2013

Technical University of Denmark

Master Thesis

Design and Analysis of Fair-Exchange Protocols based on TPMs

Supervisor: Sebastian Alexander Mödersheim

Author: Qiuzi Zhang
Student number: s104664

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk
IMM-M.Sc.-2013-106

Abstract

A crucial goal in electronic commerce is to guarantee the fairness for both vendors and customers, ensuring that both are bound to their part of a contract. Many fair exchange protocols have been proposed for this purpose, such as optimistic fair exchange protocols. In this thesis, we will investigate new designs of fair exchange protocols for online media purchase that use the Trusted Platform Module TPM as a trusted third party. TPMs provide a high level of security against manipulations of a user while being less expensive than other forms of trusted third parties. We will precisely describe the intruder model, the assumptions, and the desired properties of the protocols we propose, allowing for formal reasoning about their security and reliability. In addition, we will also consider how zero-knowledge protocols can be used to make fair exchange protocols more privacy friendly.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc.
The thesis deals with fair exchange protocols with online media purchase and how it can be developed based on the Trusted Platform Module (TPM) chips.

Lyngby, 02-September-2013

Qiuzi Zhang

Acknowledgements

My sincerest thanks to my supervisor Sebastian Alexander Mödersheim. Without his support and guidance, this dissertation could never been finished. I also thank Per Friis for helping me rebooted the system and re-enabled the TPM chips many times. Without his patience and help, I can never to familiarize with TPM in such short time.

Contents

Table of Figures

1. Introduction

1.1 Background

In the past decade, the electronic commerce has become more and more common. Recent surveys show that e-commerce as a gradually matured market, the largely increasing number of people has started using electronic transactions. In 2012, e-commerce sales reached $1.08875 trillion, is up 21.9% from 2011[1]. In U.S, e-commerce was nearly $225 billion in 2012, and China as a most potential market reached $207 billion[2]. Along with the development of e-commerce, network security is getting more and more attention. Especially, in the electronic commerce, how to guarantee the strong fairness and security for all participants is even more important. Consider this scenario: Alice wants to download a movie from a movie website, and it cost 10$. There are two potential possibilities, one is, Alice paid money to vendor, but she still cannot download this movie, because the vendor denies that he received the payment. Another possibility is that vendor release the movie and never gets the payment from Alice. In order to avoid such problems, the non-repudiation is designed for achieving such goals.

Non-repudiation refers to the ability to guarantee that the sender of a message cannot deny previous commitments or action and the recipient cannot deny having received the message. Since no security technology is absolutely foolproof, when disputes arise because of a party denying that certain actions were taken, a procedure involving a trusted third party (TTP) to resolve the fairness. Due to the importance of non-repudiation, a number of protocols have been designed to achieve this goal. All relevant theoretical perspectives will be briefly presented in the following chapters and in this thesis we will propose a new non-repudiation protocol, which is based on Trusted Platform Module (TPM). The TPM is a hardware chip, which is embedded into a computer to securely store cryptographic keys, certificates or other sensitive information. It can also authenticate the platforms.

1.2 Scope of Study

Most recent studies showed that the hackers are getting more interested in client PCs, because with the rise of online banking and online payment, stolen username and password can enable hackers to take a huge amount of money from a user account. In order to protect user's sensitive information and ensure the strong fairness, we

propose a new non-repudiation protocol based on TPMs. According to the previous research, all existing non-repudiation protocols ensuring strong fairness have to assume that all participant entities has resilient channel with the trusted third party, when this condition can not be guaranteed, all these protocols becoming invalid[3]. So in this study we assume all computers have installed a TPM chip, and the TPM plays a role as an online TTP. In digital contract signing section, the TPM is used as a trusted third party to verify digital signatures and the validity of the contracts.

However, although TPM has many functions, it also has some limitations. Therefore, how to combine TPM with Non-repudiation together to achieve greater levels of security is the primary problem of this study.

1.3 Protocol verification

Due to the difference of models and every models language has its strengths and weaknesses. Therefore, when deciding on the methods to use, many factors was evaluated. For design part, the activity diagram has been chosen for present the workflow of the protocol. Because activity diagram can provide a good overview of protocol process and also can be used for identify alternative scenarios. For protocol verification part, we chose Open Source Fixed point Model Checker (OFMC). OFMC is a very useful tool for verifying the protocol with a formal specification language called AnB language. It is very easy to use.

For experimentation, the software stack TrouSerS has been chosen to communicate with TPM. TrouSerS is the most popular way to interact with TPM, is easy to control and working with c programming language. More details will present in Chapter 5.

1.4 Structure of the Report

This dissertation comprises six chapters. The first chapter provides a basic introduction to the study and briefly presents the problem of the study and research method.

Chapter 2 provides the relevant theoretical work. The chapter explains the background of non-repudiation protocols, already existing protocols and other relevant theories.

Chapter 3 describes how the TPM works, the architecture of the TPMs and the hardware components of the TPMs.

Chapter 4 gives a detail of description about our new protocol. This chapter contains three sections, each section represents one design version.

Chapter 5 gives a result of the implementation and describes the method we have used.

Chapter 6 contains a discussion of the primary results of the study. Analyzed the limitation of the protocol. In the end of the chapter, we discuss the possibilities of the protocol in the future.

2. Related theories

2.1 Non-repudiation

2.1.1 Introduction

Non-repudiation is a property that ensures all participants cannot deny their signature on a document or actions in a digital transaction. In other words, when two parties sign a contractual text over the public Internet, no matter what happens, only two results can end this transaction. Both of parties got the valid contract that they expected, or neither one does. The primary feature of non-repudiation is evidence validity. It means when Alice sends a message to Bob, the non-repudiation evidence of Alice should be delivered to Bob, and also the non-repudiation evidence of Bob is delivered to Alice. When dispute arises, a judge can examine this evidence to resolve the disputation. In the Zhou Gollmann non-repudiation protocol [9], the authors classified the non-repudiation property into four types:

- Non-repudiation of origin (NRO): provides the evidence of origin (EOO) for recipient, in case the originator denies the message he has sent. The EOO is generated by originator and held by recipient.
- Non-repudiation of receipt (NRR): provide the evidence of receipt (EOR) for originator, in case the recipient denies the message he has received. The EOR is generated by recipient and held by originator.
- Non-repudiation of submission (NRS): this service applies only when the TTP is involved, it provides the evidence of submission, which it is generated by the TTP (also called delivery agent) and held by originator.
- Non-repudiation of delivery (NRD): provides the evidence of delivery. It is generated by TTP and held by originator. Normally, the NRS and NRD refer to the optimistic protocols, which will describe in the following sections.

2.1.2 Fairness

The most challenging part of non-repudiation protocols is to avoid one of the participants to get an unfair advantage. Because if the two participant entities do not trust each other, then no one wants to send their item first. For instance, Alice and Bob want to exchange their digital item over the Internet. Alice sends request and commitment to Bob, Bob replies his commitment to Alice. Alice has no doubt, so she

sends the digital item to Bob. In this step, Bob may terminate the transaction, because he already has the item and the commitment of Alice. For this unfair situation, the solution is involving a trusted third party (TTP). The first approach of the solution is based on an inline TTP [11]. The inline TTP is acting as a delivery authority, and is involved in all message exchanges. But, along with development of electronic transactions, the heavy involvement of the TTPs raises many problems. In order to reduce the TTPs involvement, an improvement approach called online TTPs has been introduced. Online TTPs only intervenes in each communication session. However, this approach still heavy for transaction, therefore, Asokan et al.[8] proposed a new lightweight TTP called off-line TTPs. The off-line TTPs is involved only in the situation that misbehaviors occur. The off-line TTP is also called the optimistic approach [8], as it assumes most participant entities are honest and they all follow the protocol correctly. The TTP intervenes only the dispute arises.

2.1.3 Timeliness

Another important feature of the non-repudiation is timeliness. The network sometimes delay or one of the entities holding back the message intentionally. The deadline to limit time should be set. The timeliness guarantees all participant entities always finish the protocol in a reasonable amount of time. If timeout, no matter at any state of protocol, each party can stop the protocol without loss of fairness.

2.2 Non-repudiation protocol based on smart card

2.2.1 Introduction

Because many protocols are aiming for the goal of non-repudiation and all these protocols have respective strengths and weaknesses, it is hard to analyse and compare them all. Therefore, in this dissertation we decided only represent the non-repudiation protocol, which is based on receiver-side smart cards. This protocol was proposed by Jing Liu et al. in 2009. In nowadays network environment, the author deemed that it is really hard to ensure strong fairness for mobile user (like ad hoc network, no one can guaranteed there has available infrastructure), especially when an electronic item is time sensitive. For solving this kind of problem, the author suggested using a temper resistant processor to take over the role of a dedicated TTP, like smart card[3]. The advantage of doing so is that can ensure strong fairness when only unreliable channels can be guaranteed or no infrastructure is available.

2.2.2 Approach

As mentioned 2.1.1, the non-repudiation property defines three main entities. In this

protocol the author treats vender as Originator, consumer as Recipient and smart card as unilateral online TTP. All communication channels between vendor and consumer are unreliable. The protocol has two sub-protocols: encryption key establishment sub-protocol and exchange sub-protocol. The goal for key establishment sub-protocol is freshness. For every new sessions of the protocol, the smart card (T) will generate a key K for encrypting the message M. This key K will be added into a unique transaction label L. Whenever a problem occurs, the smart card (T) could check L for ensuring whether a request is new or old. The goal for exchange sub-protocol is fairness. When vendor (O) receives key K, he checks the signature of both consumer (R) and smart card (T), if all these signatures are valid then O will generate a message which includes cipher C (the exchange message M with encryption key K), a hash value h (K) and evidence of O (EOO). When R received this message he will add his own evidence (EOR) into the message and forward to T. T checks the signature of evidence EOO and signature of evidence EOR, and also use h(c) and locally K to compute a value L. If all evidence are valid and label L is correct, then T signs the evidence EOR for R and R forward this message to O. So now T hold evidences of both O and R. If R received a formal acknowledgement from O then T will release K for R and terminate protocol, otherwise, T will sends rebuttal evidence Rebuttal for R. For the alternative scenario, if the item is time-sensitive, then when R received formal acknowledgement from O, T will first asks R if the item still available, if so then T will release key K, otherwise, T will sends Rebuttal to R.

2.3 ASW protocol

The ASW protocol was proposed by Asokan, Shoup and Waidner.[13] The goal of the protocol is enable two parties to reach an agreement on the contractual text and digital signing it. The protocol is an optimistic fair exchange protocol, which means the trusted third party is only involved if dispute rises. Based on [12,13] the authors formulate several protocol requirements.

- **Effectiveness**: if both originator O and responder R follow the contract correctly and both of them do not want to abandon the current protocol. Then, both of them have a valid contract when the protocol has finished.
- **Fairness**: when the protocol finished, either two parties have a valid contract, or no one has. For instance, if agent O aborted the protocol, then the agent R cannot possess the valid contract.
- **Timeliness**: Agent O and R must be sure that the protocol will be finished in a finite amount of time.
- **Non-repudiabilitiy**: after the protocol finished, both parties will be able to prove

that other party has accepted the contract.

The ASW contains three sub-protocols: exchange sub-protocol, abort sub-protocol and resolve sub-protocol. The notations used in ASW shows in the table 2.
Table 1

Sigo(M)	Digital signature of message M with private key of agent O
Vo,V_R	The verifications of public keys that are corresponding with the signing keys
T	Trusted third party
No,N_R	The nonce are freshly generated by O and R
text	Contractual text
h()	Cryptographic hash function

The exchange sub-protocol only involves two parties: the originator O and the responder R. The trusted third party as an off-line TTP only involves in dispute resolution.

Exchange subprotocol:

1. $O \rightarrow R: \quad me_1 = Sig_O(V_O, V_R, T, text, h(N_O))$
2. $R \rightarrow O: \quad me_2 = Sig_R(me_1, h(N_R))$
3. $O \rightarrow R: \quad N_O$
4. $R \rightarrow O: \quad N_R$

Abort subprotocol:

1. $O \rightarrow T: \quad ma_1 = Sig_O(aborted, me_1)$
2. $T \rightarrow O: \quad ma_2 = \text{if } resolved(me_1) \text{ then } Sig_T(me_1, me_2)$
   $\text{else } Sig_T(aborted, ma_1) \; ; \; aborted(ma_1) = true$

Resolve subprotocol:

1. $O \rightarrow T: \quad mr_1 = me_1, me_2$
2. $T \rightarrow O: \quad mr_2 = \text{if } aborted(me_1) \text{ then } Sig_T(aborted, me_1)$
   $\text{else } Sig_T(me_1, me_2) \; ; \; resolved(me_1) = true$

Figure 1
The figure 1 has been taken from "ASW protocol revisited"[12], it shows sub-protocols of ASW.

Exchange sub-protocol:

1. Originator generate a nonce No, which is called secret commitment[12]. And he hashes this value to be the so-called public commitment. Then he add this public

commitment h(No) in the signed message and sends it to responder R.

2. Responder also generate the secrete commitment N_R and hashes it to the public commitment $h(N_R)$. Then he sends the signed message to originator with the public commitment.
3. 4. Originator and responder exchange their respective secret commitment.

Afterward, each party can hash this nonce to verify that the secret commitment is corresponding to the public commitment he received. If the evaluate to be true, then both parties have a valid standard contract. Otherwise, each party can invoke TTP to abort or replace the contract.

The abort sub-protocol and resolve sub-protocol are as same as the one presented in the section 4.3.2. The TTP is assumed to have a permanent database of contract. When originator sends abort request to TTP, based on the database registration the TTP will be able to decide whether aborts contract or replies with a replacement contract. Likewise, if any of parties send resolve request to TTP, the TTP will check database and if he already aborted the contract, he will replies an aborted token. Otherwise, he replies a replacement contract.

Generally speaking, no matter what happens (such as network delay, or any of the participant entities is dishonest). All participant entities can reach one of the final states within a reasonable amount of time. And there will be only two consequences: either both entities possess a valid standard contract, or no one does.

2.4 Other related knowledge

In this sub-section, we will briefly introduce RSA, digital signature and hash value, because all these concepts are relevant with the protocol and will be repeatedly mentioned in the following chapters.

RSA is a public-key cryptography for both encryption and digital signature. RSA involves a public key and private key. Everyone knows public key, and private key is only held by creator. For encryption purposes, user Alice generates a cipher C that is a message M encrypted by user Bob's public key pk(Bob) and sends it to Bob. Since only Bob has the corresponding private key, only Bob can decrypt. For authentication purpose, if Bob wants to send a signed message to Alice, he could hash the message to a hash value h (M) and sign this hash value with private key (use signing algorithm) then sends message M and S to Alice. To verify the signature, Alice uses Bob's public key and the same hash algorithm to compare the resulting hash value and message's hash value. If comparison passed, then Alice can ensure this message is generated by

Bob and has not been modified.

The digital signature is a mathematical schema for signing digital message or document. The main purpose of a digital signature is to guarantee that the message was sent by the claimed and honor party.

A Hash value is a fixed-length output that generated by hash function. The hash value is shorter than the original text itself and with the properties that hard to find a pre-image or two value colliding.

3. Trusted Computing

3.1 Introduction

The TPM has been designed for supporting trusted computing platforms, in order to understand the TPM. It is necessary to understand what the features of trusted computing are. The aim of trusted computing is to protect sensitive information, such as asymmetric keys and symmetric keys from attacker. Trusted computing assumes when client software is attacked, such sensitive keys should be protected. Therefore, in January 2000, the trusted platform module (TPM) was first proposed by TCG for providing such functions. Based on many improvements, the newest version of TPM has many nicely features. The details will present in the following sections.

3.2 Features

According to "TCG Specification Architecture Overview 1.4", the trusted platform is embedded into a computer and working with CPU. It should at least contain three basic features: Protected Capabilities, Integrity Measurement, Storage and Reporting and Attestation Capabilities.

- Protected Capabilities: compute and securely store data in trustworthy way
  Protected capabilities are a set of commands that allow to access shielded locations (like memory, register). And shielded locations only can be accessed by protected capabilities. The TPM can report and protect integrity measurement by using protected capabilities and shielded locations.

- Attestation Capabilities: a process for confirming information correctness
  The external entities can use such capabilities to attest shielded locations, root of trust and integrity of platform. TPM associate all measurement committed to the PCRs value and sign it with attestation identity key (AIK). Thus, a trusted client can prove to other parties that the platform has or has not been compromised. And for every TPM chip has a unique and secret key. Software can use it to authenticate hardware devices.

- Integrity Measurement, Storage and Reporting: most main goal for TPM is provide a trusted way to measure and report the platform environment. In trusted boot, the chip stores configuration information in the Platform Configuration Registers (PCRs) during the boot sequence. Once, the system booted, all data will be sealed with current PCRs value, the data only can be unsealed when PCRs value match with the same value of data sealed. Thus, when attacker attempts to boot in alternative system, the PCR value will not match and data cannot be unsealed. Which can protect the data from attacker or malicious code.

But TPM cannot distinguish what is secure or insecure state. It only provides measurement.

3.3 Hardware components

In order to better understand the TPM workflow, we decided to briefly introduce each components of TPM. The figure 2 shows internal architecture of TPM:

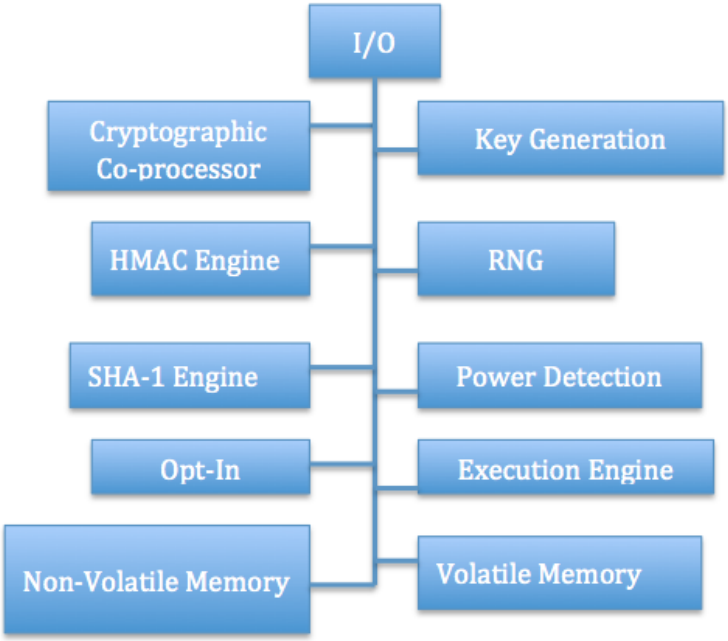

Figure 2

I/O: The input/output component controls information flow between TPM and external communication bus and also the flow between all internal components. The I/O component is also able to manage access to all the components in TPM, but this access right is determined by Opt-In.

Cryptographic Co-processor: The TPM uses a cryptographic co-processor to implement cryptographic operations, such as asymmetric key generation, asymmetric encryption and decryption, hashing and random number generation (RNG). These capabilities help TPM to perform generation of random data, signing and store data. The TPM normally use RSA algorithm for encryption/decryption and the minimum key size is 2048 bits. The TPM may also support other algorithms (such as DSA) and key size, but we are not representing more here.

Key Generation: obviously, key generation is used for creating key pairs. According the TCG specification, the key generation creates RSA key pairs and symmetric keys. The TCG did not set limitation of generation times for both symmetric and asymmetric keys. Once the asymmetric keys created, the private key will be held in a shielded location.

HMAC Engine: HMAC engine provides HAMC calculation. It can proof of knowledge of authdata (the data relevant with authorization) and also proof the arriving message is authorized and has not been modified during transition.

RNG: RNG is an internal random number generator. These random values are normally used for nonce, key generation and randomness in signatures. Typically, instead of using true random number generator, many TPMs have implemented pseudo random number generator that are using entropy from timing measurements or other source of entropy within the TPM. By doing this, the random number can be generated more effectively.

SHA-1 Engine: SHA-1 is a hash algorithm called Secure Hash Algorithm. It takes any size file as input and output 20 bytes of data. If one bit of the input data is changed, then the resultant output will be changed also. For instance: here has a new measurement of platform, and now needs associate with current PCRs value. So we take PCRs value p and measurement m as input (p||m) and then use SHA-1 hashing it.

$$p' := \text{SHA-1}(p||m)$$

The new PCRs value p' will replaces the current PCRs value and the size of p' is 20 bytes.

Power Detection: Power detection is a component for managing the TPM power state.

Opt-In: Opt-In is a component provides mechanisms and protections to allow the owner change the TPM state. The states include turned on/off, enabled/disabled, and activated/deactivated. The state only can be changed by owner.

Execution Engine: Like many temper-resistant hardware, the TPM has an execution engine for running "programme code". This piece of programme code is stored permanently on the TPM and use to initialise the device. Since the programme code is fixed by manufacturer, so we may assume that it is trustworthy.

Non-Volatile Memory: This memory is used for storing long-term keys, such as endorsement key (EK), Storage Root Key (SRK) and also stores owner authdata (like. owner's password), persistent identity and TPM state. All keys store in the non-volatile memory cannot be moved out from TPM, more details will be represented in the following section.

Volatile Memory: Volatile memory is counterpart of non-volatile memory all data are short term. The memory only retains the data as long as the power supply is on. Normally, volatile memory stores platform configuration register, attestation identity keys and loaded keys.

3.4 TPM Features

TPM provides several useful security functions that can defend against main threats in e-commerce environment. These functions are very important and directly related with our protocol. Therefore, in this section, we will dig some details about these features.

3.4.1 Key management

In order to better understand the TPM, we need first understand the key's type of TPM. The TCG defines seven key types. These keys can be characterized as migratable or non-migratable key. Migratable keys provide a function to transfer keys from one TPM platform to another. And the migratable keys can be used for more than one system. For instance, a person might want to sign the e-mail or contract at home and office or other place. Non-migratable keys are keys that are only generated inside the TPM and permanently associated with a specific TPM (such endorsement key). The keys architecture is illustrated in figure 3

- Endorsement Key (EK)

Each TPM contains a unique 2048 bit RSA key pair called Endorsement Key (EK). The EK set at TPM during manufacture time and certified by the manufacturer. The EK is permanent embedded in TPM and only can be stored in the shielded locations. The private portion of the endorsement key is never released outside of the TPM. The public portion used to unique identity the TPM. The EK cannot be used for both signing and encryption. It only can be used for AIK certificates decryption. The AIK is the unique identity keys of TPM. The detail is given in the following sections.

- Storage Root Keys

Storage Root Key (SRK) is a non-migratable storage key that embedded in the TPM. It is created when user taking ownership. The SRK is a RSA key pair that used to encrypt other keys stored in the disk. It is a root for key chain.

- Storage keys

Storage keys are asymmetric keys used for storing data, or other key types such as binding keys or signing keys. It can be either migratable or non-migratable.

- Signing Keys

Signing keys are standard RSA signature key used to sign data or message. It also can be either migratable or non-migratable.

➢ Binding Keys

Binding keys are keys used to encrypt small amount of data, normally these data are symmetric keys. The binding key are typical migratable keys, the data can be encrypted with binding key on one platform and decrypted on another.

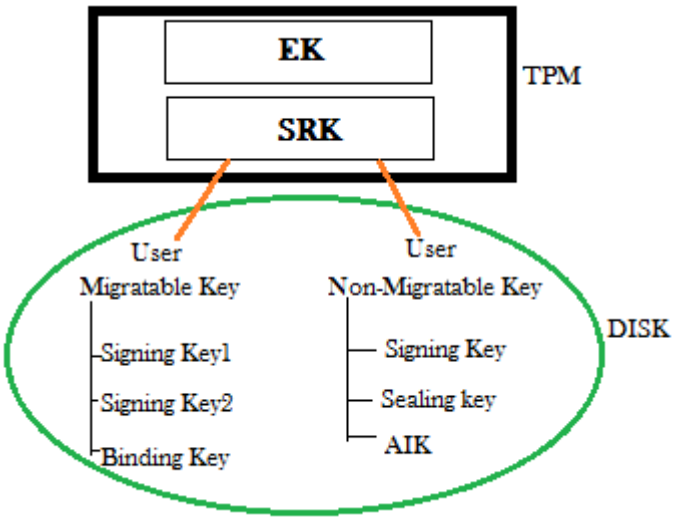

Figure 3

➢ Identity Key

Identity key are special non-migratable signing keys. They are generated inside TPM and certified by trusted third party. The identity keys are used for two purposes. One is used for signing PCRs value and another one is for signing other keys. The identity keys are always generated inside TPM and always take SRK as the parent key.

Due to the storage space limitation of the TPM, most keys and data are stored in the disk (as shown in the figure 3). All these keys and data are encrypted with parent key (like SRK), when user needs one of them, he needs load this key into the TPM and decrypt it with parent key. The more detail about storage will be represented in the secure storage section.

3.4.2 Platform measurement

➢ **Integrity measurement**

Let's assume this scenario: when vendor received an order from consumer, how can vendor be sure this consumer's PC has not been hacked, the platform is integral and under control? TCG handles this kind of problem with a chain design. TCG checks each hardware component before passing control to the next component and store the measurement as PCRs value. After the measurement finished, the user can check the system is running in an expected configuration or not. But the problem is how to perform this chain. To illustrate, assume here has three entities A, B and C. In order to trust C, we must trust B, and in order to trust B, we must trust A. But who measure A? The answer is Root of Trust. The root of trust is a small subset of the BIOS and is the

first thing to get control of the system. This small subset needs to be assumed trustworthy. The figure 4 shows workflow of the measurement.

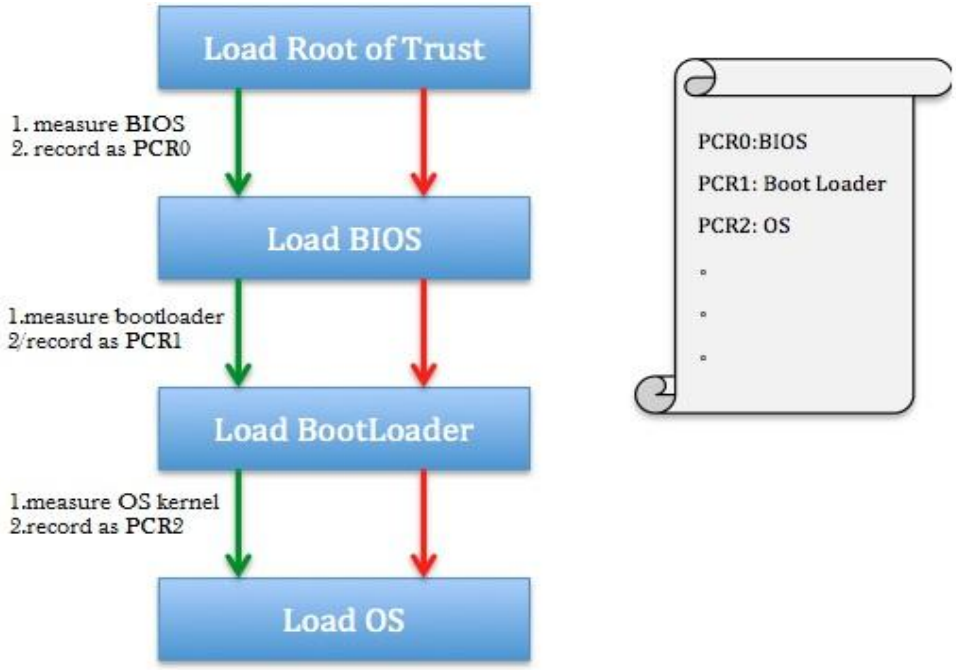

Figure 4（The red line means “passing control”, the green line means “measuring”）

The chain starts with the Root of Trust. First, the Root of Trust measures integrity of the BIOS before passing control to it. The measurement is stored in one of the PCR registers, in this case we stored as PCR 0. The BIOS measures the bootloader and stored the measurement as PCR1, then passes the control to the bootloader. The bootloader measures OS kernel, records the measurement as PCR2 and passes control to the OS. After OS is loaded, one can check the PCRs value to see if it is an expected configuration. However, TPM cannot prohibit booting into insecure OS. TPM only reports the status of the platform, and let user to decide to continue or not.

So now, back to the scenario mentioned above, when vender received an order from consumer, he can request to check the configuration of the consumer’s platform. This process is illustrates in the Figure 5.

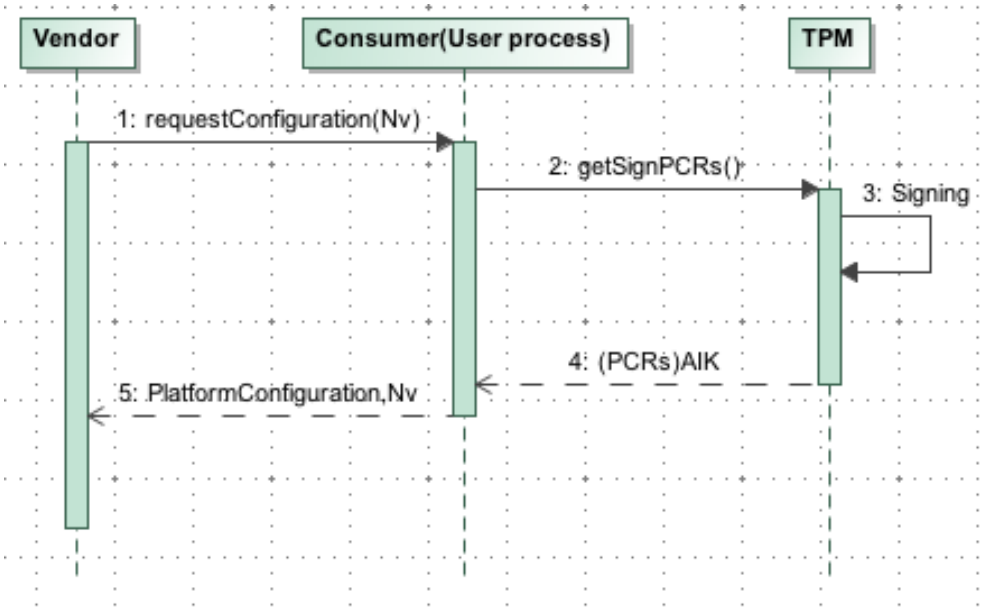

Figure 5

1. Vendor sends a configuration request to consumer, which contains a freshly generated nonce.
2. User processes requests TPM to sign PCRs value
3. TPM signs the PCRs value with an attestation identity key (AIK).
4. User processes add the nonce and forward the message to vendor.
5. Vendor checks the status of the consumer's platform and the freshness of PCRs value.

The purpose of the integrity measurement is to allow the entities to check each other's platform. The entities can compare the newly received configuration with the old one to ensure that the other's platform is running in an expected status. The entities also can determine the freshness of the message by checking the nonce.

- **Attestation platform**

Although vendor now has the consumer's platform measurement, he still cannot trust it. Because he cannot ensure this measurement is fresh and signed by the valid TPM. Therefore, the TPM needs an identity key to prove itself. As mentioned in 3.4.1, the endorsement key is a unique key for each TPM. It directly associated with the hardware. Using the endorsement key as an identity key might involve privacy problem. So instead, attestation identity keys (AIKs) are designed for signing PCRs values or other specific data. The AIK is a unique unlikable identity key and certified by a privacy CA. As illustrated in the figure 6,

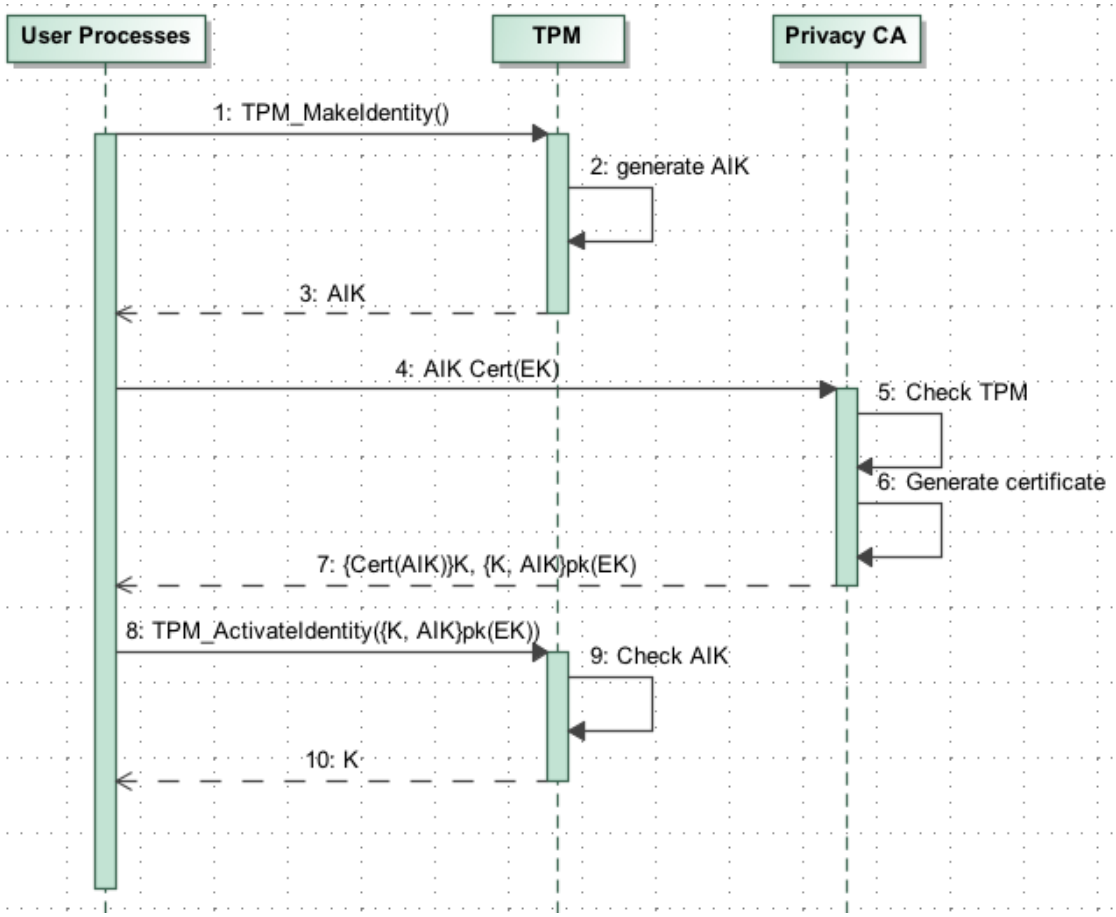

Figure 6

1. User processes invokes command TPM_makeIdentity to ask TPM to generate an attestation identity key (AIK).
2. TPM generates AIK
3. TPM sends the public portion of AIK to user processes
4. The user processes sends AIK public portion to the privacy CA together with the credential of the endorsement key.
5. The CA checks the certificate on EK (to check if it is revoked).
6. The CA signs a certificate for the public portion of AIK.
7. The CA encrypts the certificate with a newly created key K. It also encrypts the key K and the public AIK together under the public EK (which contained in the endorsement credential).
8. User processes requests the TPM to decrypts {K, AIK}pk(EK), using command TPM_ActivateIdentity.
9. TPM checks that AIK in {K, AIK}pk(EK) is really the one of its AIKs.
10. TPM releases the key K for user.

The purpose for checking the certification of the AIK is allow the entities to ensure that the platform configuration is signed by a valid TPM and this valid TPM can be

trusted. But this attestation has a main drawback that a privacy CA has to always be available and is involved in every transaction. This would be a privacy concern if there has no CAs available or the privacy CA and the person who is verifying the attestation collude. Therefore, a solution called Direct Anonymous Attestation (DAA) has been included in TPM 1.2. The DAA protocol defines three entities: the DAA issuer, the TPM and the verifier. The DAA protocol consists two main sections. One is called "join", a TPM join a TPMs group and get a DAA credential. The second step is called "Sign", the TPM sign a message with an existing DAA credential and sends it to the verifier. Then based on the zero-knowledge the verifier can verify the credential without revealing the platform's privacy. The more details about zero-knowledge will introduce in chapter 6. Nonetheless, this DAA protocol is very complicated and requires a large amount of platform recourse, so we deemed that the DAA protocol is not appropriate for our protocol and we still use privacy CA for attestation.

3.4.3 Secure storage

In the section of key management it was mentioned that SRK are the root of key chains. It is generated by TPM when user taking ownership and associate with a shared secret called authorisation data (e.g, password). Typically, this authdata is 160-bit and chosen by user processes. The TPM stores authdata with the relevant keys or sensitive information and at any time when use requests such sensitive data, he needs use authdata to attest his access permission. After the SRK generated, the user processes can create new key with an existing parent key. The private portion and new authdata of the new key are encrypted with parent key and combine with public portion to a blob, and then return this blob to user processes. To use this new key, user processes need to load it and its parent key into TPM. The key hierarchy as illustrated in figure 7, which has been taken from TCG overview 1.4.

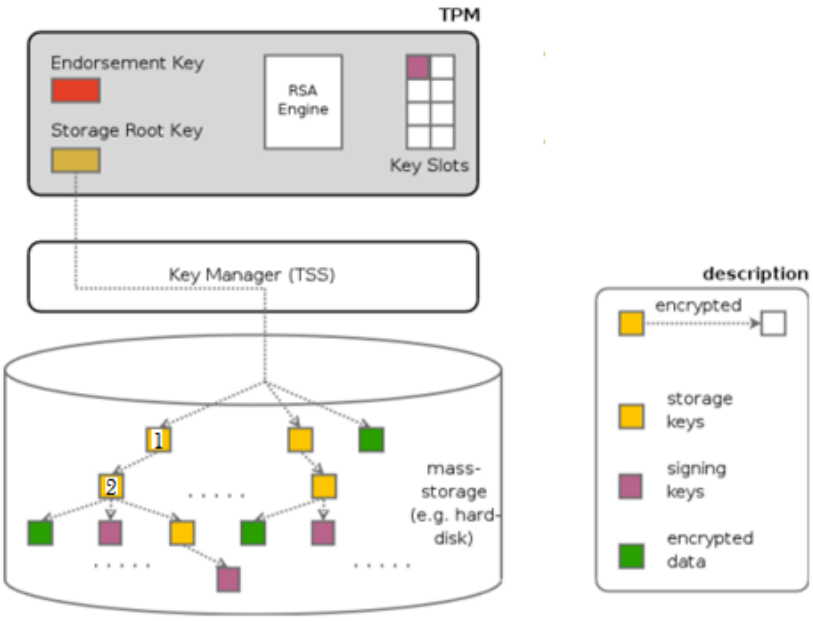

Figure 7– Key hierarchy

Considering this scenario, if user processes request key2, it must load key1 in TPM at first and using the private SRK to decrypt key1 and then put key1 in the key slots. After that, user processes load key2 into the TPM and decrypt it with key1's private key. Thus, the key2 is available for using.

TCG defines two ways of storing message, binding and sealing.

- Binding

Binding is a standard encryption with a public key. Since the binding data only need the public key, this operation can be executed either by TPM or user processes. It is possible to create migratable binding keys and transfer the private keys between multiple TPM.

- Sealing

Sealing data is more complex than binding data. Firstly, only non-migratable keys can seal data. In contrast, binding keys can be either migratable or non-migratable. Secondly, the encryption only can be done by the TPM. And thirdly, during the encryption time, the current PCRs value is encrypted as attribute of the data, when unsealing data, TPM needs compare PCRs values to ensure the data is integrity and security.

3.4.4 Authorisation sessions

The TPM provides two protocols for authorisation, OIAP and OSAP. Every time when user processes send commands to TPM, they need establish an authorisation session with TPM at first, and then authorisation protocols create an authorisation context with a handle return to user.

- Object Independent Authorisation Protocol (OIAP) is a long-term session; it can manipulate multiple objects in the same session. Therefore, OIAP can be used for loading keys operation.
- Object Specific Authorisation Protocol (OSAP) is a short-term session; it only can manipulate one object in one session. Typically, OSAP is used for creating a new key.

It should be noticed that, when user taking ownership (TPM_TakeOwnership), the OIAP should be used for authorisation, because in this case, all objects have not been created yet.

3.5 Compare with smart card

From what has been stated above we can see that TPM and smart card have many in commons, like they are both low cost, both are small devices used to provide the basis secure for computing environment. But they also have respectively features like:

- Ownership: the main different between smart card and TPM is ownership. Normally, the smart card owned by issuer before consumer receives the device. But the ownership of TPM is taken when first time in use.
- Smart card is more light: smart card can be installed into PDA, smart phone or other handheld devices, but TPM only can be installed into PC（for the present）.
- Smart card is portable: the credentials on the smart card are portable. For instance, when users need sign their e-mail or contract, they could do it in any places with smart card devices.
- Storage space: Due to the limitation on the storage capacity of a smart card, only small amounts of data or key pairs can be stored in it. However, the TPM can provide a large amount of persistent storage for user. Although the TPM itself only can store small amounts of data, the keys can be used to extend the storage in a virtually unlimited way. All files and data are encrypted by storage keys can be stored outside of the TPM.
- Attestation: The TPM can be used to attest the platform running status and the validity of the platform.

4 Protocol Design

This section presents the details of the protocol design. We have chosen vendor and consumer as example. The scenario is a consumer purchase a movie from a vendor's website. The movie was encrypted by vendor and after the consumer has paid the movie, the key for decrypting the movie will be release to consumer. We assumed that two parties have established a TLS connection before the transaction, and all messages will be encrypted by a session key, which is agreed by both parities during the handshake.

4.1 First version design

- Design

-Authentication

Due to the features of the TPM, all entities can authenticate each other. Hence, before the electronic transaction we decided that both entities send their platform measurement to each other for checking the validity of the TPM and the platform running status.

As mentioned in the chapter 3, the configuration is signed by TPM with an identity key AIK. The notations are used in figure 8 and figure 9 are shown in the following table.

Table 2

V, C	Short name for vendor and consumer
Nv,Nc	Nv is freshly generated by vendor Nc is freshly generated by consumer
Sigv SigC	Signature of vendor Signature of consumer
Sig_{TC} Sig_{TV}	Signature of consumer's TPM Signature of vendor's TPM
order	A text file that contains order's information (such as movie's name, price)
cardinfo	Credit card information
AIK	Attestation Identity Key
Cert(AIK)	A certification of AIK
sk(C,V)	A session key

KCT,KVT	The symmetric keys are freshly generated by entities
ACK	Acknowledgement; a evidence of the vendor that shows he already received the encrypted cardinfo

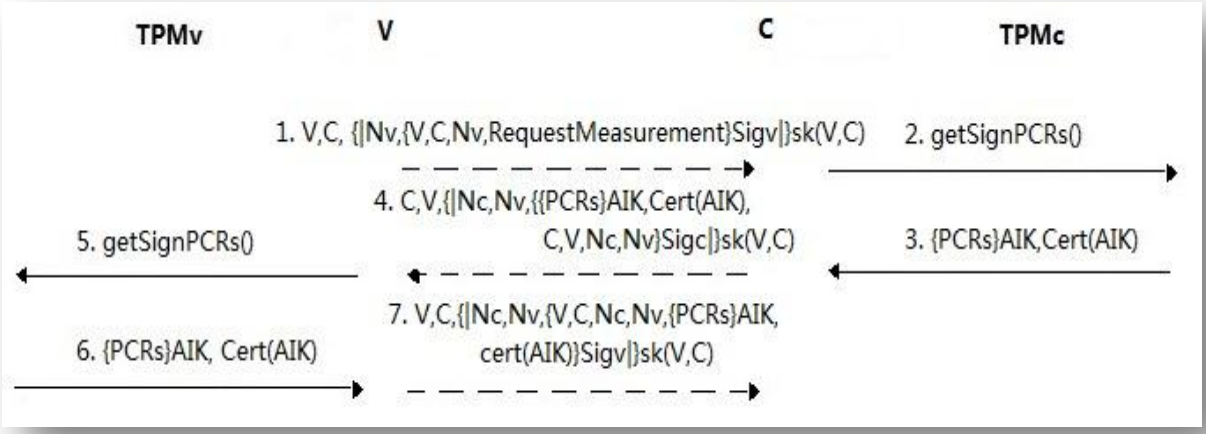

Figure 8

Figure 8 shows vendor sends a signed measurement request to consumer. The request contains a fresh generated Nonce Nv and two parties name V and C. Nonce is a random number or nonsense string that fresh generated by vendor. By including the nonce Nv in the reply message, the vendor must be sure that signature from consumer's TPM is fresh. And in order to protect the message has not been modified by attacker, two parities' name should be also included in the signature. Afterward, consumer sends its configuration with Nv and Nc to vendor. Likewise, vendor sends it configuration to consumer. Thus, both parties can check the status of platform each other and to decide whether continue the transaction or not.

-Main protocol

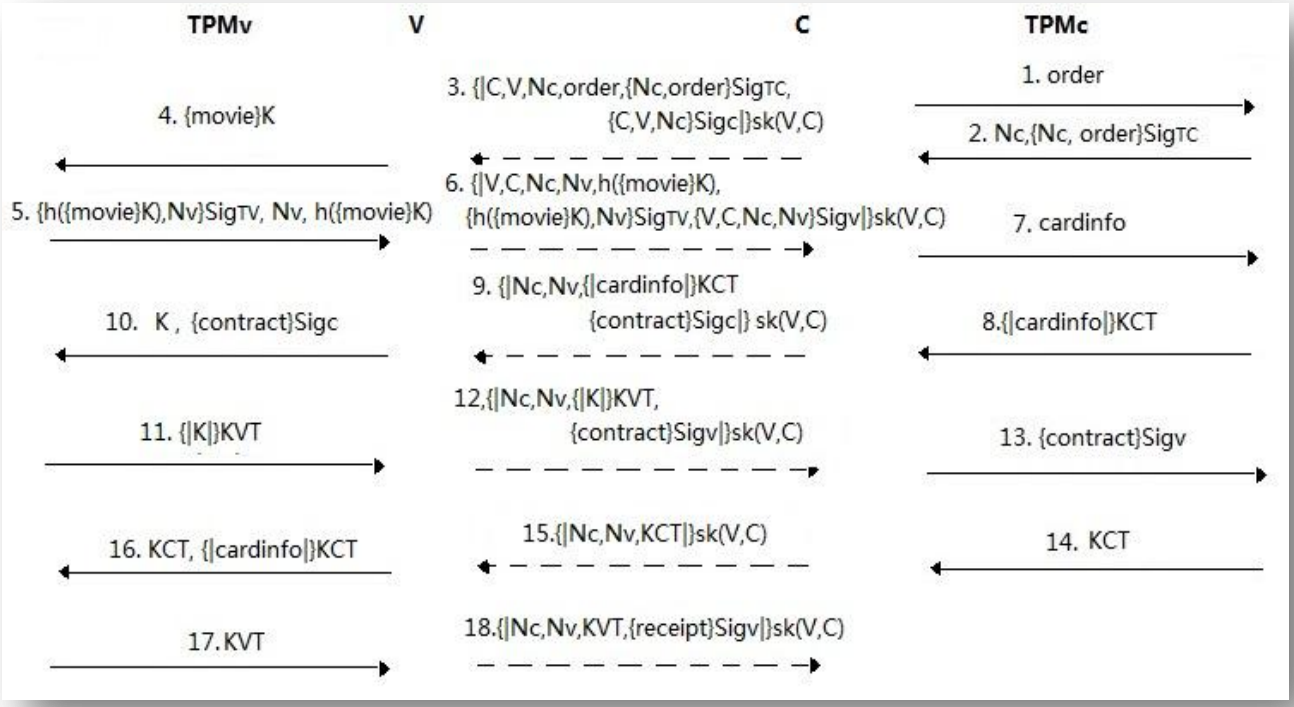

Figure 9

The figure shown above is the first version of the protocol. TPMc is the TPM that install on consumer's PC and TPMv is installed at the vendor's PC. All communication channels between TPM and user processes are reliable and presented as solid lines. All communication channels between two parities are unreliable, presented as dashed line.

1. We assume that all movies are encrypted by vendor and can be downloaded from the vendor's website. So in the first step, we assume consumer already have the movie, he just needs the key to decrypt it. Consumer (user processes) sends the "order" to TPMc. This order contains movie's information, such as movie's name, length, director, language and date.
2. TPMc generates a nonce Nc and signs {Nc,order} with a signing key Sig_{TC}
3. Consumer checks signature of the TPMc and signs "C,V,Nc" to produce message 3. Consumer then encrypts the message 3 with the symmetric key (the key that both vendor and consumer are agreed) and sends it to vendor.
4. Upon receipt of message 3, vendor checks the data that in signature is against the plaintext or not. If data are not match, then vendor ignore the message. Otherwise, vendor sends the encrypted movie (the one that consumer ordered) to TPMv.
5. TPMv generates the nonce Nv and a hash of the encrypted movie. And then TPMv

signs "Nv,h({|movie|}K)" with a newly generated signing key Sig_{TV}. TPMv sends message 5 to vendor.

6. Upon receipt of message 5, vendor signs "Nv,Nc,C,V" and produce message 6 then send it to consumer.
7. Consumer checks the hash value h ({movie}K), and compare the data that in the signature with the data in the plaintext. If that evaluates to be true, then consumer sends the credit card information to TPMs for encryption. Otherwise, consumer ignores the message.
8. TPMv encrypts the {cardinfo} with a symmetric key KCT and sends the message 8 to consumer.
9. Upon receipt of message 8, consumer produces a digital contract and signs it. This contract contains h({|movie|}K) value, movie's information, consumer's order and the encrypted credit card information{cardinfo}KCT. And then consumer signs {cardinfo}KCT and produce message 9 then sends it to vendor.
10. Upon receipt of message 9, vendor checks the validity of the contract. If it is valid, then vendor sends the symmetric key K to TPMv. Key K is the one that used for encrypting the movie.
11. The TPMv seals the contract with PCRs value in order to protect the integrity and security of the contract. Then TPMv encrypts the Km with a newly generated key KVT (a symmetric key) and sends the message 11 to vendor.
12. Upon receipt of message 11, vendor adds {|K|}KVT in the contract and signs the contract. Then vendor produces the message 12 and sends it to consumer.
13. Upon receipt of message 12, consumer checks the contract. If the signature is valid, and all data in contract are correct and valid, then consumer forwards it to TPMc. Otherwise, consumer terminates the transaction and sends abort request to the TPMc.
14. After receipt of message 13, TPMc seals the contract with PCRs value and releases the key KCT for consumer.
15. Consumer produce message {|Nv,Nc, KCT|} and sends it to vendor.
16. Upon receipt of message 15, vendor forward the key KCT and {|cardinfo|}KCT to TPMv for decryption.
17. TPMv try decrypts the {|cardinfo|}, if the key is valid then TPMv releases the key KVT for vendor. Otherwise TPMv return an error message to vendor.
18. Upon receipt message 17, if TPMv releases the key KVT then vendor forward this key with a signed receipt to consumer. Otherwise, vendor terminates the transaction and sends resolve request to the TPMv.

- Discussion

This design only satisfied authentication and online TTP. But the most important goals of fairness and non-repudiation have not been achieved. From the main protocol we can identify several weaknesses. Firstly, there has no deadline for limiting time, so the two parties may wait a message forever. Secondly, although two parities had a

signed contract they still cannot take it as evidence. Because, even the contract contains the movie information, movie's hash value and both parities' secrets, it has not contained the signature of both parties. So they cannot prove for the judge that this contract is a valid contract and they both reached the agreement. Thirdly, if the vendor finds the credit card information was invalid after he sent the key KVT to consumer, he cannot ask judge to resolve the transaction, because he does not have the valid contract. The signed contract by consumer only contains consumer's secret, cannot be used as evidence. Finally, if vendor received the incorrect contract, he could terminate the transaction and sends an abort request to the TPMv. But process did not present clearly how to abort the protocol, and who can resolve the disputes. Both TPMs cannot act as a judge, because both vendor and consumer have no right to access each other's TPM, so if the contract was already aborted by one party's TPM then how can another party to get an abort token? Likewise, if either a TPM resolves the contract, how can we ensure that both vendor and consumer can get a same replacement contract? This is a classic issue in digital transactions. There has no guarantee for message that can be integrally and securely delivered to other party. Therefore, we decided to involve a trusted third party for solving such problem. The details will describe in the following section.

4.2 Second version design

- Design

In this section, we will not present the authentication part, since it is already satisfied. In order to improve the timing problem, we add two sub-protocols called abort sub-protocol and resolve-protocol, which are designed based on the ASW protocol. We also introduce a trusted third party into the protocol. The following diagram shows the details of the process.

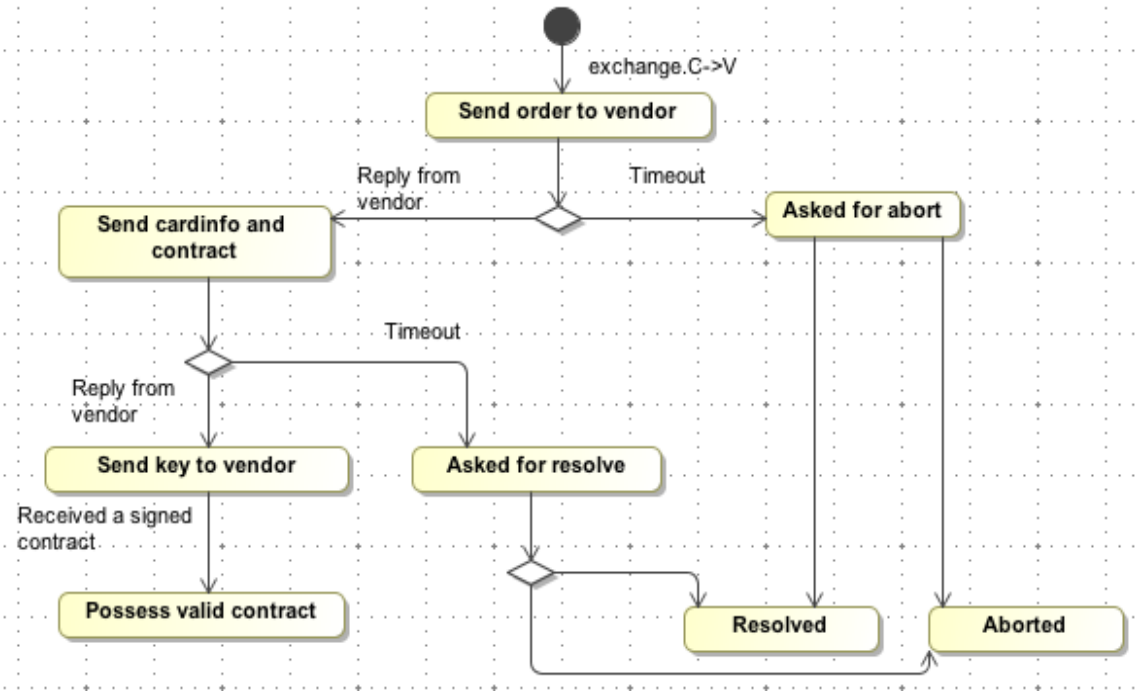

Figure 10 state transition of the process.

The figure 10 is viewed in the perspective of the consumer. When exchange protocol started, consumer will send an order to vendor. If consumer receives the reply from vendor in a reasonable amount of time, then transits to the next state. Otherwise, consumer asks TTP for abort. And if consumer received vendor's secret after he sent his own, then possess a valid contract. Otherwise, consumer can ask TTP to either replace the contract or aborted it. The process contains three final states: Possess valid contract, Resolved and Aborted. Despite the transaction successful or not, the participant entities will eventually reach one of the final states. This design can also be presented in formal specific language.

Table 3 Notations used in the protocol

Nc,Nv	The nonce are freshly generated by agent C or V
order	A text file that contains order's information (such as movie's name)
Sigx(Y)	The message Y signed by agent X
contract	Contracture text
Cardinfo	Credit card information (such as card number, authorized name)
Kc,Kv	Kc is a key used for encrypting {cardinfo}, Kv is a key used for encrypting {K}
K	K is a key used for encrypting movie
h()	Hash function
sk(C,V)	A session key for transaction (the detail is given in section 4.1)

Exchange protocol:

meg1=C->V: {C,V,Nc,{order, Nc}Sigc}sk(C,V)
meg2=V->C: {V,C,Nc,Nv,{h({movie}K), h(movieinfo),Nv }Sigv}sk(C,V)
meg3=C->V: {C,V,{Nc,Nv,contract,{cardinfo}KCT}Sigc}sk(C,V)
meg4=V->C: {V,C,{meg3,{K}KVT}Sigv}sk(C,V)
meg5=C->V: {KCT}sk(C,V)
meg6=V->C: {KVT}sk(C,V)

Abort Sub-protocol:

C->V: ma1={aborted,meg3}Sigc
T->C: if resolved(meg3)
then {meg3,meg4}SigT;
else {aborted, meg3} SigT;
aborted(meg3)=true;

Resolve Sub-protocol:
C->T: mr1=meg3,meg4
T->C: if aborted(meg3)
then{aborted,meg3} SigT;
else {meg3,meg4} SigT;
resolved(meg3)=true;

In the **Exchange Protocol**, first consumer sends the order to vendor and vendor sends back a hash value of encrypted movie. Thus, both consumer and vendor reached a consensus for the transaction. Afterward, Consumer sends the encrypted credit card information and a contract need to sign. Vendor signs the contract and sends it to consumer with his secret. After that, both of them have a signed contract and each other's secret. If they both agree the contract, then they can exchange their respective secret. As mentioned in section 4.3.1, the protocol should have deadline to ensure the entities will not wait a message forever. Therefore, if in the meg3 consumer did not receive the reply from vendor in the finite amount of time, consumer can ask TTP to abort the protocol, which is shown in the **Abort Sub-protocol**. The abort sub-protocol ensures the TTP has never issued a replacement contract and will never to so in later. Consumer sends a signed abort request to TTP, which indicating that he wants to abort meg3. TTP checks its database, if the contract already asserted then he sends a signed replacement contract {meg3, meg4}SignT. Otherwise, TTP signs consumer's abort request and register the contract as aborted in his database. For the **Resolve**

Sub-protocol, if one of the parties did not receive his respected secret in a finite amount of time, they can invoke TTP to replace the contract. If TTP already aborted contract, then he replies a signed aborted token. Otherwise, he issues a replacement contract (a valid contract) and registers the contract as resolved in his database.

- Verification

  In order to verify the protocol, we have chosen the OFMC for formally analysis. The tools can report whether the protocol can be attacked or not. The AnB code will present in the appendix A

```
Zhang-QiuzimatoMacBook-Pro:~ guiyanjingling$ /Users/guiyanjingling/Downloads/ofm
c-2012c-release/ofmc-mac ~/Desktop/fair.anb --classic
% Open-Source Fixedpoint Model-Checker version 2012c
Verified for 1 sessions
Verified for 2 sessions
Verified for 3 sessions
Verified for 4 sessions
Verified for 5 sessions
```

Figure 11

As shown in the figure 11 we verified for 5 sessions and did not find attack. During the transaction, we used symmetric key to protect the exchange message and in every party's signature, we add the nonce and parties' name to indicate that the message is fresh and belongs to the particular entities. Thus, the intruder cannot either modify the message or impersonate one of the parties to control the communication channel (Man-In-The-Middle).

- Discussion

In this version, we basically satisfied the goals. For instance, the timeliness, all participant entities can terminate the transaction without loss fairness. The TTP will help entities either to abort the protocol or replace it. And also we add two sub protocols to resolve the disputes in a better approach. Another improvement is that we modified the old contract (first design) to more specific. It now looks like:

Contract={C,V, order ,h(movie),movieinfo,Nv,Nc,{secret}C,{secret}V}

After two parties exchange their respective encrypted secret and the signed contract. They can check the contract, if all data are correct. Then they can continue the transaction. That is to say, the both parties reach an agreement on the contract.

Unfortunately, this version still has problem. Even two parties exchange the signed contract, we still cannot prove that entities have accepted the contract. If vendor stopped the protocol after he received the secret of consumer, the consumer cannot do anything for such unfairness, because he dose not have either a valid contract or a valid evidence. Hence the entities should claim themselves that they both agreed on the contract and send the evidences to each other. So, due to these drawbacks of the

protocol, we improved it to the final version.

4.3 Final version design

According to many discussions we found that the best solution for the protocol is combine the ASW protocol with TPM chips. The ideal of this design is use ASW protocol for messages exchanging and use TPM to protect the secrets.

- **ASW+TPM protocol**

This section presents the final version of protocol that combines ASW protocol and TPM. We use TPM to encrypt the secret with other TPM's public key, for our case we have chosen the public binding key for encryption and we assume that both parties has already exchanged their respectively public binding key. The encrypted message only can be decrypted by the TPM who has the corresponding binding key to get secret. The detail shows in the Figure 12

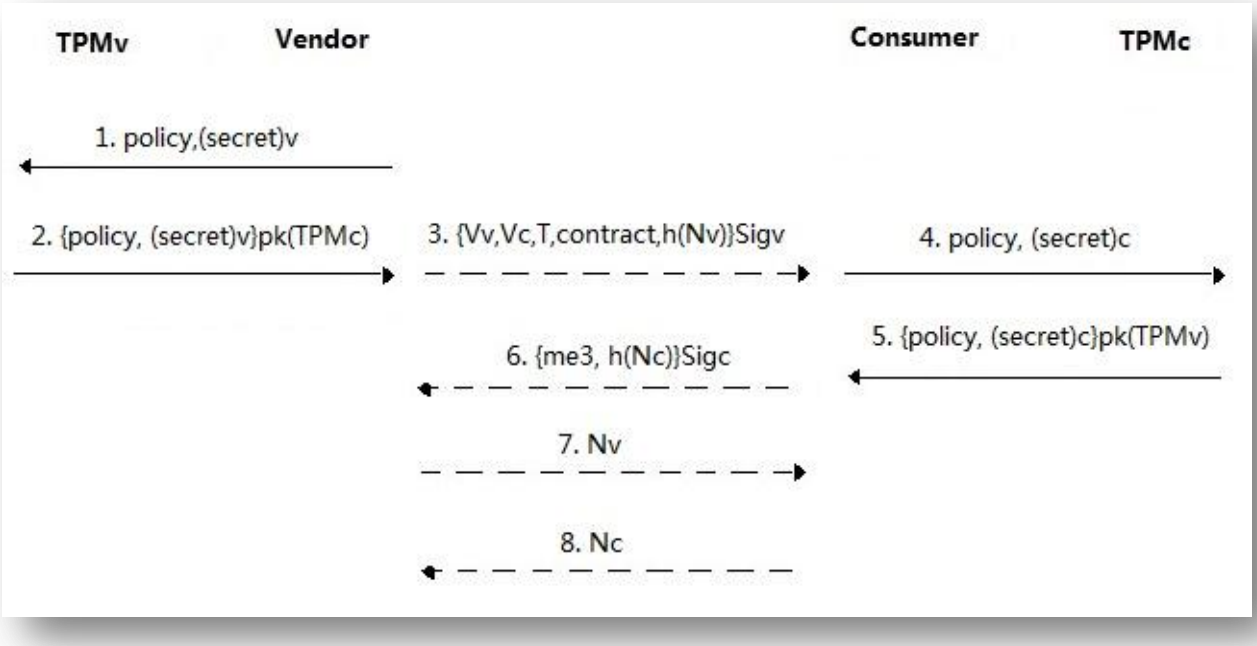

Figure 12

In order to better understand the protocol, again we decided to illustrate by the vendor/consumer example. The vendor acts as an originator and the consumer acts as a responder.

1. Vendor sends its secret (the key for decrypting movie) and a policy to TPMv(the one installed in the vendor' PC).
2. TPMv encrypts {policy,(secret)v} by using TPMc's public key.
3. Upon receipt of message 2, vendor generates the contract and public commitment h(Nv). The **contract** contains:

h(movieinfo), {policy,{policy,(secret)v}pk(TPMc)}pk(c)

The **movie information** contains: *movie name, director, actors, release date, runtime and language.*

Vendor signs contract, public commitment h(Nv) and the verifications of public keys Vv,Vc

4. Upon receipt of message 3, consumer checks the validity of signature. If the evaluate to be valid Then consumer compares the hash value of movie's information h(movieinfo). If two values are equal, consumer decrypt the component {policy,{policy,secret}pk(TPMc)}pk(c) to get policy. At the last, consumer sends policy and its secret(credit card information) to this TPMc(install in the consumer's PC).
5. TPMc encrypts {policy, (secret)c} with TPMv's public key and sends it back.
6. Upon receipt of message 5. Consumer generates its public commitment h(Nc) and produce message 6.
7. 8. Upon receipt of message 6. Vendor and consumer exchange their respectively secret commitment Nv and Nc.

- **Policy**

As mentioned above, the contract contains a policy that is used to check the secret commitment is corresponds to the public commitment from the first protocol stage. The TPM is the one to run the policy and decides whether release the secret or not.

```
if h(secret commitment) = public commitment
   then policy := true;
   else policy := false; quit;
```

As shown above, if entities received secret commitment, he sends both secret and public commitments to TPM. TPM executes the policy to compare two commitments. If they are equal, then TPM return true to user processes. Otherwise return false and quit the execution. TPM does not need save all commitments, only when user processes sends request to TPM. Then TPM executes the relevant policy.

Since TPM does not need to save policy or relevant data. Policy execution has been divided into two steps:
Checking Policy:

User process → TPM : {policy, secret}pk(TPM)
TPM → User process : if DecryptMessage({policy,secret}pk(TPM))
then return true;
else return false; quit;

The first step is used to obtain policy and secret. Looking once again at figure 12, the

contract contains an encrypted message {policy,secret}pk(TPM). When user received it, he sends it to TPM for decrypting. Once TPM decrypts the message succeeded, it returns true to user processes. If TPM decrypts the message failed, it will return false and quits execution.

Execution Policy:

User process → TPM : (secret commitment, public commitment)
TPM → User process : if policy = true
then TPM release secret;
else TPM return false; quit;

The second step is policy execution. User processes sends secret and public commitments to TPM. TPM executes policy to check the conditions. If conditions are satisfied, then TPM release secret and quits the execution. Otherwise, TPM returns false and quits execution. When TPM quit the execution, it will delete all relevant data, such as policy and secret. Whenever user processes want to execute the policy, it needs request again.

- **Verification**

We used OFMC to check the protocol, the result shows in the figure 13. The AnB code will present in Appendix B.

```
Zhang-QiuzimatoMacBook-Pro:~ guiyanjingling$ /Users/guiyanjingling/Downloads/ofmc-20
12c-release/ofmc-mac ~/Desktop/TPM.anb --classic
% Open-Source Fixedpoint Model-Checker version 2012c
Verified for 1 sessions
Verified for 2 sessions
Verified for 3 sessions
Verified for 4 sessions
Verified for 5 sessions
Verified for 6 sessions
```

Figure 13

As shown in the above, we verified for 6 sessions and did not find any attack.

- **Discussion**

This protocol has several advantages:

1. The ASW protocol is easy to understand and easy to use. The exchange sub-protocol only has four steps, but it can strongly ensure the fairness for both parties.
2. Light involvement of TPM. The TPM only needs encrypt the secret and decrypt other's secret.
3. The TPM do not need save all data.
4. Checking the signature on {policy, secret} can strongly ensure the integrity of the

secret and policy.

This version has finally achieved the protocol goals. The ASW protocol is used to sign a contract, the TPM is used to verify the contract and when disputes arises the protocol involving the trusted third party (TTP) to resolve the unfairness. To summarise, the final protocol can be deemed as three parts. The first part is authentication: two entities exchange their signed PCRs values to in order authenticate each other. The authentication goal reached. The second part is exchange protocol: two entities exchange their signed contract and secret commitment. And use TPM to verify the validity of the contract, if the evaluates to be true, then TPM will release the secret, otherwise, the TPM reject the request. The third part is used to resolve the disputes. If entities do not receive his expected item, he can invoke TTP to whether replace the contract or abort it. Thus, achieved the fairness and timeliness goals.

5. Experimentation

This chapter presents the details of program experimentation. The section 5.1 introduces the software stack of TCG and the process of taking ownership and creating storage root key SRK. The section 5.2 describes how to initialize the TPM that include taking ownership and creating SRK. The section 5.3 presents the result of the implementation. Due to the limitation of the TPM, unfortunately, we cannot implement the policy with the TPM at present. The more details will be documented in the chapter 6 and this chapter concerns achievements of implementation.

5.1 TCG Software Stack

The TCG software stack (TSS) is the set of software components intended to provide application programming interface (API) which allow applications to access TPM functions. The TSS was defined by TCG with four perspectives[15].

1. Provide the entry point for applications to TPM functions.
2. Provide both local and remote machines access to the TPM.
3. Manage TPM resources.
4. Conversion of data blobs from applications to TPM.

The TSS defines three software components: TSS Core Services (TCS), TSS Service Provider (TSP) and TPM Device Driver Library (TDDL). The TSS is divided into two main modules: user mode and kernel mode [14]. The user mode functions as executing user applications and services. The operating system provides protect area called processes for different applications and services, which are needed to protect. The software component TSP, TCS and TDDL are part of user mode. **The TSS Service Provider** provides interface for c program language. It implements as a linked library and can be called directly by user applications. The TSP collects all data ready for the command, and then passes it to TCS. The TCP has cryptographic capability. It might encrypt the sensitive information. Any data has reached TCS is either encrypted by TSP or is public. **The TSS Core Service** provide interface for local platform or RPC service from remote platform. It implement as system process in user mode. It has capabilities such configuration management, key management and resource management. Since the TCS might run multiple TSP in parallel, therefore, TCS also manages access to TPM. And in additional, the TCS also reply to commands when TPM has not responded. The **TPM Device Driver Library (TDDL)** provides the interface for TPM Device Driver. It enable the different applications of TSS can communicate with TPM. Generally, the TDDL is used for sending/receiving data blobs, query the device driver's properties and turn on/off the driver device.

Below the user mode is kernel mode. Based on the TCG TPM specification, the kernel mode is the place for core component of operating system and TPM device derive reside. TCS sends the request to TPM via TPM device driver interface. All modification and update require administration authorization.

5.2 Initialize TPM

5.2.1 Initialize TPM

At present it has many software can manage the TPM. The most efficient is TrouSerS and Trusted Computing for the Java Platform. Based on our equipment, we decided to use TrouSerS. The TrouSerS is an open source implementation of the API [16].

To use the TrouSerS, the system needs install it first. Once the TrouSerS install successful, then TPM can be accessed. We used the following command to check the TPM.

```
s104664@pcsamo:~$ sudo tpm_version
  TPM 1.2 Version Info:
  Chip Version:         1.2.8.28
  Spec Level:           2
  Errata Revision:      3
  TPM Vendor ID:        STM
  TPM Version:          01010000
  Manufacturer Info:    53544d20
```

Figure 14 Check TPM is accessible

If the TPM is accessible, the system shows the information of TPM.

5.2.2 Taking Ownership

The TPM now is ready for taking ownership. As mentioned in chapter 3, the TPM must be owned before it can be used. So after the TPM has been initialized. A command called tpm_takeownership should be invoked. This command has several options: set the secret of all zeros as owner's secret, or set the secret of all zeros as the SRK secret. In our case, we decided to take ownership without options. The details are shown in following:

```
s104664@pcsamo:~$ sudo tpm_takeownership
Enter owner password:
Confirm password:
Enter SRK password:
Confirm password:
```

Figure 15 Taking ownership

We set both the owner password and SRK password. After the secrets have been created, the TPM is ready for creating new key or other functions.

5.2.3 Endorsement Key

Every TPM has its unique endorsement key, after took the ownership, the user can check the public endorsement key by invoke the command called tpm_getpubek. The key of our TPM is given in figure 16.

```
s104664@pcsamo:~$ sudo tpm_getpubek
Tspi_TPM_GetPubEndorsementKey failed: 0x00000008 - layer=tpm, code=0008 (8), The
 TPM target command has been disabled
Enter owner password:
Public Endorsement Key:
  Version:   01010000
  Usage:     0x0002 (Unknown)
  Flags:     0x00000000 (!VOLATILE, !MIGRATABLE, !REDIRECTION)
  AuthUsage: 0x00 (Never)
  Algorithm:          0x00000020 (Unknown)
  Encryption Scheme: 0x00000012 (Unknown)
  Signature Scheme:  0x00000010 (Unknown)
  Public Key:
        e2d9c589 f1aeb33a eb4e52e2 0fec5ca1 4f53009f bd5fb739 61f04410 64f8e237
        136d23b5 b529877c a202f788 8a18a27d 21193ee1 6c7e0364 ce667295 3296aae2
        35016311 7cfde65a 13469a2b 0bc2f669 42607799 8659b356 57419aa2 ebd514c6
        876e74d2 08e3c796 38024dfa f342cdde 81f71704 138434f6 8f3a4f95 ce1509da
        40df2e40 2be37c0e 0589d406 c51d39d6 bf4a6771 7b06624a 073c9d55 b6e47b4e
        84b7da4b 9832626e ff746512 05bbc5c4 e6b5563e eca2cdc3 476fa976 5f95e3c5
        114b84db ed773ccb dfc5070a 2a3ffe3e cf2f6850 89355935 828d9985 3d11d744
        86c9c613 f61bf4b9 5c8b2856 c781421d b30d8dde a74f549f 4b5e3bdd 007cf9f3
```

Figure 16 Get endorsement public key

The endorsement key is a 2048 bit RAS public and private key pair [16]. The public key normally is used to encrypt the sensitive data or attestation, such as encrypt AIK.

5.3 Implementation

In this section, we only present result of the implementation and describe some parts of the program in order to illustrate how TPM works. All program code is based on

the TPM sample code to modified [17]. The complete code will be presented in Appendix C and Appendix D.

5.3.1 Pre-Test

Typically, the program code contains three main structures. The first part is preamble part. It is the beginning code of every program that used to establishes a context for connecting the program to the TPM. It also get handle of the TPM and SRK. The second part is the operation. All operations program code such as creating a new key or binding data will be documented in this section. The third part is postlude. It is used to clean up the context. This is the end of every program. In addition, before the main code, the program should define some basic interpretation of error message. The TrouSerS defines DBG() that used for representing the error message.

The pre-test c file is used for getting a handle of the TPM and the SRK. The program code is shown in Appendix C and the result is given in the following.

```
s104664@pcsamo:~$ ./tpmTest
----------------Program start----------------
Line32, main)Create Context returned 0x00000000. Success.
Line34, main)Context Connect returned 0x00000000. Success.
Line37, main)Get TPM Handle returned 0x00000000. Success.
Line40, main)Got the SRK handle returned 0x00000000. Success.
Line44, main)Got the SRK policy returned 0x00000000. Success.
Line47, main)Set the SRK secret in its policy returned 0x00000000. Success.
```

Figure 17

The reason why implement the pre-test before creating a new key, is that we have to ensure the TPM can be successfully invoked and the SRK can be handled. Thus, all following operations can be proceeded upon on this test. If the SRK cannot be handled, for instance, set a wrong password in the program, then the system may give errors that show the SRK cannot be found in the storage. The result shows in the Appendix D.

5.3.2 Create A New Key

The TCG defines several types of keys, for our case we decided to use the binding key. Because the binding key as one of the encrypting keys that stores small amount of data and only need the public binding key for binding data, the TPM do not necessary to involve too much in this operation. Another encrypting method is sealing data, which only use the non-migratable storage keys. During the encryption, the current PCRs values will be encrypted as the attribute of the data. The data only can be decrypted in the same TPM. Therefore, the sealing data is not suitable for our protocol.

The command TPM_CreateWrapKey used to create a new key, but it does not store it.

The use the key, it must be loaded by command TPM_LoadKey2. Looking once again at section 4.4 for each operation, the user process has to establish an authorisation session with TPM. For creating a new key, we should call TPM_OSAP.

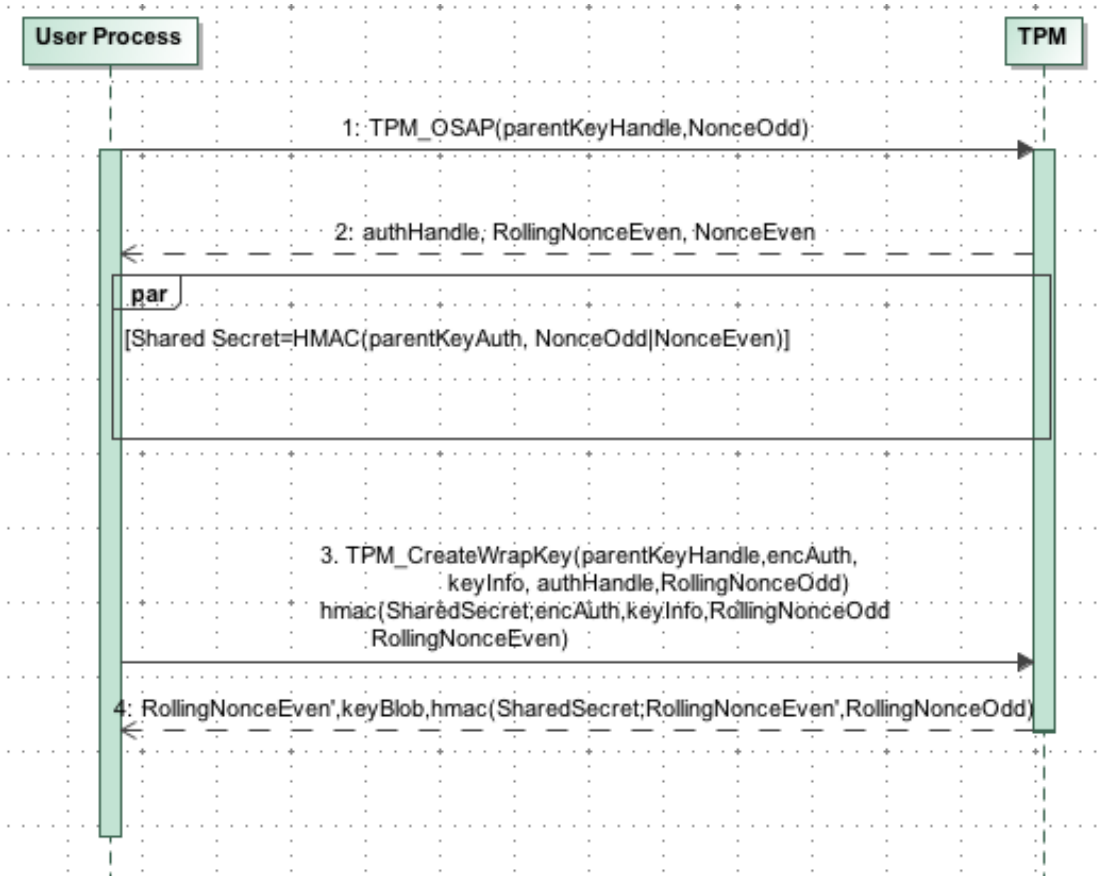

Figure 18

The user process sends a command TPM_OSAP () to the TPM. The parameters of OSAP are object's name (parentKeyHandle) and an odd nonce. The TPM response with an authorisation handle, the current even nonce for rolling secret and even nonce. Then, the TPM and user process each computers a shared secret. The calculation of the shared secret is to hash the odd nonce and even nonce by applying the HMAC key, parentKeyAuth. After that, the user process sends the TPM_CreateWrapKey command with arguments that include the handle for the parent key of the new key, the encrypted authdata (encAuthe) of the new key and an odd rolling nonce that freshly generated by user. Lastly, the TPM returns a keyblob contains the public portion of the new key and an encrypted package. The encrypted package is encrypted with the parent key. It contains the private portion of the new key and the authdata of the new key.

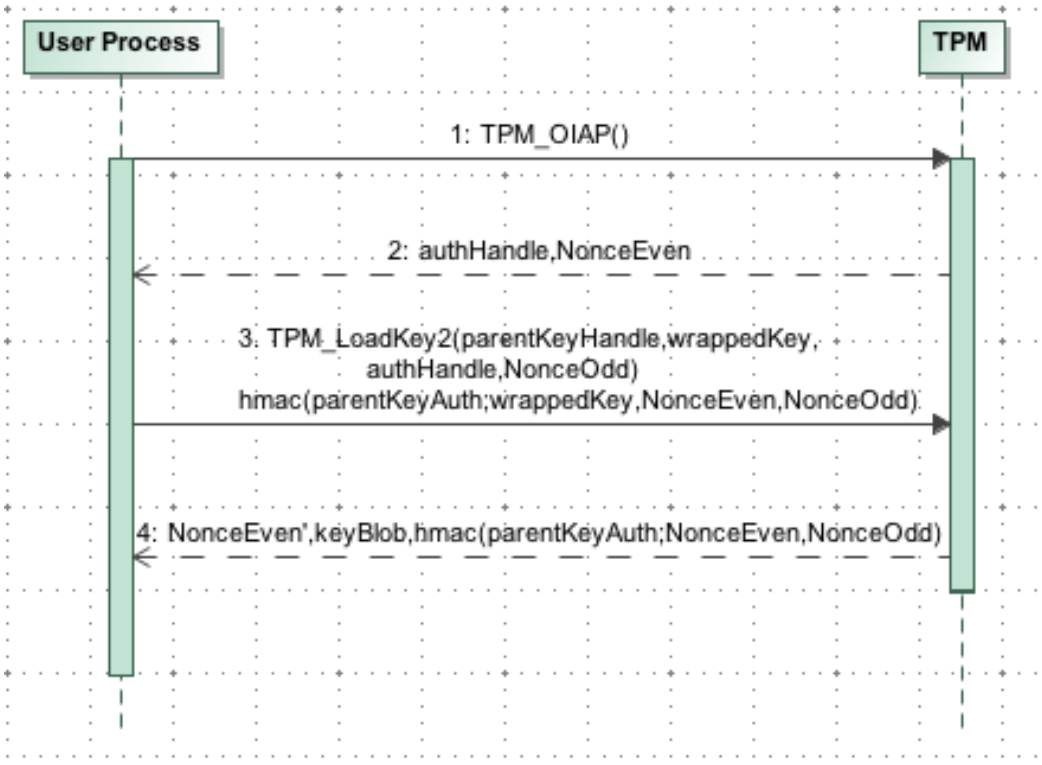

Figure 19

For loading key, the user process sends TPM_LoadKey2() to TPM. The TPM returns a handle and an even nonce. To load a key, it also needs to load parent key for description, therefore the parent key and the authorisation HMAC based on the parent key authdata should also be loaded.

```
s104664@pcsamo:~$ gcc -o creatKey2 tpmCreatBkey.c -ltspi
s104664@pcsamo:~$ ./creatKey2
----------------Program start----------------
Line38, main)Create Context returned 0x00000000. Success.
Line40, main)Context Connect returned 0x00000000. Success.
Line43, main)Get TPM Handle returned 0x00000000. Success.
Line46, main)Got the SRK handle returned 0x00000000. Success.
Line49, main)Got the SRK policy returned 0x00000000. Success.
Line54, main)Set the SRK secret in its policy returned 0x00000000. Success.
Line59, main)Create a backup policy object returned 0x00000000. Success.
Line61, main)Set backup policy object secret returned 0x00000000. Success.
Line65, main)Create the key object returned 0x00000000. Success.
Line68, main)Set the key's padding type returned 0x00000000. Success.
Line72, main)Assign the key's policy to the key returned 0x00000000. Success.
Creating the key could take a while
Line76, main)Asking TPM to create the key returned 0x00000000. Success.
Line79, main)Register the key for later retrieval returned 0x00000000. Success.
Registering key for later retrieval
Line84, main)Load key in TPM returned 0x00000000. Success.
Line87, main)Get public portion of key returned 0x00000000. Success.
Finished writing BackupESSBindKey.pub
```

Figure 20

The figure 20 shows the implementation result of creating a new binding key. Every time of creating a new key, we should define a new UUID for the key to be created. The UUID is a form that used for key register. Every key has its unique UUID, if the UUID has already associated with other key, then system shows the UUID has already register and reject the request. The result is given in Appendix F. After the binding key has been created, we load it into the TPM and write the public key in the file BackupESSBindKey.pub. The result is shown in Appendix G.

6. Conclusion

6.1 Our contribution

In this study, we combined the non-repudiation protocols with TPMs to a new protocol. Based on many times modified, we finally achieved the goals. This protocol can provide both strong fairness and security for electronic transactions without heavy involve TPM. And for the implementation, we established a TLS connection between the server and client, which could be used for remote communication and may support further development, since TLS connection is not our main goals, so we did not present it in the implementation. We also successfully get handled of the TPM and SRK. We can use the SRK to create a new key and load the key into the TPM. Since we decided to use the binding key, so later on we can choose the data is either encrypted by user process or by TPM. But the unbinding data has to decrypt by TPM.We just load the key and read the encrypted data from appointed file into the TPM. Afterward, the TPM decrypts the data and return it to the user process. But due to the limitation of the time, the unbinding data part has not been finished.

6.2 Limitation

A number of important limitations need to be considered. Firstly, the current investigation was limited by TPM hardware component. According to the current situation, the TPM cannot realise such non-repudiation protocol. Because the TPM does not has the function for policy estimation and be a judge for user. The TPM does have an execution engine, but it is only used for special program code execution (a piece of code permanent resident in TPM). It is cannot be used for running other program. Therefore, to achieve the non-repudiation protocol, the first thing need to do is update the TPM chips. Due to the inexpensive feature of TPM, the manufacturer only need add some functions to TPM. For instance, add a function for comparing two conditions, whenever a user process request to check a policy, the TSS transfer the c language (or other program language) to the code that TPM can understand, and send both policy and secret to the TPM. Once the conditions are met, TPM can release the relevant secret and clear the slots. The TPM does not need capabilities for performing the program language or save policy or relevant data in the memory.

Secondly, all keys are encrypted by their parent keys, if the TPM needs key1 for an operation, it needs load both key1 and key1’s parent key into the key slots. It looks very safety, but once the SRK secret has been forgotten or the TPM occurred errors.

All these keys cannot be used again. According to the recent research, over 70% people had forgotten their passwords, and some people needs reset their password for every month. Hence, if some businessmen used the TPM keys for signing contract or sensitive data encryption, and he accidentally forgot the password of TPM, some serious problem could be occurred. Therefore, people need to be very cautions for using TPM.

Additionally, due to the TPM can only perform one operation at one time, and the number of resources is finite (like key slots, auth slots). So we conclude that the TPM's efficiency might be a little bit slow.

6.3 Further discussion

A number of possible future studies could be helpful at this concerning area.

- **Smart card**

  The first possibility is working with smart card. We already compared TPM and smart card in chapter 3. Both of them have respectively advantages and disadvantages. It may provide a stronger union by combining them together. In the [4], the authors briefly introduced two possibilities：

  Smart Memory Cards and TPMs can be used to store PIN-protected of random numbers. When user wanted to use some specific machines, he just need insert the smart card into the machine, and load a migrated blob into the system via secondary signing blob.

  Smart Signing Card and TPMs is a model that put TPM into the smart card. The smart card based on the TPM can be used to store migration blob and protect the migration keys. The nice thing of this design is that allow the cards to be portable between systems.

  To use the smart card based on TPMs, it allow the user to safety sign contract and encrypt data in any places.

- **Protect user's privacy**

  Another possibility is to use Zero Knowledge to protect user's privacy. Zero Knowledge allows one party A(prover) can prove to another party B(verifier) that a given statement of secret is true, without revealing any useful information about the secret. For instance, if some movies only sales for consumer who over 18 years old, then the consumers have to prove that he/she met the requirement. To use the zero knowledge proof, the consumer can send the identity card number (eg, CPR number) to TPM, and based on the calculation the TPM can sign a statement to prove the consumer is over 18 years old. Thus, the sensitive information has not been revealed.

Bibliography

[1] Internet RETAILER (network news)
http://www.internetretailer.com/2013/02/05/global-e-commerce-tops-1-trillion-2012
[2] CNN (network news)
http://finance.fortune.cnn.com/2013/06/21/ecommerce-global-internet-economics/
[3] Design and verification of non-repudiation protocol based on receiver-side smart card by J.Liu and L.Vigneron
[4] IBM. A Practical Guide To Trusted Computing *by David Challener, Kent Yoder, Ryan Catherman, David Safford, Leendert Van Doorn in 2008*
5] TCG Specification Architecture Overview 1.4:
http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf
[6] TPM Introduction:
http://courses.cs.vt.edu/~cs5204/fall10-kafura-BB/Papers/TPM/Intro-TPM-2.pdf
[7]Trusted Computing
http://www.cs.bham.ac.uk/~mdr/teaching/modules/security/lectures/TrustedComputingTCG.html
[8] Fairness in Electronic Commerce *by N.Asokan in1998*
[9] A Fair Non-repudiation Protocol *by Jianying Zhou and Dieter Gollmann*
[10] A Multi-party Optimistic Non-repudiation Protocol *by Oliver Markowitch and Steve Kremer*
[11] An intensive survey of fair non-repudiation protocols *by Steve Kremer, Olivier Markowitch and Jianying Zhou in 2002*
[12] The ASW Protocol Revisited: A Unified View *by Paul Hankes Drielsma, Sebastian Modersheim*
[13] Asynchronous protocols for optimistic fair exchange *by N. Asokan, V.Shoup, and M.Waidner*
[14] TCG Software Stack(TSS) Specification Version 1.2 level 1 *by Trusted Computing Group in 2007 Page 1-47*
[15] Linux TCG Softerware Stack Low Level Design version 0.8 r2 *by Linux Technology Center*
[16] How to use a TPM with Linux
http://www.grounation.org/index.php?post/2008/07/04/8-how-to-use-a-tpm-with-linux
[17] Programming With TrouSerS *by David Challener in 2011*

Appendix A

Protocol verification <second design>

Protocol : fair_exchange

Types: Agent C, V;
Number Nc,Nv, CONTRACT, CARDINFO,MOVIE,ORDER,K;
Function pk,hash,sk;
Symmetric_key skvc,KCT, KVT

Knowledge: C: C, V, sk,pk(C), inv(pk(C)), pk(V),hash, skvc,KCT;
V: V, C, sk,skvc, pk(V), inv(pk(V)), hash,KVT,pk(C)

Actions:

C -> V : C, V,
{| ORDER, Nc|}skvc

V -> C : V, C, Nc,Nv,
{|hash(MOVIE), Nv,Nc|}skvc

C -> V : C, V,
{|{C,V, {|CARDINFO|}KCT, CONTRACT, Nc,Nv}inv(pk(C))|}skvc

V -> C : V, C,
{| { V,C, {C,V, {|CARDINFO|}KCT, CONTRACT, Nc,Nv}inv(pk(C)),
{|K|}KVT}inv(pk(V)) |}skvc
C -> V : C, V,
{|C,V, KCT,Nc,Nv|}skvc

V -> C : V, C,
{|V,C,KVT,Nc,Nv|}skvc

Goals:
V authenticates C on KCT
C authenticates V on KVT

Appendix B

Protocol verification <Final version>

Protocol : TPM_exchange

Types: Agent C, V, T;
Number ORDER, MOVIE, CONTRACT,Nc,Nv;
Function pk,hash,sk;
Symmetric_key skcv

Knowledge: C: C, V, T, pk(C), inv(pk(C)), pk(V),hash,sk,Nc;
V: V, C, T, pk(V), inv(pk(V)), pk(C),hash,sk,Nv;
T: C,V, pk(V),pk(C),pk(T),inv(pk(C)),hash,sk

Actions:

C -> V : C,V, {|C,V,ORDER|}sk(C,V)
V -> C : V,C, {| V,C,hash(MOVIE)|}sk(C,V)
C -> V : C,V,
{|{CONTRACT,hash(Nc),T}inv(pk(C))|}sk(C,V)
V -> C : V,C,
{|{{CONTRACT,hash(Nc),T}inv(pk(C)),
hash(Nv)}inv(pk(V))|}sk(C,V)
C -> V : C,V, {|Nc|}sk(C,V)
V -> C : V,C, {|Nv|}sk(C,V)

Goals:
V authenticates C on Nc
C authenticates V on Nv
CONTRACT secret between C,V

Appendix C

TPM Pre-Test<tpmTest.c>

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <tss/tss_error.h>
#include <tss/platform.h>
#include <tss/tss_defines.h>
#include <tss/tss_typedef.h>
#include <tss/tss_structs.h>
#include <tss/tspi.h>
#include <trousers/trousers.h>
#define DBG(message,tResult) printf("Line%d, %s)%s returned
0x%08x. %s.\n",__LINE__ ,__func__,message,tResult,(char
*)Trspi_Error_String(tResult))
#define BACKUP_KEY_UUID {0,0,0,0,0,{0,0,0,2,10}}
#define THIS_UUID_SRK {0, 0, 0, 0, 0, {0, 0, 0, 0, 0, 1}}
int main(int argc,char **argv)
{
TSS_HCONTEXT hContext;
TSS_HTPM hTPM;
TSS_RESULT result;
TSS_HKEY hSRK;
TSS_HPOLICY hSRKPolicy=0;
TSS_UUID SRK_UUID=THIS_UUID_SRK;
BYTE *pubKey;
UINT32 pubKeySize;

char someWords[100] = "----------------Program
start----------------";
printf("%s\n",someWords);
//Pick the TPM you are talking to.
result=Tspi_Context_Create(&hContext);
DBG("Create Context",result);
result=Tspi_Context_Connect(hContext, NULL);
DBG("Context Connect",result);
// Get the TPM handle
```

```
result=Tspi_Context_GetTpmObject(hContext,&hTPM);
DBG("Get TPM Handle",result);
// Get the SRK handle
result=Tspi_Context_LoadKeyByUUID(hContext,TSS_PS_TYPE_SYSTEM,
SRK_UUID,&hSRK);
DBG("Got the SRK handle", result);

//Get the SRK policy
result=Tspi_GetPolicyObject(hSRK,TSS_POLICY_USAGE,&hSRKPolicy)
;
DBG("Got the SRK policy",result);
//Then set the SRK policy to be the well known secret
result=Tspi_Policy_SetSecret(hSRKPolicy,TSS_SECRET_MODE_PLAIN,
0,"");
DBG("Set the SRK secret in its policy",result);

// Clean up
//Tspi_Context_Close(hobjects you have created);
Tspi_Context_FreeMemory(hContext,NULL);
// This frees up memory automatically allocated for you.
Tspi_Context_Close(hContext);
return 0;
}
```

Appendix D

Set wrong password of SRK

```
s104664@pcsamo:~$ ./tt
Line30, main)Create Context returned 0x00000000. Success.
Line32, main)Context Connect returned 0x00000000. Success.
Line35, main)Get TPM Handle returned 0x00000000. Success.
Line38, main)Got the SRK handle returned 0x00002020. Key not found in persistent storage.
Line41, main)Got the SRK policy returned 0x00003126. Invalid handle.
Line46, main)Set the SRK secret in its policy returned 0x00003126. Invalid handle.
*** stack smashing detected ***: ./tt terminated
```

Appendix E

Creating a Bind Key <tpmCreatBkey.c>

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <tss/tss_error.h>
#include <tss/platform.h>
#include <tss/tss_defines.h>
#include <tss/tss_typedef.h>
#include <tss/tss_structs.h>
#include <tss/tspi.h>
#include <trousers/trousers.h>
#define DBG(message,tResult) printf("Line%d, %s)%s returned
0x%08x. %s.\n",__LINE__ ,__func__,message,tResult,(char
*)Trspi_Error_String(tResult))
#define THIS_UUID_SRK {0, 0, 0, 0, 0, {0, 0, 0, 0, 0, 1}}
#define BINDING_KEY_UUID {0,0,0,0,0,{0,0,0,2,37}} //choos your
key's UUID
int main(int argc,char **argv)
{
TSS_HCONTEXT hContext;
TSS_HTPM hTPM;
TSS_RESULT result;
TSS_HKEY hSRK;
TSS_HPOLICY hSRKPolicy=0;
TSS_UUID SRK_UUID=THIS_UUID_SRK;
BYTE *pubKey;
UINT32 pubKeySize;

TSS_UUID KEY_UUID=BINDING_KEY_UUID;
TSS_HPOLICY hBackup_Policy;
TSS_FLAG initFlags;
FILE *fout;
TSS_HKEY hESS_Bind_Key;

char someWords[100] = "----------------Program
start----------------";
```

```
printf("%s\n",someWords);
//Pick the TPM you are talking to.
result=Tspi_Context_Create(&hContext);
DBG("Create Context",result);
result=Tspi_Context_Connect(hContext, NULL);
DBG("Context Connect",result);
// Get the TPM handle
result=Tspi_Context_GetTpmObject(hContext,&hTPM);
DBG("Get TPM Handle",result);
// Get the SRK handle
result=Tspi_Context_LoadKeyByUUID(hContext,TSS_PS_TYPE_SYSTEM,
SRK_UUID,&hSRK);
DBG("Got the SRK handle", result);
//Get the SRK policy
result=Tspi_GetPolicyObject(hSRK,TSS_POLICY_USAGE,&hSRKPolicy)
;
DBG("Got the SRK policy",result);
//Then set the SRK policy to be the well known secret
result=Tspi_Policy_SetSecret(hSRKPolicy,TSS_SECRET_MODE_PLAIN,
0,"");
// Use the 20 bytes as they are.
DBG("Set the SRK secret in its policy",result);


///////Create a policy for the new key
result=Tspi_Context_CreateObject(hContext,TSS_OBJECT_TYPE_POLI
CY,TSS_POLICY_USAGE,&hBackup_Policy);
DBG("Create a backup policy object",result);
result=Tspi_Policy_SetSecret(hBackup_Policy,TSS_SECRET_MODE_PL
AIN,0,"");
DBG("Set backup policy object secret",result);
// Create a key object
initFlags =
TSS_KEY_TYPE_BIND|TSS_KEY_SIZE_2048|TSS_KEY_NO_AUTHORIZATION|T
SS_KEY_NOT_MIGRATABLE;
result=Tspi_Context_CreateObject(hContext,TSS_OBJECT_TYPE_RSAK
EY,initFlags,&hESS_Bind_Key);
DBG("Create the key object", result);
// Set the padding type
result =
Tspi_SetAttribUint32(hESS_Bind_Key,TSS_TSPATTRIB_KEY_INFO,TSS_
TSPATTRIB_KEYINFO_ENCSCHEME,TSS_ES_RSAESPKCSV15);
```

```
DBG("Set the key's padding type", result);

// Assign the key's policy to the key object
result=Tspi_Policy_AssignToObject(hBackup_Policy,hESS_Bind_Key
);//hESS_Policy
DBG("Assign the key's policy to the key", result);
// Create the key, with the SRK as its parent.
printf("Creating the key could take a while\n");
result=Tspi_Key_CreateKey(hESS_Bind_Key,hSRK,0);
DBG("Asking TPM to create the key", result);

result=Tspi_Context_RegisterKey(hContext,hESS_Bind_Key,TSS_PS_
TYPE_SYSTEM,KEY_UUID,TSS_PS_TYPE_SYSTEM,SRK_UUID);
DBG("Register the key for later retrieval", result);
printf("Registering key for later retrieval\r\n");

//load key into tpm
result=Tspi_Key_LoadKey(hESS_Bind_Key,hSRK);
DBG("Load key in TPM", result);

result=Tspi_Key_GetPubKey(hESS_Bind_Key,&pubKeySize,&pubKey);
DBG("Get public portion of key", result);
// Save it in a file. The file name will be "BackupESSBindKey.pub"
fout=fopen("BackupESSBindKey.pub", "wb");
if(fout != NULL)
{
write(fileno(fout),pubKey,pubKeySize);
printf("Finished writing BackupESSBindKey.pub\n");
fclose(fout);
}
else
{
printf("Error opening BackupESSBindKey.pub \r\n");
}



// Clean up
Tspi_Policy_FlushSecret(hBackup_Policy);
//Tspi_Context_Close(hobjects you have created);
Tspi_Context_FreeMemory(hContext,NULL);
// This frees up memory automatically allocated for you.
```

```
Tspi_Context_Close(hContext);
return 0;
}
```

Appendix F

The result of UUID has already registered

```
s104664@pcsamo:~$ ./tpmTest
----------------Program start----------------
Line32, main)Create Context returned 0x00000000. Success.
Line34, main)Context Connect returned 0x00000000. Success.
Line37, main)Get TPM Handle returned 0x00000000. Success.
Line40, main)Got the SRK handle returned 0x00000000. Success.
Line44, main)Got the SRK policy returned 0x00000000. Success.
Line47, main)Set the SRK secret in its policy returned 0x00000000. Success.
s104664@pcsamo:~$ ./creatKey
----------------Program start----------------
Line41, main)Create Context returned 0x00000000. Success.
Line43, main)Context Connect returned 0x00000000. Success.
Line46, main)Get TPM Handle returned 0x00000000. Success.
Line49, main)Got the SRK handle returned 0x00000000. Success.
Line52, main)Got the SRK policy returned 0x00000000. Success.
Line57, main)Set the SRK secret in its policy returned 0x00000000. Success.
Line62, main)Create a backup policy object returned 0x00000000. Success.
Line64, main)Set backup policy object secret returned 0x00000000. Success.
Line68, main)Create the key object returned 0x00000000. Success.
Line71, main)Set the key's padding type returned 0x00000000. Success.
Line75, main)Assign the key's policy to the key returned 0x00000000. Success.
Creating the key could take a while
Line79, main)Asking TPM to create the key returned 0x00000000. Success.
Line82, main)Register the key for later retrieval returned 0x00002008. UUID already register
ed.
Registering key for later retrieval
Line87, main)Load key in TPM returned 0x00000000. Success.
Line90, main)Get public portion of key returned 0x00000000. Success.
Finished writing BackupESSBindKey.pub
```

Appendix G

Read the public portion of the Binding key

```
s104664@pcsamo:~$ vi BackupESSBindKey.pub

^@^@^@^A^@^B^@^A^@^@^@^L^@^@^H^@^@^@^@^B^@^@^@^@^@^@^A^@º8c^_^UÌìçG¶ö^EGÉ^PôL5Å<88>^<83>
zBÖ^WÑ¢7<8a>Ù <89>Rí^Fn<89>Õìù^EïsF|Æ^DË)vE¯+^Q%ÕÀTé!<<8d>í)^Na¨CRâTñÿpÔÜë×ÐT,ÝdBÏÙt¶jý
p+^?W<85><98><92>»*Í¸^BÛ·M^Hn¬^Zø)^H:m¿<9a>f<95><yª)â?=^M|}y¢IÕj`Û^Uæ^ZwRí^VÀþ<82>\^]/`s
ý^Z÷<90>&^Wµf<8c>½è^ö^P+ÁØÊ^A|<9d>aZñÝFo^P<91>^M<85><8a> äðBÕDû²^RÓ¡wç±«³\ðþm·<97><99>i<
84>ä/<9c>É§<92>ìøÕ^S<92>®ßâÕ8<9c><Åí<88>^T¥_qéxVMÅQAÖ3·'^LE¼^?ÿ<9b>æ¢µz5
~
~
```