKCS #11 function	Implemented	TPM Unit tests	PKCS #11 interface tests
C_DecryptInit			
C_Decrypt			
C_DecryptUpdate			
C_DecryptFinal			
C_DigestInit	✓	✓	✓
C_Digest	✓	✓	✓
C_DigestUpdate	✓	✓	✓
C_DigestKey			
C_DigestFinal	✓	✓	✓
C_SignInit	✓	✓	✓
C_Sign	✓	✓	✓
C_SignUpdate	✓	✓	✓
C_SignFinal	✓	✓	✓
C_SignRecoverInit			
C_SignRecover			
C_VerifyInit	✓	✓	✓
C_Verify	✓	✓	✓
C_VerifyUpdate	✓	✓	✓
C_VerifyFinal	✓	✓	✓
C_VerifyRecoverInit			
C_VerifyRecover			
C_DigestEncryptUpdate			
C_DecryptDigestUpdate			
C_SignEncryptUpdate			
C_DecryptVerifyUpdate			
C_GenerateKey			
C_GenerateKeyPair	✓	✓	
C_WrapKey			
C_UnwrapKey			
C_DeriveKey			
C_SeedRandom	✓		✓
C_GenerateRandom	✓	✓	✓
C_GetFunctionStatus			
C_CancelFunction			

Chapter 6

Discussion

6.1 Security compared to smart card

The access to smart card is protected by a PIN code, which is required to use keys. Equally, keys in TPM are protected by a PIN code or a password. This PIN code is typed using keyboard on both the smart card and TPM. The keyboard and the operating system are not verified and could be malicious and capture the typed PIN code. Some smart card readers use build-in keypads to mitigate the operating system from spying on PIN code. For TPM, normal keyboard is the only available input method. Still, remote attestation could be used to attest to the OS. In the end, PIN code security of the TPM and smart card could be equal and both techniques could counter the PIN code capture.

The Secure Element (SE) is a separate chip usually embedded on smart cards or mobile SIM cards. The distinct properties of SE include the tamper-proof chip. On the other hand discrete TPM chips are not required to be tamper proof. Software based TPMs run on separate execution environments or distinct processor modes but are not mandated to run on trusted hardware [29, Part 1, p. 41]. Hence, TPM based solution has lower physical security properties compared to SE, which needs to take into consideration.

The security of the TPM 2.0 based eID implementation can be enhanced significantly by combining it with an operating system attestation and TPM Enhanced Authorization (EA). The previously mentioned PIN code interception is possible as the BIOS, bootloaders and operating system cannot be ensured to be trustworthy. By using EA, the access to keys could be linked to a particular state of the platform stored on the PCR memory. With the help of software vendors, this could be extended further to include not just one state but all the possible unaltered states of the software stack. If the OS is measured to be unaltered and the vendor of the OS is deemed trustworthy, the PIN code interception is only possible through a bug in the OS implementation.

6.2 Usability compared to smart card

The use of smart card based eID is limited to environments in which a smart card reader is available. Additionally, USB readers are commonly available and they are easy to connect to desktops and laptops, but are unusable on mobile phones and tablets. This will give a strong advantage for a TPM based eID solution. The TPM 2.0 standard enables implementing TPM functionality inside firmware, which allows TPMs to be implemented on TEEs. Most of the current smart phones already include ARM TrustZone capable hardware and therefore could be easy targets on which to build a TPM. Making eID accessible to smart phones and tablets would increase the usage as more and more Internet based services are used on phones and tablets.

Based on the performance metrics presented on section 5.2, an eID implemented on a TPM is three times faster than similar smart card based solutions. This translates directly to faster login times in web browsers. As web browsers do not give any progress indicator on signature creation process, the waiting time is seen as part of the page loading time. Fast, modern web pages load in less than a second, and a 4 s login time would deteriorate the user experience. With 1-second signature creation, the web page load time could be the dominant part of the whole web page load time.

In home environments in which the computer is shared between multiple users, the TPM could have multiple identities stored inside. Multiple identities would not require multiple OS user accounts, as the currently used PKCS #11 interface can include more than one identity, and authentication is authorized with a personal password when required. On more heterogeneous environments more than one identity could be used simultaneously if required.

Unlike traditional eID smart cards, the TPM based solution could not be moved from one computer to another one as the keys are required to stay inside a TPM. To allow users to authenticate themselves on multiple computers, a separate key and certificate would be needed on each computer.

This would require another method to authenticate the user using strong authentication to bootstrap the trust on new computers.

The most obvious limitation is also the requirement for a TPM. At the moment, TPMs are included in virtually all business class desktop and laptop computers, but only a few consumer grade computers contains the chip. This will probably change as TPM 2.0 allows firmware based TPMs, which Intel has already used on their NUC platform. Similarly, Microsoft has included TPM 2.0 on their Surface tablet series. TPM 2.0 is expected to be part of almost all new PC hardware, as Microsoft has included it in the requirements for Windows 10 hardware. Windows 10 minimum hardware requirements states that TPM 2.0 is required for Windows 10 Mobile OS and, after 1-year transitional period, also for Windows 10 desktop version. A device must meet these requirements to gain the *Windows Compatible* status. [18]

6.3 Improvements

The implemented library is a proof-of-concept and contains multiple issues. Five improvements that could be implemented in the future are described here.

Store authentication certificates on TPM A certificate used for authentication could be stored inside the TPM non-volatile memory. This would remove the need to use external storage and make all user controlled data to reside inside the TPM. Further DER encoded certificate could be divided into three parts: fields before the public key, the public key, and fields after the public key. Only the fields before the public key and after the public key need to be stored on non-volatile memory, as the public key is already stored on the TPM with the private key. Now reading the certificate from TPM could be done with two `TPM2_NV_Read` commands and one `TPM2_ReadPublic` command.

Reformat the library to be compatible with the TSS In this work, the library was implemented as a non-compatible replacement for the TPM Software Stack (TSS). This was due to lack of TSS implementation on Linux for TPM 2.0 at the time of writing. The library could be reformatted to work on top of TSS by converting TPM command module to use TPM command transmission interface (TCTI) with a small changes. Communication with

TPM device driver is centralized into `tpm2_transmit`, which handles both sending and receiving data. The functionality of the `tpm2_transmit` function could be changed to call `transmit` and `receive` functions of TCTI. Additionally, TCTI `finalize` function would need to be called before the library shutdown. This would make the implemented library compatible with TSS APIs, which could help porting the software to other platforms. For example, Windows has a TSS implementation available, which would be an easy target. Further, the use of TSS would allow multiple applications to use the TPM at the same time. Intel published an open-source TSS implementation on July 2015[1].

Limit password guessing The current implementation does not limit how many times a wrong password can be typed, enabling brute-force attack to recover the password. Brute-force attacks could be mitigated by implementing a EA based access control, where a policy defines how many times incorrect password could be typed before key is locked. This type of system was defined by Nyman et al. using complex EA policies and multiple sessions inside the TPM to separate the privileges to increment the counter, reset the counter, and access the keys [19]. Shao et al. have performed a formal analysis of TPM 2.0 Enhanced Authorization (EA) and they found a misuse case in this system, which leads to a time-of-check time-of-use (TOCTOU) attack [26].

Create a proof of TPM stored keys One of the benefits of TPM compared to software based cryptography is the secure key creation and storage. The key can be created inside TPM using a template specified by the user. The template is an additional input to the key creation algorithm, which combines it with TPM based secrets to create a new unique key. The TPM can create a proof to assure a remote system, that the keys are stored securely inside the TPM. The proof could be created by signing a key and key parameters with an Endorsement Key (EK) stored only inside TPM, which has a certificate from the TPM manufacturer stating this. The proof is created with `TPM2_Certify` command, and based on the proof a remote party can verify the key storage mechanism [29, Part 1, page 28].

[1]`https://github.com/01org/TPM2.0-TSS`

Store keys on persistent key hierarchy Current implementation uses TPM *null hierarchy* to store the keys. A proper storage hierarchy must be used before the system is operational as the null hierarchy is not persistent and is cleared on every hardware restart. A similar issue is the usage of primary keys as eID identity keys. On production ready system the keys would be created in TPM Platform hierarchy in which they could be stored on their own subtree. The eID software would create a primary key which contains all the identity keys as sub keys. This requires implementing a `TPM2_Create` command, which is not implemented in current work. The required work is minimal as the `TPM2_CreatePrimary` command is almost identical to `TPM2_Create` and can therefore be used as a template.

Bibliography

[1] Act on Strong Electronic Identification and Electronic Signatures (Laki vahvasta sähköisestä tunnistamisesta ja sähköisistä allekirjoituksista 617/2009), 2009. Available at http://www.finlex.fi/en/laki/kaannokset/2009/en20090617.pdf.

[2] Arthur, W., Challener, D., and Goldman, K. *A Practical Guide to TPM 2.0.* Apress, 2015.

[3] Brink, D. Endpoint Security: Hardware Roots of Trust. Research Brief, Aberdeen Group, June 2012.

[4] Federation of Finnish Financial Services. Tupas Identification Services for Service Providers - Service Description and Service Provider's Guidelines. http://www.finanssiala.fi/maksujenvalitys/dokumentit/TUPAS_service_description_v23c.pdf, January 2011.

[5] Federation of Finnish Financial Services. Tupas Identification Service Identification Principles. http://www.finanssiala.fi/maksujenvalitys/dokumentit/TUPAS_identification_principles_v20c.pdf, December 2013.

[6] FiCom ry. Mobiiliasiointivarmenne - Varmennepolitiikka Operaattoreiden mobiiliasiointiavarmenteita varten Versio 1.1. http://www.mobiilivarmenne.fi/documents/Mobiiliasiointivarmenne-Varmennepolitiikka.pdf, April 2011.

[7] Finnish Communications Regulatory Authority. Strong electronic identification, electronic signatures and certificates. https://www.viestintavirasto.fi/en/cybersecurity/electronicidentificationandsignature.html, October 2013.

[8] Finnish Population Register Centre. HTML5 and Digital Signatures - Signature Creation Service 1.0.1. `https://eevertti.vrk.fi/Default.aspx?id=0&docid=1335&action=Publish`, June 2015.

[9] Florencio, D., and Herley, C. A Large-scale Study of Web Password Habits. In *Proceedings of the 16th International Conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, pp. 657–666.

[10] GlobalPlatform Device Technology. TEE System Architecture v1.0. `http://www.globalplatform.org/specificationsdevice.asp`, December 2011.

[11] Infineon. Infineon's TPM Security Chips Are First to Receive Global TCG and Common Criteria Certification and UK Government Approval; Showing World Trust in Infineon Security Expertise for PC and Data Network Protection. `http://www.infineon.com/cms/en/about-infineon/press/press-releases/2009/INFCCS200912-015.html`, December 2009.

[12] Infineon. Infineon Expands its Trusted Computing Expertise to Mobile Devices: OPTIGA^{TM} TPM 2.0 Chips Secure Microsoft Surface Pro 3 Tablet . `http://www.infineon.com/cms/en/about-infineon/press/press-releases/2015/INFCCS201502-026.html`, February 2015.

[13] Jonsson, J., and Kaliski, B. RFC 3447: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. `https://tools.ietf.org/html/rfc3447`, February 2003.

[14] Koeberl, P., Schulz, S., Sadeghi, A.-R., and Varadharajan, V. TrustLite: A security architecture for tiny embedded devices. *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014* (2014).

[15] Kostiainen, K., Ekberg, J.-E., Asokan, N., and Rantala, A. On-board Credentials with Open Provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (New York, NY, USA, 2009), ASIACCS '09, ACM, pp. 104–115.

[16] Menezes, A. J., Vanstone, S. A., and Oorschot, P. C. V. *Handbook of Applied Cryptography*, 5th ed. CRC Press, Inc., Boca Raton, FL, USA, 2001.

[17] Merkle, R. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed., vol. 293. Springer Berlin Heidelberg, 1988, ch. Lecture Notes in Computer Science, pp. 369–378.

[18] Microsoft. Windows Hardware Dev Center: Minimum hardware requirements for Windows 10. `https://msdn.microsoft.com/library/windows/hardware/dn915086%28v=vs.85%29.aspx`. Accessed: 2015-12-16.

[19] Nyman, T., Ekberg, J.-E., and Asokan, N. Citizen Electronic Identities Using TPM 2.0. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices* (New York, NY, USA, 2014), TrustED '14, ACM, pp. 37–48.

[20] Nystrom, M., and Kaliski, B. RFC 2986: Public-Key Cryptography Standards (PKCS) #10: Certification Request Syntax Specification Version 1.7. `https://tools.ietf.org/html/rfc2986`, November 2000.

[21] Police of Finland. Applying for an identity card. `http://poliisi.fi/identity_card/applying_for_an_identity_card`. Referenced on 4th of April 2016.

[22] Police of Finland. Issuing an identity card to a foreign citizen. `http://poliisi.fi/identity_card/applying_for_an_identity_card`. Referenced on 4th of April 2016.

[23] Raj, H., Saroiu, S., Wolman, A., Aigner, R., Cox, J., England, P., Fenner, C., Kinshumann, K., Loeser, J., Mattoon, D., Nystrom, M., Robinson, D., Spiger, R., Thom, S., and Wooten, D. fTPM: A Firmware-based TPM 2.0 Implementation. Tech. Rep. MSR-TR-2015-84, November 2015.

[24] RSA Security Inc. Public-Key Cryptography Standards (PKCS) #11: Cryptographic Token Interface Standard (Cryptoki) V2.30, April 2009.

[25] RSA Security Inc. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Standard V2.2, October 2012.

[26] Shao, J., Qin, Y., Feng, D., and Wang, W. Formal Analysis of Enhanced Authorization in the TPM 2.0. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2015), ASIA CCS '15, ACM, pp. 273–284.

[27] Trusted Computing Group. TCG Software Stack (TSS) Specification version 1.20 Errata A. `https://www.trustedcomputinggroup.org/resources/tcg_software_stack_tss_specification`, March 2007.

[28] Trusted Computing Group. TCG Software Stack Feature API Family 2.0, Level 00 Revision .12. `https://www.trustedcomputinggroup.org/resources/tss_feature_api_specification`, November 2014.

[29] Trusted Computing Group. Trusted Platform Module Library Specification, Family 2.0, Level 00, Revision 01.16, Parts 1-4. `https://www.trustedcomputinggroup.org/resources/tpm_library_specification`, October 2014.

[30] Trusted Computing Group. TCG PC Client Platform TPM Profile (PTP) Specification, Family 2.0, Level 00, Revision 00.43. `https://www.trustedcomputinggroup.org/resources/pc_client_platform_tpm_profile_ptp_specification`, January 2015.

[31] Trusted Computing Group. TSS System Level API and TPM Command Transmission Interface Specification Family 2.0, Level 00, Revision 01.00. `https://www.trustedcomputinggroup.org/resources/tss_system_level_api_and_tpm_command_transmission_interface_specification`, January 2015.

[32] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. Intel virtualization technology. *Computer 38*, 5 (May 2005), 48–56.

Appendix A

Introduction to common TPM2 commands

`TPM2_StartUp`	Initialize the TPM.
`TPM2_GetCapability`	Inquire implementation specific information from TPM.
`TPM2_FlushContext`	Remove object (key, session, sequence, etc.) from TPM memory.

`TPM2_Hash`	Calculate hash function in one operation.
`TPM2_HashSequenceStart`	Calculate hash function from large data buffer.
`TPM2_SequenceUpdate`	Append data to sequence.
`TPM2_SequenceComplete`	Append data to sequence and close the sequence.

`TPM2_Create`	Create new key-pair on TPM.
`TPM2_CreatePrimary`	Create new primary key-pair on TPM. Primary keys form a key hierarchy and therefore do not have parent key.
`TPM2_LoadExternal`	Load external public or public-private key into TPM.
`TPM2_ReadPublic`	Read public part of a key-pair.
`TPM2_Sign`	Cryptographically sign a hash created by TPM.
`TPM2_VerifySignature`	Verify a cryptographic signature.

`TPM2_StartAuthSession`	Start *HMAC* or *policy* based authorization session on TPM.
`TPM2_PolicyRestart`	Returns the policy authorization session to its initial condition.
`TPM2_PolicyAuthorize`	If the current *policyDigest* value is signed with valid certificate X then the *policyDigest* is replaced with value equal to certificate X.
`TPM2_PolicyAuthValue`	Updates *policyDigest* if user can provide right *authValue* to access some restricted object.
`TPM2_PolicyCommandCode`	Limit the use of policy session to only one command code.
`TPM2_PolicyGetDigest`	Read the current *policyDigest*.
`TPM2_PolicyNV`	Updates the *policyDigest* value based on value of non-volatile memory. Can test if the value on NV memory is equal, non-equal, less than or more than some arbitrary.
`TPM2_PolicyOR`	If the *policyDigest* matches one of the input digest values, then the *policyDigest* is reset to zero and updated by the concatenation of all input digest values.
`TPM2_PolicyPassword`	Authorize the access if a valid password authorization is provided when the policy session is used.
`TPM2_PolicyPCR`	Updates the *policyDigest* value based on a value of particular PCR.
`TPM2_PolicySigned`	Update the policy if a valid signature for the parameters of the command is presented.
`TPM2_PolicySecret`	Update the policy if a knowledge of a secret is presented in a form of HMAC hash of the command parameters.

`TPM2_NV_DefineSpace`	Allocate a space from non-volatile memory.
`TPM2_NV_Extend`	Add a new value to hash chain/tree stored on NV memory. Behaves as PCRs.
`TPM2_NV_Increment`	Increment a counter stored on non-volatile memory.
`TPM2_NV_Read`	Read from non-volatile memory.
`TPM2_NV_SetBits`	Modify non-volatile memory allocated as bit field.
`TPM2_NV_UndefineSpace`	Deallocate space in non-volatile memory.
`TPM2_NV_Write`	Write to non-volatile memory.