Abstract

This dissertation is a report of my project:”Security of the Privacy-CA solution.” The aim of my project is to verify the security of the protocol Privacy-CA solution with an automatic protocol verification tool ProVerif.

Privacy Certification Authority solution (PCAS) is a protocol for the authentication of a platform, and more specifically, it is used to authenticate a platform by verifying whether a module of the platform is legal or not. It is published by a group named Trusted Computing Group (TCG). TCG also published a protocol Direct Anonymous Attestation (DAA) which plays a similar role as PCAS. ProVerif is an automatic verification tool which is used to verify cryptographic protocol commonly. There are many researches verify the DAA protocol by using ProVerif, however, not so many similar researches on the PCAS protocol. This dissertation is aimed to analysis the PCAS protocol and performs a suitable model for PCAS in ProVerif to verify the security and secrecy.

This dissertation makes several achievements.

1) From page 2 to page 11, the dissertation introduces ProVerif.

2) From page 12 to page 19, the dissertation gives information on TCG and Privacy-CA.

3) Form page 20 to page, three versions of PCAS protocol has been built. The dissertation gives specific analysis on the procession and result of each model.

Table of content:

1. Introduction

On the public network or private network, there are data and messages transferred or exchanged in or between equipments. These equipments maybe a computer system, an ATM, a telephone, even a chip on the main board. People use protocols to make rules for these operations. Protocol describes data formats and orders the way they communicated. In the cryptographic field, security protocol is a communication protocol based on the cipher system, by using the key algorithm to get the aims of securing data transport or certificate authority. Before a protocol is published, researcher must verify the security of the protocol, make sure it is strong and hard to be against. On the other side, researchers try to find methods to against an existing cryptographic protocol, then give their theory or help to improve the protocol.

Formal methods of verify cryptographic protocols are categorized as two classes: the computational approach and symbolic approach. The symbolic approach assumes the cryptographic scheme is perfect, but do not prove the correctness of the perfect assumption. Compare to symbolic approach, computational approach focus on the actual protocol implementation, using mathematic proof to analyse the protocol. The computational approach always gives all conditions of a protocol, and then the verifier (always human) gives some hypothesizes on the conditions and uses mathematic methods to proof the probability of the assumptions. The symbolic approach is that the verifier performs a protocol in a certain language (possibly logic language), then gives all the possible routes and results, finds the routes and results which break the security or not secret. Automatic tools help verifiers to do the job. Most of the analysis tools are based on the symbolic approach. There are many automatic analysis tools exist, such as Interrogator [12], ProVerif [13] and so on.

The aim protocol of this dissertation is Privacy Certification Authority solution (PCAS). The goal of this report is to verify the protocol at a suitable level of its specification.

2.Analysis tools

Normally there are many kinds of analysis tools which can verify security of a protocol. Verifier based on symbolic computation is popular and well-designed in this field. Most of them are based on the Dolev-Yao model [14]. The verifier separates the protocol and the algorithm of a protocol. Verifier always assumes the algorithm in the protocol is perfect. Normally, the validity of this assuming is not considered. Then the verifier describe the protocol, gives each element of the protocol a symbol with certain meaning. Then produce a model based on the relationship between different symbols. After that, the verifier defines a secure aim, such as confidentiality, authentication, non-repudiation and anonymity. At last, the verifier gives the illation whether the protocol model is fit for the secure aims. There are two methods which is the most popular for a verifier.

1) The first is by using logic analysis. The most famous example is the BAN logic [15]. BAN logic uses postulates and definitions to build a logic system with logic of belief and action. The basic definitions and implications are shown below:

   - $P \mid\equiv X$: P believes that X is true.
   - $P \Longrightarrow X$: P has jurisdiction over X.
   - $P \mid\sim X$: P used to sends a message X.
   - $P \triangleleft X$: P receives message X.
   - $\{X\}_K$: X is encrypted with key K.
   - #(X): X is freshness, has not previously been sent in any message.
   - $P \overset{K}{\leftrightarrow} Q$: P and Q can communicate with key K, only P, Q and entity they trusted know K.
   - $\overset{K}{\rightarrow} p$: P has K as a public key, only P and entity P trusted knows the decryption key.
   - $P \overset{X}{\Leftrightarrow} Q$: X is a secret key only known to P and Q.
   - $<X>_Y$: The message that X combined with secret Y.

   By the help of these basic rules, verifier can change the protocol to a serious function, and use logic illation to distinguish whether the protocol can win the assuming secure aims.

2) The second method is trying to find an attack on the protocol by checking the protocol. Firstly, verifier describes the original conditions of the protocol. Then verifier list all the knowledge the attacker knows. After given the verifier an aim, the verifier will exchange and act all the states as routes, and search all the possible routes the protocol and the attacker can reach. Normally, this procession

is using automatically operation. Finally the verifier can know if the secure aim is satisfied with all the routes, then get the conclusion. If there is any route does not satisfy with the aim, verifier will give the conditions.

ProVerif [13] is one of the most popular verifiers which use the second method. The advantages of this method is that it is automatic verify the protocol, user do not need to control the verification procession. If there is weakness of a protocol, verifier will give an example to show the state of the weakness. The disadvantages includes: firstly, to a too complex protocol, it is hard for verifier to control the routes it acts; secondly, certain parameters need to be input to the verifier, if the protocol pass the verification, it is not means the protocol is satisfied with the secure aims.

Also there is the other method, computational approach. It focuses on the computation of probability and complexity and need to make the verification by hand. In this way, the security properties are considered under the real situation of the protocol works. So the analysis results are normally accuracy and trustworthy. This method are complex, and the verifier must have capability on mathematic. Normally it assumes that the algorithm in the protocol is not perfect; the attacker can attack both the protocol and the algorithm, then try to use mathematic proof to find attack strategy. Because it is complex and hard to be automatically, there are not many tools exist. However, cryptographers usually use this computational approach to prove and analyse their theory.

2.1 Introduction to ProVerif

The protocol analysis tool ProVerif was developed by B. Blanchet in 2001 [14]. Now the version is 1.85. According to the author, ProVerif has several advantages: it avoids the problem of the state space explosion by the use of unification; it is not need to limit the number of runs of the protocol to analyse it; it is efficient, many protocols can be analysed by ProVerif with very small resources.

The main structure of ProVerif is represented in Figure 1. There are two kinds of logic clauses using in ProVerif [14]: the Horn clauses and the applied Pi clauses. The working procedure of ProVerif is shown in the Figure 1. In the input process, ProVerif takes a model of the protocol in the Pi clauses with cryptography. Also, the properties which need to prove have to be input in the applied Pi clauses. Then the translator translates the protocol model from applied Pi clauses into an internal representation by the Horn clauses. The properties also translate into a set of derivability queries. After ProVerif gets the clauses and the queries, it begins to use an algorithm which can free to select and determine whether a query is derivable from the clauses. If the query is not derivable, which means the properties is proved, ProVerif will give the result security aim is proved. If the query is derivable, there may be several reasons.

Firstly, exists an attack that can against the properties, and then ProVerif will give the conditions how the attack happened. Secondly, the ProVerif may output “false attack”. There are many reasons can lead to a false attack [16], such as: the property is wrong; two different values of random are associated to the same parameter; or there is an internal approximation made by ProVerif.

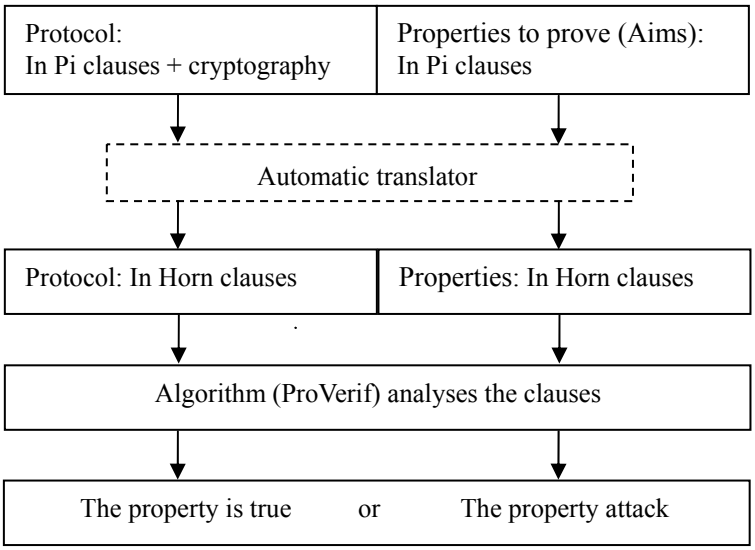

Fig.1. Structure of ProVerif.

There are two logic clauses exist in ProVerif. User can either input the Pi clauses or the Horn clauses. The difference is that if the input is in applied Pi clauses, it will be translated to the Horn clauses. If the input is in the Horn clauses, ProVerif will directly call the algorithm to determine the clauses and queries.

The syntax of a clause using in ProVerif is shown in Figure 2. In fact, the syntax of a clause using in ProVerif contains two kinds of grammar, both the Horn clauses and applied Pi clauses. The Horn clauses and applied Pi clauses will be introduces respectively.

$M, N ::=$	terms (data)
x, y, z	variable
a, b, c, k, s	name
$f(M_1 \ldots M_n)$	constructor application
$P, Q ::=$	processes
$out(M, N).\, P$	output
$in(M, x).\, P$	input
0	nil
$P \mid\mid Q$	parallel composition
$!\, P$	replication
$new\ a.\, P$	restriction
$let\ x = g(M_1 \ldots M_n) in\ P\ else\ Q$	destructor application
$let\ x = M\ in\ P$	local definition
$if\ M = N\ then\ P\ esle\ Q$	conditional

Fig.2. Syntax of a clause using in ProVerif. Source is from [18].

2.2 Horn clauses

In 1951, a logician named Alfred Horn firstly point out the significance of this kind of clauses [17]. So Horn clauses are named by Alfred Horn's name. A Horn clause is a clause with at most one positive literal. It means that a horn clause must have a word with no symbol meaning.

Because there are positive and negative literal exists, so there are four types of horn clauses, which is that a clause contains both positive and negative literal; contains only positive literal; contains only negative literal; contains neither positive nor negative literal.

According to the four types of clauses, there are four definitions. A clause contains one positive literal and at least one negative literal called a rule; a clause only contains one positive literal called a fact or a term; a clause only contains one negative literal called a negated aim; a clause contains neither positive nor negative literal called a null clause.

In Figure 2, a, b, c, k, s are names, and x, y, z are variables. The syntax assumes a set of symbols for constructor f and destructor g. Constructor is used to build terms (data). Normally, terms of Horn clauses contain names, variables, and constructor applications; the terms are untyped. Destructors do not act in the terms; they operate terms in the processes. They are partial functions on terms that processes can apply. For example, the process $let\ x = g(M_1 \ldots M_n)\ in\ P\ else\ Q$ is a destructor application

used to evaluate $g(M_1 \ldots M_n)$. If it works, then x is bounded to the form $g(M_1 \ldots M_n)$ and process P is executed, else process Q is executed.

2.3 Applied pi clauses

In 2001, Abadi extended the Pi clauses to the applied Pi clauses based on extending the lambda clauses to the applied lambda clauses [19]. In Figure 2, there are many constructs come from the applied Pi clauses.

a) $\mathrm{in}(M, x).P$: inputs a message on channel M, and executes P with x bound to the input message.
b) $\mathrm{out}(M, N).P$: outputs the message N on the channel M and then executes P. (The term M which is used to present as a channel can be either a name or a variable, even a constructor application.)
c) 0: sown as nil process, does nothing.
d) P||Q: the parallel composition of P and Q.
e) !P: an unbounded number of copies of P in parallel.
f) new a.P: creates a new name a, and then executes P.
g) let $x = M$ in P: executes P with x bound to the term M.
h) if $M = N$ then P esle Q: executes P if M and N are the same term at onetime, else executes Q. if $Q \equiv 0$, this condition process can be if $M = N$ then P.

According to Abadi [19], there are some special definitions.

- Definition 1: Structural equivalence. It is the minimum equivalence defined on the extended processes. Parameters are shown as follows:

PAR-0	$A \equiv A\|0$
PAR-A	$A\|(B\|C) \equiv (A\|B)C$
PAR-C	$A\|B \equiv B\|A$
REPL	$!P \equiv P\|!P$
NEW-0	$(\nu n)0 \equiv 0$
NEW-C	$(\nu m)(\nu n)A \equiv (\nu n)\,(\nu m)A$
NEW-PAR	$(\nu n)(A\|B) \equiv A\|(\nu n)B$, if $n \notin f_n(A)$
ALIAS	$(\nu x).\{M/x\} \equiv 0$
SUBEST	$\{M/x\}\|A \equiv 0\{M/x\}\|A\{M/x\}$
REWRITE	$\{M/x\} \equiv \{M/x\}$ when $\sum \vdash M = N$

By the definitions, every closed extended process can be written to an active substitution and a plain process:

$$A \equiv (\nu \tilde{n}) \left\{ \frac{\nu \tilde{M}}{\nu \tilde{x}} \right\} |p$$

- Definition 2: internal reduction. It is the minimum reduction of the structural equivalence and the context. The definition is:

COMM	$\bar{a}\langle x\rangle.P \mid a(x).Q \rightarrow P \mid Q$
THEN	if M = M then P else $Q \rightarrow P$
ELSE	if M = N then P else $Q \rightarrow Q \sum M = N$

In the Pi clauses, it has

$$\bar{a}\langle M\rangle.P \mid a(x).Q \equiv (\nu x).(\{M/x\} \mid \bar{a}\langle x\rangle.P \mid a(x).Q) \equiv (\nu x).(\{M/x\} \mid P \mid Q) \equiv P \mid Q\{M/x\}$$

- Definition 3: Observational equivalence. If there is a $(\text{context})^* C[]$, which makes $A \rightarrow^* C[\bar{a}\langle Mx\rangle.P]$, we define it $A \Downarrow a$.

  1) If $A \Downarrow a$, then $B \Downarrow a$;
  2) If $A \rightarrow^* A'$, then there are some B' that $B \rightarrow^* B'$ and $A' \mathcal{R} B'$;
  3) To any of the closed context, there is $C[A] \mathcal{R} C[B]$.

ProVerif can use applied Pi clauses as the input language, however, the input has to be transferred to the form of Horn clauses.

2.4 The attacker abilities in ProVerif

In the procession of verifying the model of protocol, ProVerif defines a set of computation abilities of the attacker. This is a very important part to ProVerif, because ProVerif need to use these abilities to prove the security properties. ProVerif [16] defines the computational abilities for the attacker in Horn clauses and in applied Pi clauses respectively.

- Clauses from the applied Pi calculus
  Definition:

F ::=	facts
$\text{attacker}(x)$	attacker knowledge
$\text{mess}(y, m)$	message on a channel

The facts $\text{attacker}(x)$ means attacker may have the knowledge x. $\text{mess}(x, x')$ means the message m may appear in the channel y. The clauses are in the form of a set of facts $F_1 \wedge F_2 \wedge \ldots \wedge F_n \Longrightarrow F$. The abilities of attacker in applied Pi clauses are:

- For each $a \in S$, attacker(a[]) (Init)
- attacker(b_0[]) (Rn)
- For each public constructor f of a arity n,
  $\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$ (Rf)
- For each destructor g,
  For each rewrite rule $g(M_1, \ldots, M_n) \rightarrow M$ in def(g) (Rg)
  $\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M)$
- $\text{mess}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y)$ (Rl)
- $\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{mess}(x, y)$ (Rs)

The meanings of each clause are:

- (Init): the initial knowledge the attacker has.
- (Rn): the attacker can generate new name.
- (Rf): the attacker can do all the operations to the constructors.
- (Rg): the attacker can do all the operations to the destructors.
- (Rl): the attacker can listen on all the channels he has.
- (Rs): the attacker can send all messages he has on all channels.

➢ Clauses from the Horn calculus

The clauses in the form if applied Pi clauses will be transferred into Horn clauses. ProVerif has its Horn clauses on describing attacker in three ways [19]:

1) Computational abilities: constructors, destructors, and name generation.
2) Facts that Attackers have initial knowledge with them. E.g. the public key and the name of participants.
3) The protocol contains a set of messages. To the ith message, the p_{ji} is the pattern of the message received by a principal X, before sending the ith message. p_i is the pattern of ith message, there is:

$$\text{attacker}(P_{j1}) \wedge \ldots \wedge \text{attacker}(P_{ji}) \Rightarrow \text{attacker}(P_i)$$

Clauses in the Horn calculus:

For each constructor f of arity n:
$\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$
For each destructor g, for each rewrite rule $g(M_1, \ldots M_n) \rightarrow M$ in def(g)
$\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M)$
that is

aenc	$\text{attacker}(m) \wedge \text{attacker}(pk) \Rightarrow \text{attacker}(\text{aenc}(m, pk))$
pk	$\text{attacker}(sk) \Rightarrow \text{attacker}(pk(sk))$
pdec	$\text{attacker}(\text{anec}(m, pk(sk))) \wedge \text{attacker}(sk) \Rightarrow \text{attacker}(m)$
sign	$\text{attacker}(m) \wedge \text{attacker}(sk) \Rightarrow \text{attacker}(\text{sign}(m, sk))$
getmess	$\text{attacker}(\text{sign}(m, sk)) \Rightarrow \text{attacker}(m)$

check	$\text{attacker}(\text{sign}(m, sk)) \wedge \text{attacker}(\text{pk}(sk)) \Rightarrow \text{attacker}(m)$
senc	$\text{attacker}(m) \wedge \text{attacker}(k) \Rightarrow \text{attacker}(m, k)$
sdec	$\text{attacker}(\text{senc}(m, k)) \wedge \text{attacker}(k) \Rightarrow \text{attacker}(m)$
name generation	attack(a[])

Initial knowledge is attacker(pk(sk_A[])), attacker(pk(sk_B[])). So protocol is:
First message: $\text{attacker}(\text{pk}(x)) \Rightarrow \text{attacker}(\text{aenc}(\text{sign}(k[\text{pk}(x), sk_A])\text{pk}(x)))$
Second message: $\text{attacker}(\text{aenc}(\text{sign}(y, sk_A[]), \text{pk}(sk_B[]))) \Rightarrow \text{attacker}(\text{senc}(s, y))$

Fig.3. Summary of attacker's contribution. Source is from [28].

Besides the abilities of the attacker, there are also clauses which describe the protocol. This part will be studied in the future.

2.5 The principle of proof search in ProVerif

Blanchet's paper [13] gives how to search the proof in ProVerif. Before we know the principle, there is a definition of most general unifiers must be known.

Most general unifiers is also called mgu, which is a substitution σ of a set of expressions E on the conditions of it unifies E and for any unifier ω of E, there is a unifier λ which satisfied with $\omega = \sigma \circ \lambda$.

Suppose $R \triangleq G \Rightarrow F$ and $R' \triangleq H \wedge F_0 \wedge H' \Rightarrow F'$. G, H, H′ is conjunctions of facts. Then we denote σ is the most general unifier of F and F_0, we have

$$R \circ_{F_0} R' \triangleq \sigma(H) \wedge \sigma(G) \wedge \sigma(H') \Rightarrow \sigma(F')$$

After giving a computational abilities of an attacker, a property A need to prove, and a set of rules R. An R-derivation can be seen as a tree. Each node of the tree is a fact. It is following the definition: if $\{G_1, \ldots, G_n\}$ is the children set of R, then $G_1 \wedge \ldots \wedge G_n \Longrightarrow R$ is an instance of some rule G in R. The search algorithm works in two steps. On the first step, ProVerif takes up rules M and M′ in R and F_0 satisfied to the conditions: $M \circ_{F_0} M'$ exists; F_0 is not in the form of attacker(x); M has its left side as the form of $\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n)$. Then append the $M \circ_{F_0} M'$ into the rules set R. After that, delete useless rules in R. Finally, return to start the first step again until there is no such M and M′ exists.

On the second step, ProVerif begins a search from the root of the derivation tree, of which the property A is the root. Then ProVerif tries to grow an R-derivation tree. A depth-first search is used to help to build the tree. A branch of the tree grows firstly until all the nodes of this branch are complete, and then turns to the next branch.

By using such a method, ProVerif can find the nodes which are satisfied with the conditions of the query, and return the result by listing all the facts; otherwise, there is

no node fit for the query, which means the property A is proved.

2.6 Evaluation of ProVerif

In the previous section, I have shown the survey on ProVerif, including the syntax of input language, some analysis of the function of this verifier. In this section, I will evaluate ProVerif from two points. Firstly, I will discuss the limitation of ProVerif. Secondly, I will compare ProVerif with other protocol verification tools.

M. Peters and P. Rogaar [20] discuss the limitation to ProVerif: Approximation, XOR-operator, and Diffie-Hellman key exchange. Approximation in ProVerif is used to limit the number of runs of protocol. When ProVerif verifies a protocol with a party uses one rules of the protocol definition many times, the Approximation can only results in more false positive, but has no influence on the sensitivity of the protocol.

Many protocols use this XOR-operator in their definition. However, ProVerif does not provide any algebraic structure of the primitives. Kusters and Truderung describe a solution in [23] by using a special form $\oplus -\text{linear}$, and also publish an improve version XOR-ProVerif of the origin to solve the problem.

The Diffie-Hellman key exchange means that the protocol enables two participants run to construct a shared key without any previously shared secret. ProVerif cannot verify such a protocol at first, because it is too hard for ProVerif directly assert properties by using such primitives. Kusters and Truderung [24] give a solution. They rewrite the protocols without using such primitives so that these protocols can be encoded as Horn clauses, yet there is no improvement for ProVerif on original Diffie-Hellman key exchange.

There are many features can be used to evaluate the worthy of a verifier. The author of ProVerif said that the protocol is efficient, and listed a series reason [13]. It avoids listing number of runs of the protocol, by using a new algorithm. It avoids the state space explosion by using unification. It verifies many protocols using small resources and less time.

Research by Chirs [21] supports the point of view. Chirs in [21] devise an evaluation methodology to compare different protocol verifiers. He verifies four features of the CPSA (Cryptographic Protocol Share Analyser [22]) and ProVerif. The four features are Scope, Correctness, Performance and Usability. CPSA is a text-based system designed for using on POSIX based platform. It has two kinds of modes: interactive mode and batch mode. The experiment tests CPSA in the batch mode. Table 1 shows the results of the performance for CPSA and ProVerif.

Protocol	Execution Time(CPSA)	Execution Time(ProVerif)
NS	5.49sec	0.09sec
NSL	13.94sec	0.05sec
Kerberos	60+hr	0.03sec
Memory use	8.348Mb	6.124MB

Table 1. Performance of CPSA and ProVerif. Source is from [21];

From the data in Table 1, it is clearly shown that ProVerif is an efficient verifier. All the execution time of verifying three protocols (NS, NSL, Kerberos), ProVerif using much less then the compare tool CPSA. ProVerif also uses less memory than CPSA.

3. TCG and the Privacy-CA

3.1 Trusted Platform

After the first personal computer is invented, it plays an important role in almost every field in the human society. The most common computer systems (including personal computer and commercial operating system) are open platform. They are open and flexible, which supported people to run different software or set different hardware as they needed. This improves development of computer, however, the hardware and software on the platform is not credible. On the opposite, there is closed platform, which uses particular hardware to run particular software and is accessed from certain interface by verifying the privacy key. The benefit of closed platform computing is that it is security and credible, however, it cannot be used widely.

This situation brings problems: how we can know a computing platform is trustworthy, how we can know software from the third party is trustworthy. Trust, in a word it means believable. The common definition is that "Trust is that a psychological state comprising the intention to accept vulnerability based upon positive expectations of the intentions or behavior of another [1]." Standard ISO/IEC gives an explanation of trust in the information technology filed: "A trusted component, operation, or process is one whose behavior is predictable under almost any operating condition and which is highly resistant to subversion by application software, viruses, and a given level of physical interference [2]."

Trusted Platform is designed between the definition of open platform and closed form. It at least has several basic features. Protection: compute and save data by a trust way. Integrality: verifying the integrality of the platform code and data. Authentication: the platform can verify the platform itself and any other third party and show the security results to the platform user. Trusted Platform is focuses on using the common platform structure with an additional chip, Trusted Platform Module (TPM), by which the security of such operation can be proofed and controlled.

3.2 TCG and TPM

Before 1999, people began to produce coprocessor [3] which is used to solve the security problem on the computing platform such as IBM 4758 [4]. However, there is not a standard until the Trusted Computing Platform Alliance [5] (TCPA) was founded in 1999. The alliance contains AMD, Hewlett- Packard, IBM, Intel and Microsoft. The aims of TCPA are including: firstly, to improve the trustworthiness and security of the common computing platforms; secondly, to found a mechanism which is used to verify whether a computing platform is a trusted platform.

In 2003, the TCPA was renamed Trusted Computing Group (TCG), with membership of 14 companies, including Promoters and board members AMD, Hewlett-Packard, IBM, Intel Corporation, Microsoft, Sony Corporation and Sun Microsystems, Inc. TCG organization are creating structure and standard to promise the extension of trusted computing beyond the personal computer into the commercial computer. TCG adopts existing specifications for the Trusted Platform Module (TPM) secure cryptoprocessor for clients and publishes as an open industry specification. And then, some famous international processor/chip suppliers like National semiconductor, Infineon Technologies began to produce TPM cryptoprocessor based on the specification.

At the same time, Microsoft also offered a similar specification: Next Generation Secure Computing Base (NGSCB). P.Englan specified NGSCB in his article [25], and suggested that using secure hardware base to achieve trusted computing. Then, Microsoft chose TPM as its secure hardware base [26]. NGSCB has been used in the operation system Windows Vista. When a pc is started, the TPM in NGSCB will check whether there is malicious code exists in the pc.

3.3 Structure and important definitions of TPM.

In 2004, TCG released the Trusted Platform Module (TPM) specification 1.2. According to TCG [7], there are four type keys on the TPM:

- Endorsement Key (EK): this is a 2048-bit RSA key pair, with the name of public portion the PUBEK and private portion the PRIVEK. Every TPM must have and only have one EK. The PRIVEK only exists in a TPM-shielded location. It is used to show the identification of the TPM. It is produced when the TPM is produced.

- Attestation Identity Key (AIK): this is a pair of public/private key generated by the TPM owner, used to perform signatures for data from TPM and information of the TPM. Because every TPM only has one EK and cannot be changed, in order to protect PUBEK and PRIVEK of EK, TPM owner generates AIK use to instead of EK. TPM owner can package its EK and other related certificates into an AIK request and send the request to a third party. The third party verifies the EK to show it is from a valid TPM then return a valid AIK.

- Storage Root Keys (SRK): this is used to provide security storage for data and other keys. It is generated when the owner holds the TPM. The private key of SRK exists in a TPM-shielded location. All the other keys are generated by SRK directly or indirectly. It is the root of the key system of TPM. When clean out the owner of a TPM, the SRK is also cleared, all the keys and data signed by SRK (directly or indirectly) cannot be used.

- Signing Keys: this is used to sign the common data and messages.

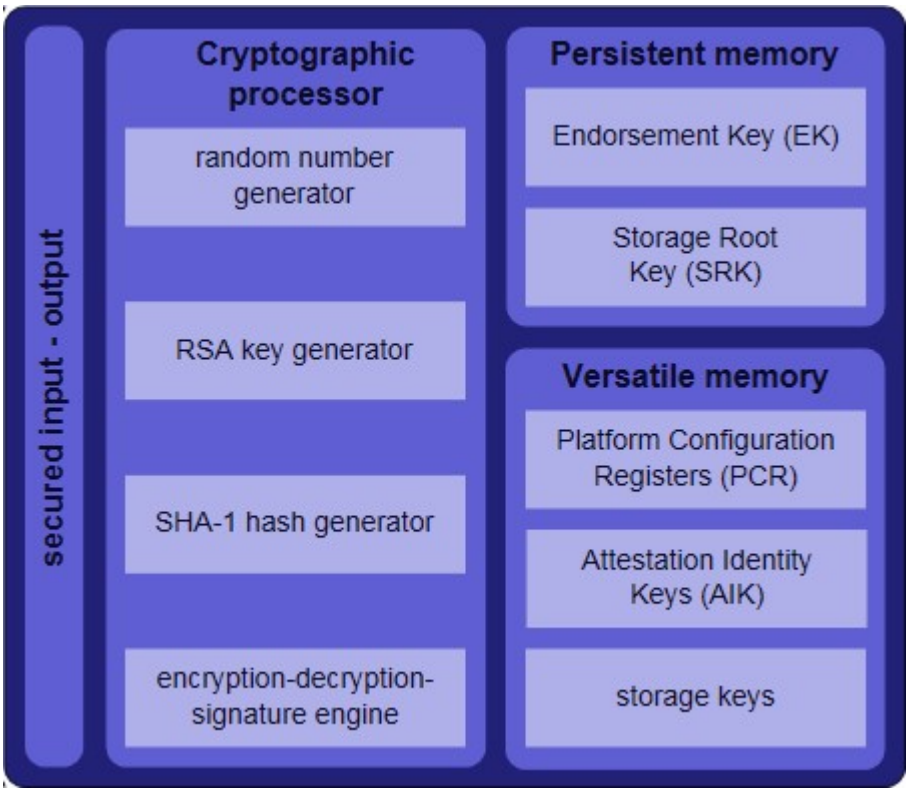

Fig.4. Simplified schema of a Trusted Platform Module. Source is from [8].

Attestation is an important function of the TPM. It means that TPM need to show its security Identity to other entities. By processing attestation, the external entity can distinguish whether the platform is trustworthy. A TPM contains three kinds of roots of trust [7].

- Root of Trust for Measurement (RTM). The RTM is used to make sure that the TPM is worked on a trusted platform. All the RTM codes are executing when the platform is powering up then cannot be changed, so they are believable. One trustworthy operation firstly measures its measurement and save it secure, then begins the next operation. By this way of recursion, the trustworthy operation system is started.

- Root of Trust for Storage (RTS). The RTS is used to protect data used by TPM but saved in external storage devices. RTS is also used to make sure that all the keys generated by TPM, the private portion only used in the TPM-shielded location.

- Root of Trust for Reporting (RTR). The RTR is in charge of establishing platform identities, providing a function for attesting to reported values, reporting platform configurations, and protecting reported values.

When users start a platform with TPM, they can get data measured by RTM, saved by RTS and reported by RTR. They can compare the data with believable data from a trusted third party, and then they will know whether the platform is secure and believable.

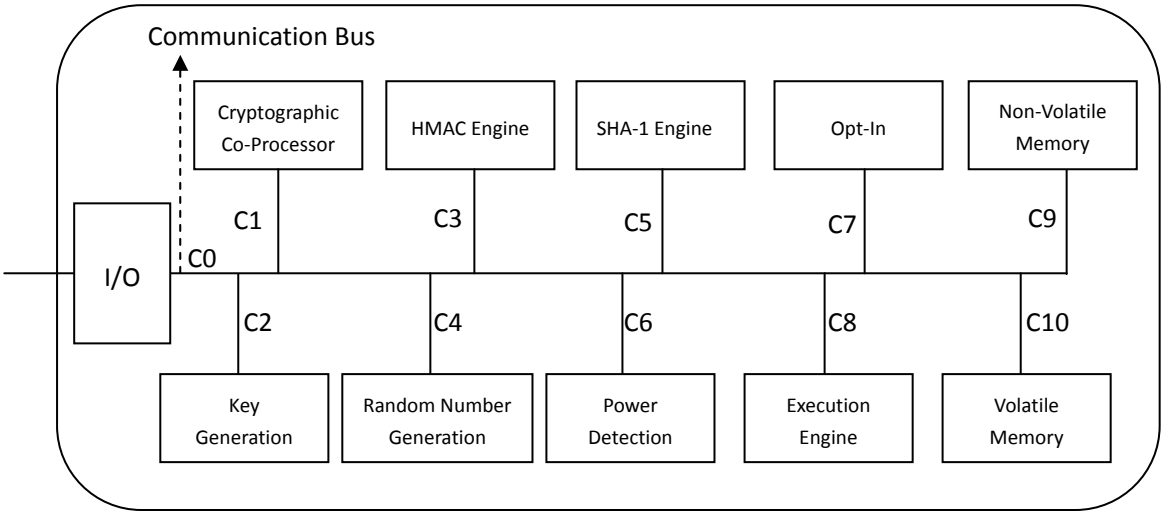

Fig.5. Architecture of a Trusted Platform Module. Source is from [27].

Figure 5 shows the architecture and communication bus structure of TPM. It contains I/O, cryptographic co-processor, key generation, HMAC engine, random number generation (RNG), SHA-1 engine, power detection, opt-in, execution engine, and memories.

- I/O: I/O, which is related to C0, is in charge of managing the information flow on the communication bus. It just transfers messages from the platform to a certain component of TPM, and returns the encrypt/decrypt messages. There are many TPM functions in the module, the platform cannot control the functions directly, it calls the functions and gets results via the I/O component.

- Cryptographic co-processor: This is the core of a TPM, it plays encryption or decryption operations inside TPM. These operations contains key generation of RSA, encrypt/decrypt of RSA, SHA-1, HMAC and Random Number Generation (RNG).

- Key generation: It can generate key of symmetrical encryption and key pair of asymmetric encryption. There is no limitation on the generation of the keys.

- HMAC engine: It is used to check the validity of data in and out the TPM. This computation function is followed the definition of RFC2104, with 20 bits cipher and 64 bits block.

- Random number generation (RNG): It is in charge of generating random numbers

which the TPM needs, with the length of 32 bits. It is generated inside TPM, cannot be predicted by any entity besides TPM.

- SHA-1 engine: This is the mainly hash function used by TPM. It can be called by the platform and output 160 bits data.

- Power detection: TPM has strict requirement on the power management. The power of TPM can be connected to the power of the trusted platform, power detection component can help to take suitable modification when the power changes.

- Opt-in: This is the switch of TPM. TPM can be open or closed by given order to the opt-in component. Before executing the order, the validity of the order must be verified.

- Execution engine: This component is in charge of the execution of the command code. The command code is transferred to TPM via the I/O component, executes by the execution engine. Before executing the command code, security of the executing environment must be verified.

- Memories: Volatile Memory is used to store any data (key or cipher) TPM used. Non- Volatile Memory is used to store the special keys such as EK and SRK.

3.4 Solutions given by TPM

In a TCG system, we suppose there are Alice and Bob. Alice holds a TPM. Bob wants to get access to Alice. Alice has an EK and several AIKs. Alice wants Bob to know that she has a Trusted Platform with TPM, but Alice do not hope Bob knows the identity of Alice. Alice uses one of the AIK she holds to communicate with Bob. So Bob cannot assure which TPM he has communicated with. In theory, any Public Key Infrastructure can solve the problem mentioned. Generate a pair of public/private Key, and then put the private key in the TPM, both Alice and Bob use the public key. Because all the TPM use the same public key, Bob do not know which TPM he has communicated with. Actually, it is not a practical method.

In practice, the TCG gives two solutions for the TPM identity authentication [6], Privacy-CA Solution and Direct Anonymous Attestation (DAA).

a) Privacy-CA Solution

TCG firstly gives a solution which is called Privacy Certificate Authority (privacy-CA) solution. The mainly procession is: TPM has a RSA key pair EK. Privacy-CA knows all the EK of all the TPM. If a TPM Alice wants to communicate

with a verifier Bob, Alice then generates a RSA key pair AIK, and sends the public portion of the AIK and its EK to Privacy-CA. Privacy-CA will check if the EK is valid (check the EK on its TPM list). If it is valid, the Privacy-CA will sign a certificate, and encrypt by EK. Only the Alice who holds the private portion of the EK can decrypt the certificate. Then Alice can send the certificate to Bob to show that she is a trusted platform.

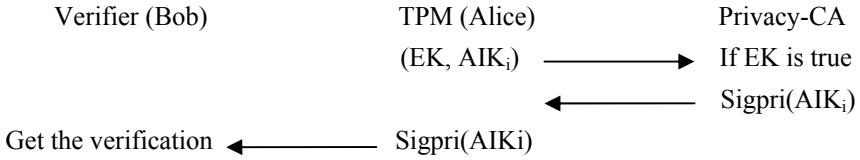

Fig.6. Procession of Privacy-CA solution.

In the Privacy-CA solution model, there are two methods to check the TPM is valid or not. The first one is that if the private portion of an EK from a TPM is blabbed, Privacy-CA will compute the public portion of the EK and delete it from the valid EK list. The second one is that if Privacy-CA receives more than one request from the same EK, Privacy-CA can reject these requests.

According to [7], the creation of PCAS contains several steps. Firstly, TPM host uses TPM_MakeIdentity function to generate a 2048 bits RSA key pair as AIK. At the same moment, there is also a structure of TPM_IDENTITY_CONTENTS produced, which contains the public key of the AIK and some information of the TPM who generate the AIK. The information contains the EK and information about the host (platform). Secondly, TPM uses the private portion of the EK to sign TPM_IDENTITY_CONTENTS. Finally, TPM sends the signature and TPM_IDENTITY_CONTENTS to a Privacy-CA, then encrypts the private portion of AIK by the SRK and save the encrypted message in the memory inside the TPM.

On the other side, Privacy-CA receives TPM_IDENTITY_CONTENTS structure and the signature from TPM. It then firstly verifies the validity of the signature. If the signature is valid, Privacy-CA will produce a certificate based on the public key of the AIK. Secondly, Privacy-CA will generate a random session key, and signs the certificate by this session key. Thirdly, Privacy-CA sign the session key by the given public key of the EK in the received TPM_IDENTITY_CONTENTS. At the same moment, a structure of TPM_SYM_CA_ATTESTATION is produced, which contains the signed session key, the signed certificate, and some related encrypt algorithms. Finally, Privacy-CA sends TPM_SYM_CA_ATTESTATION to TPM.

After TPM receives TPM_SYM_CA_ATTESTATION structure, TPM uses a function called TPM_ActivateIdentity to get the certificate. Firstly TPM uses the private key of the EK to decrypt the signed session key, then uses the session key decrypt the certificate.

b) Direct Anonymous Attestation

After the Privacy-CA solution is given, TCG gives another solution, the Direct Anonymous Attestation (DAA) solution.

In a TPM based trusted platform, 4 entities and two sub-protocols are related to the DAA protocol. 4 entities include the ISSUER, TPM, HOST, and VERIFIER. 2 sub-protocols are JOIN and SIGN protocols. The JOIN protocol is used in the communication between the ISSUER and TPM. Firstly, TPM generates a key DK, and then sends the EK and DK as a pair to the ISSUER request for the signature. The ISSUER verifies EK assure that the EK is valid. Then ISSUER signs the DK as CeDK (a DAA proof) and returns it to the TPM and HOST. After that, TPM can use this CeDK to sign the public portion of an AIK to show this TPM is trusted. The SIGN protocol is used by the VERIFIER to verify the signed message from a TPM. In this procession, TPM and HOST will sign the public key of an AIK and send it to VERIFIER. VERIFIER uses ISSKEY to check this AIK is held by a valid TPM, which means that if the TPM, holds a valid DDA proof CeDK.

Because TPM is a small resource limited chip, some operations are changed to the HOST. Such operations are not related to the key generation and signature sign. The HOST also controls the communication between the TPM and other entity.

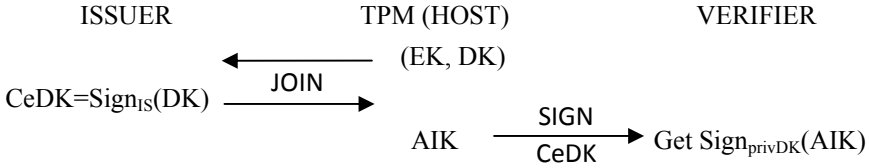

Fig.7. Procession of DAA solution.

c) Comparison between Privacy-CA and DAA

Both Privacy-CA and DAA have their own advantages and disadvantages. In the Privacy-CA solution, Privacy-CA provides public key to TPM, each time the AIK and public key from TPM are different, the verifier cannot distinguish whether they are from the same TPM. On the opposite side, the biggest problem is that TPM need to ask for certificate from Privacy-CA if the verifier needs to communicate with TPM. So the privacy-CA becomes an efficiency limitation to the TPM. The other problem is that who found the Privacy-CA. If Privacy-CA is founded by someone related to TPM, the verifier will be doubtful of the validity of the verification. If Privacy-CA is founded by someone who has relationship with the verifier, it is easy for verifier to get the public portion of EK from TPM. If verifier takes multiple verifications with a TPM, by the help of Privacy-CA verifier will get identity of the TPM.

The main difference between Privacy-CA and DAA is anonymous. In Privacy-CA protocol, the identity of TPM can be known if verifier and the third party (Privacy-CA)

have certain relationships. However, in the DAA protocol, the anonymous of TPM has always been protected. DAA protocol uses more complex algorithm than Privacy-CA. In the verification of DAA, many parameters are randomness. TPM also uses the Zero Knowledge proof in the procedure of getting DAA proof from ISSUER. Such methods ensure ISSUER cannot get any information which can help to recognize identity of the TPM.

Comparing with Privacy-CA, the DAA solution performs more efficiency. TPM only need to ask for a DAA proof once, and then the proof can be used in other procession. In [9], Camenisch and Better give a solution by using both two protocols to ensure both efficiency and anonymous. Also in [10], an experiment shows that DAA need to be improved on its timing management field. On a IBM ThinkPad T41(with Intel Pentium 1.7Ghz CPU, 1GB RAM, Linux), a DAA system is running, then it is shown that in the processing of TPM ask for DAA proof, TPM takes 25% of the total time, the HOST takes 25% and the ISSUER takes 50%. In the processing of communication between TPM and the verifier, TPM takes 8% of the total time, the HOST takes 47% and verifier takes 45%. We can get the result that DAA takes too much time on the ISSUER and VERIFIER parts.

4. Model the PCAS #1.0

The point of this dissertation is making a suitable model of the Privacy-CA Solution protocol in ProVerif. In the original document of TCG, the description of Privacy-CA Solution is general, so it is necessary to choose a suitable strategy to play the most suitable protocol.

On the first step, a simple model is built to analyze the Privacy-CA Solution. It is based on the original document of the TCPA main specification 1.1b [5]. Although this model is simple and easy to be broken, its value is helps to understand the protocol. Then it is easy and convenient to improve the model.

In order to build a model, on the first step we should fix the protocol. In the protocol, the TPM and the platform should be considered as two entities. The Privacy-CA transfers different message to the platform and the TPM chip. Besides this, the platform and the TPM chip also have their own communication, they are different entities. However, for the facility of building a model, I treat them as one entity. In fact, from the point of get certificate for the TPM, they can be treating as one entity.

4.1 The PCAS oracles

In the TPM Main Specification [6], there is an introduction to the Privacy-CA Solution. The protocol is shown as figure 8.

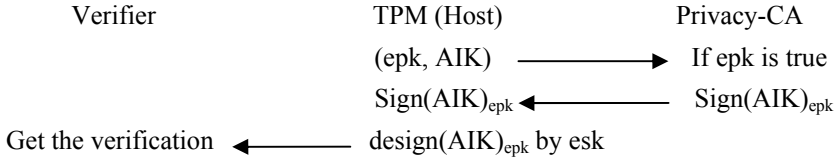

Fig.8. Procession of Privacy-CA solution.

TPM is set on a platform, which is the host. TPM has an direct connection to the Privacy-CA. It send public portion of the Endorsement Key (EK) and public portion of the Attestation Identity Key (AIK) to Privacy-CA. CA holds a list of legal TPM, which contain each public portion of EK of TPMs. It then checks whether the received public EK is in the list. Then the Privacy-CA signs the received public portion of the AIK by its csk, then encrypts this $\{AIK\}_{csk}$ with epk. This $\{AIK\}_{csk}$ is the certificate the Privacy-CA generates for the TPM. Then the Privacy-CA saves this certificate as a proof and sends the encrypted ciphertext to the TPM. TPM gets encrypted ciphertext, then it decrypts the ciphertext by the held esk, and uses $\{AIK\}_{csk}$ as the certificate, show it to the verifier as the proof.

In the protocol, there are two oracles. The first one is TPM oracle.

1) TPM sends epk and AIK;
2) TPM receives $\{\{AIK\}_{csk}\}_{epk}$.
3) TPM decrypt $\{\{AIK\}_{csk}\}_{epk}$ with esk.
4) TPM gets $\{AIK\}_{csk}$ as the certificate.

The second one is Privacy-CA oracle.

1) PCA receives epk and AIK.
2) PCA checks epk. If ok then
3) PCA signs AIK with csk, then encrypts with epk

There are three variables which need to be defined in ProVerif: epk, esk, AIK. The whole protocol can be written as figure 9.

PCAS protocol

Message 1: TPM -> CA : (epk, pk)
Message 2: CA -> TPM : { { pk }csk}epk
Message 3: TPM -> Verifier : { pk }csk

Fig.9. Procession of Privacy-CA solution.

So it is easily to build as a simple model with the version 1.0 in ProVerif.

4.2 Definitions of Model #1.0

Before model PCAS, there are some related work need to be done. There are three variables and functions of signature and Public key cryptography need to be defined. The message channel also should be pointed out before the model start.

Channel is a very useful definition in the ProVerif. A channel is used to transfer message. There are two type channels. The first type is free channel, the message on this channel can be accept by every entity, which means the channel is only used to transfer messages like ciphertext, public key, which can be know by everyone. Also the adversary can create forged message instead of the valid one, and put them in the free channel. The second type is private channel. Message transferred in this channel is security, adversary does not have ability to obtain messages on this kind of channel.

The difination syntax is shown as follow:

```
free c.                                                    (1)
private free cs.

fun puk/1.
fun encrypt/2.                                             (2)
reduc decrypt(encrypt(x, puk(y)),y) = x.

fun sign/2.
reduc checksign(sign(x, y), puk(y)) = x.                   (3)
reduc getmess(sign(x, y)) = x.

fun host/1.
private reduc getkey(host(x)) = x.                         (4)
```

Fig.10. Some definition of PCAS #1.0

Part (1) is a definition of the channels. It means that c is a free channel. Specifically, it means that messages of TPM and Privacy-CA are transferred on channel c. Also, the attacker can obtain, send, modify any message on c. Channel cs is defined as a private channel. It just holds by TPM, and no adversary can control the channel, which means the adversary cannot obtain, send, or modify the message sending on channel cs. Channel cs is used to transfer the secret key pair of EK and AIK of TPM.

In ProVerif, data is defined as terms and term algebras. Part (2) is the definition of public key cryptography. In part (2), puk and encrypt is terms. The syntax begins with "reduc" rewrites rules of the function as needed. In ProVerif, there are many term algebras exists, however, they may not suitable to the aim protocol. ProVerif has the function of rewriting term algebra, so it is very convenient for user to verify protocol. (2) contains one rewrite rule, how to decrypt a message which is signed with a public key.

Part (3) is a definition on signature. It contains two rules: one is for checking sign of a message, the other is for getting message from the given information.

Part (4) defines a table hold by the Privacy-CA. In the PCA oracle, Privacy-CA need to check epk in a list to find whether the TPM is legal. So it is necessary to define a list. The CA has a list epk (TPM), which we represent by the function method getkey to get the value from the list.

4.3 Query property

There are several properties need to be verified. The first one is attacker's ability. Define "attacker" as the adversary. Then

query attacker : cer.

This is used to check whether the adversary can obtain the secret certificate cer. In fact, this cer is the value $\{pk\}_{csk}$. This is the standard secrecy of the PCAS protocol. The core idea for the PCAS protocol is that a Privacy-CA generates a certificate for a TPM. So the most important parameter in the protocol is the certificate. If attacker can hold the certificate, the protocol is not security, because the attacker can modify the certificate or replace the value by others.

Besides ability of the adversary, one-one correspondence also needs to be verified, which is defined as

query evinj: endCA(x, y) ==> evinj: beginCA(x, y).

query evinj: endTPM(x, y) ==> evinj: beginTPM(x, y).

These two queries mean to monitor the event of exchanging messages between TPM and CA. Specifically, it is used to verify if event endCA is always preceded by event beginCA, and event endTPM is always preceded by event beginTPM. Also, it is used to check if every trace contains at least the same numbers of events beginCA and endCA, beginTPM and endTPM.

4.4 Procession of PCAS #1.0

The PCAS model #1.0 contains two oracles and one main process. Two oracles are for the Privacy-CA and the TPM. Although TPM is set on a platform, here I treat the TPM and the platform as one entity, does not consider information flew inside the platform. The one main process is used to control the two oracles. The process for the TPM and the process for the Privacy-CA are with parallel relationship. While verify the model, the TPM and the Privacy-CA send/receive messages between each other in the same procession. The main process is in charge of producing parameters for TPM and make sure two processes run with a parallel relationship. The parallel relationship is played by a function symbol "|".

Figure 11, 12, 13 shows the processCA, the processTPM and the main process respectively. Each process is in the form of "in(x, y)" and "out(x, y)", which means the message output or input from the channel, and with some determine condition. The "in" of one entity is corresponded to another "out" of another entity. The numbers of "in" syntax must equal with the numbers of the "out" syntax.

4.4.1 Process of the Privacy-CA oracle

Figure 11 shows the process of the Privacy-CA oracle.

```
let processCA =
    in(c, (epk1, pk1));
    event beginCA(epk1, pk1);
    let epk = getkey(pk1) in
    if epk=epk1 then
    new csk;
    out(c,(encrypt (sign(pk1, csk), puk(epk))));
    event endCA(epk1, pk1).
```

Fig.11. Process of the CA oracle.

The processCA is beginning with receiving a message (epk1, pk1). Although in the protocol the message contains epk and pk, in the model, the message may be modified by the adversary, so we need to use two parameters instead of original values. Then Privacy-CA will start the event beginCA, which means the Privacy-CA starts the authorization work. After that, Privacy-CA will check the received epk1, make sure that the TPM is legally. Then Privacy-CA signs the AIK pk1 with its csk as the certificate. Finally, there is the event endCA, which means the authorization work is finished.

4.4.2 Process of the TPM oracle

Figure 12 shows the process of TPM oracle.

```
let processTPM =
    in(cs,(epk, pk));
    event beginTPM(epk, pk);
    out(c, (epk, pk));
    in(c, ms1);
    new cer;
    let (=cer) = decrypt(ms1, epk) in
    event endTPM(epk, pk).
```

Fig.12. Process of the TPM oracle.

The processTPM is starting with receiving a message (epk, pk). This message is

generated by the main process. Epk is the public portion of the EK of TPM. Although it is fixed by the TPM, here we assume the main process generates the value for different TPM, which also means the main process produces a new TPM for authorizing. (epk, pk) is transferred on the private channel cs. It means that this message is security and secrecy. The adversary cannot obtain, send, or modify the message. Then there is the event beginTPM, means that the TPM starts the authorization work. Firstly the TPM sends the message (epk, pk) on channel c to the Privacy-CA. Then it receives a message ms1 sending back from the Privacy-CA. In fact, this message is pk signed by epk. Finally, there is the event endTPM, which means the authorization work is finished.

After the TPM gets the message ms1, it will decrypt the message by the private portion of the EK and gets the certificate.

4.4.3 Procession of the main process

Figure 13 shows the procession of the main process.

```
Process
    (!processCA | !processTPM |
    new esk;
        let epk = puk(esk) in
    new sk;
        let pk = puk(sk) in
    out (cs, (epk, pk)))
```

Fig.13. Procession of the main process.

The main process control all the whole model in ProVerif. Here the symbol "|" means the parallel relationship. "!processCA" and "!processCA" mean running the procession of Privacy-CA oracle and TPM oracle. While running the two oracles, a new esk and sk is generated for a TPM. The parameter esk is the private portion of the EK, and sk is the private portion of the AIK. Then the main procession uses function puk to generate the public portion of the EK and AIK, which are shown as epk and sk. After that, the main process uses "out(cs,(epk, pk))" to send the message (epk, pk) to the TPM procession on the private channel cs.

The whole process has another meaning. Two symbol "|" concatenate processCA, processTPM and the key generation procession together. It means that while start a new one of the three, other two will also start for a new round.

4.5 Analysis of result

Figure 14 shows the results of verifying model version 1.0 in ProVerif.

-- Query evinj:endTPM(x_25,y_26) ==> evinj:beginTPM(x_25,y_26)
Completing...
Starting query evinj:endTPM(x_25,y_26) ==> evinj:beginTPM(x_25,y_26)
RESULT evinj:endTPM(x_25,y_26) ==> evinj:beginTPM(x_25,y_26) is true.
-- Query evinj:endCA(x_159,y_160) ==> evinj:beginCA(x_159,y_160)
Completing...
Starting query evinj:endCA(x_159,y_160) ==> evinj:beginCA(x_159,y_160)
goal reachable: begin:beginCA(x_277,host(x_277)), pk1_14 = host(x_277), epk1_13 = x_277, @sid_58 = endsid_278, @occ3_197 = @occ_cst() & attacker:x_277 -> end:endsid_278,endCA(x_277,host(x_277))
RESULT evinj:endCA(x_159,y_160) ==> evinj:beginCA(x_159,y_160) is true.
-- Query not attacker:sign(pk1_287,csk_16[pk1 = v_288,epk1 = v_289,!1 = v_290])
Completing...
Starting query not attacker:sign(pk1_287,csk_16[pk1 = v_288,epk1 = v_289,!1 = v_290])
goal reachable: attacker:v_401 -> attacker:sign(host(v_401),csk_16[pk1 = host(v_401),epk1 = v_401,!1 = v_402])
Abbreviations:
csk_423 = csk_16[pk1 = host(v_421),epk1 = v_421,!1 = v_422]

1. We assume as hypothesis that
attacker:v_421.

2. By 1, the attacker may know v_421.
Using the function host the attacker may obtain host(v_421).
attacker:host(v_421).

3. By 1, the attacker may know v_421.
By 2, the attacker may know host(v_421).
Using the function 2-tuple the attacker may obtain (v_421,host(v_421)).
attacker:(v_421,host(v_421)).

4. The message (v_421,host(v_421)) that the attacker may have by 3 may be received at input {2}.
So the message (encrypt(sign(host(v_421),csk_423),puk(v_421))) may be sent to the attacker at output {7}.
attacker:(encrypt(sign(host(v_421),csk_423),puk(v_421))).

Fig.14. Results of the PCAS model #1.0.

5. By 4, the attacker may know (encrypt(sign(host(v_421),csk_423),puk(v_421))).
Using the 0th inverse of function 1-tuple the attacker may obtain encrypt(sign(host(v_421),csk_423),puk(v_421)).
attacker:encrypt(sign(host(v_421),csk_423),puk(v_421)).

6. By 5, the attacker may know encrypt(sign(host(v_421),csk_423),puk(v_421)).
By 1, the attacker may know v_421.
Using the function decrypt the attacker may obtain sign(host(v_421),csk_423).
attacker:sign(host(v_421),csk_423).

A more detailed output of the traces is available with
param traceDisplay = long.

in(c, (a_424,host(a_424))) at {2} in copy a_425

event(beginCA(a_424,host(a_424))) at {3} in copy a_425

new csk_16 creating csk_16_426 at {6} in copy a_425

out(c, (encrypt(sign(host(a_424),csk_16_426),puk(a_424)))) at {7} in copy a_425

event(endCA(a_424,host(a_424))) at {8} in copy a_425

The attacker has the message sign(host(a_424),csk_16_426).
A trace has been found.
RESULT not attacker:sign(pk1_287,csk_16[pk1 = v_288,epk1 = v_289,!1 = v_290]) is false.

Fig.14. Results of the PCAS model #1.0.

Sentences like "RESULT evinj: endTPM(x_25, y_26) ==> evinj: beginTPM(x_25, y_26) is true." shows the results of the query properties. Here the syntax means that the query property is correct. On the other side, "RESULT not attacker: sign (pk1_287, csk_16[pk1 = v_288,epk1 = v_289,!1 = v_290]) is false." means the query property is failed. The adversary can obtain the property "sign (pk, csk)".

ProVerif then provide specific explanation on how the query failed. It trys to found a trace that makes the query fail. "*A more detailed output of the traces is available with param traceDisplay = long.*" means that the explanation is long, if shows all the result, we need to add a syntax "*param traceDisplay = long.*" to the script.

The first version of PCAS is very simple. It is not a security protocol. The result

shows that it can easily be broken by an adversary. For example, an attacker can easily get the ability:

1) The adversary can have some term p.
2) The adversary can use function puk to generate puk(p).
3) The message encrypt (sign(pk1, csk), epk) may be sent to the adversary.
4) The adversary can use getmess function to get the messageenpk1.
5) The adversary can use his puk(p) instead of epk.
6) The adversary can obtain cer which is sign(pk1, csk).

Although channel cs is secret, the adversary can obtain, change and send message on the free channel c. The adversary can obtain epk and pk on channel c, and use changed ciphertext as he like. So it is not IND-CCA security. Both Privacy-CA and TPM are lacking methods of checking received messages is validity. This point will be improved in version 1.1 of the model.

PCAS model #1.0 is the most basic script of the work. The next two versions are based on this model. Also, this model is not worth using to verify the security of the Privacy-CA Solution protocol, it is used to help to transfer the protocol to a suitable ProVerif model.

5 Model PCAS #1.1

The model PCAS #1.0 is not a successful model. There are several reasons that it is not security. The most important one is that lacking some methods to verify the validity of the received message. Here exists a method for break some cryptographic system called Man-in-the-Middle attack, which means between two entities A and B of a cryptographic system, an adversary pretends A to B and pretends B to A. It holds the valid message and uses forged message to cheat for the secret. Similar to this, model #1.0 is not security. So some new processes need to be considered to prevent forged message. There is a good choice for the model: using random nonce as a proof.

Before fixing the version 1.0, it is necessary to analyse the Privacy-CA Solution protocol again.

5.1 Further analysis of the PCAS protocol

In the model version 1.0, the main problem is that there is no method for either Privacy-CA or TPM to check if the message is sent from a legal TPM or Privacy-CA. Using random nonce is a good choice. Both Privacy-CA and TPM generate random nonce and send the encrypted nonce to the target entity. Then they receive the feedback ciphertext. By decrypting the ciphertext, they can get the returned random nonce. Then by checking the value of the random nonce, both the Privacy-CA and the TPM can know whether the message is from a legal entity.

Figure 15 shows how the protocol implements.

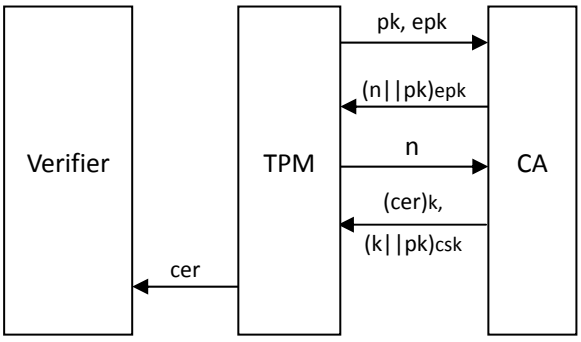

Fig.15. Procession of Privacy-CA solution

Comparing to the previous version, there are two new parameters n and c. N is a random nonce generated by the Privacy-CA. “n||pk” means the concatenation of n and pk. The Privacy-CA signs the value of “n||pk” with epk, then sends the ciphertext to the TPM.

Because only the valid TPM holds the private portion of the EK esk, so only the valid

TPM can decrypt the ciphertext and get n. Then the TPM sends n back to the Privacy-CA. Because n is a random value, there is no possible two same n exist. So by checking the value of n, the Privacy-CA gets the authority that the TPM is legal.

When the Privacy-CA knows the TPM is legal, it then makes the certificate for the TPM. Firstly it signs the public portion of the AIK (pk) with its csk as a certificate. Then the CA saves this certificate. The next step is sending the certificate to the TPM safety. For this target, the CA does similar as its last step, using another random nonce. So the CA generates a k, then encrypts "k||pk" with epk. Finally, the CA encrypts the certificate with this k and sends both to the TPM.

For the TPM, because only the valid TPM holds the private portion of the EK esk, so only the valid TPM can decrypt the ciphertext and get k. Then by decrypting the other ciphertext, the TPM can obtain the certificate and use it to show its validity.

The Protocol also contains two oracles. The first one is the TPM oracle.

1) TPM sends epk and AIK;
2) TPM receives $\{n||AIK\}_{epk}$.
3) TPM decrypt $\{n||AIK\}_{epk}$ with esk.
4) TPM sends n back;
5) TPM receives $\{k||AIK\}_{epk}$ and $\{\{pk\}_{csk}\}_k$

The second one is Privacy-CA oracle.

1) PCA receives epk and AIK.
2) PCA checks epk. If ok then
3) PCA chooses a random nonce n
4) PCA signs n||AIK with epk
5) PCA sends $\{n||AIK\}_{epk}$
6) PCA receives n
7) PCA check n if ok then
8) PCA generates a random nonce k
9) PCA encrypts k||AIK with epk, signs pk with csk, then encrypts $\{\{pk\}_{csk}\}_k$
10) PCA sends $\{k||AIK\}_{epk}$ and $\{\{pk\}_{csk}\}_k$

Figure 16 shows procession of Privacy-CA Solution version 1.1.

```
Message 1: TPM -> CA : (epk, pk)
Message 2: CA -> TPM : { n||pk }epk
Message 3: TPM -> CA : n(fresh)
Message 4: CA -> TPM : { k||pk }epk, {{pk}csk}k
```

Fig.16. Procession of Privacy-CA solution #1.1.

Then the Privacy-CA solution protocol can be modeled to the version 1.1.

5.2 Definitions of Model #1.1

In the second version of PCAS model, several parameters are added to make sure that TPM and Privacy-CA can receive correct message. Also, new term and term algebra need to be defined.

Figure 17 shows the first part of ProVerif syntaxes of PCAS #1.1.

```
free c.                                                    (1)
free ch.
private free cs.

fun puk/1.
fun encrypt/2.                                             (2)
reduc decrypt(encrypt(x, puk(y)),y) = x.

fun sign/2.
reduc checksign(sign(x, y),puk(y)) = x.                    (3)
reduc getmess(sign(x, y)) = x.

fun host/1.
private reduc getkey(host(x)) = x.                         (4)

fun sencrypt/2.
reduc sdecrypt(sencrypt(x, y), y) = x.                     (5)
```

Fig.17. Definitions given by PCAS #1.1.

There are two free channels defined as c and ch. One is for the Privacy-CA, the other is for the TPM. Part (5) is the added term and term algebra. It is used to define a symmetric key cryptographic function. In the protocol model, there are two kinds of encryption. On one side, the public key encryption system is used to sign the concatenation of the random nonce and pk. On the other side, symmetric key

encryption system is used in the encryption to the certificate by the Privacy-CA. This is the main difference between version 1.0 and 1.1 in this part of the model. Basically, they are almost the same.

5.3 Query property

Similar to version 1.0, the first property which need to verify is attacker's ability. Define "attacker" as the adversary. Then there is

query attacker : cer.
query attacker : k.

This is used to check whether the adversary can obtain the secret certificate cer and the key k. In fact, this cer is the value {pk}csk. The value k is used to encrypt the certificate cer. So both the two parameters are important to the security of the protocol. If the adversary obtains the certificate, which means the adversary can modify the certificate, so the protocol is broken. If the adversary can get value of k, because the encrypted ciphertext $\{cer\}_k$ is on the free channel, and the encryption is symmetric key encryption, so the adversary can also obtain the cer.

Besides ability of the adversary, one-one correspondence also needs to be verified, which is defined as

query evinj: endCA(x, y) ==> evinj: beginCA(x, y).
query evinj: endTPM(x, y) ==> evinj: beginTPM(x, y).

This aims to check the event of exchanging messages between TPM and CA is operated once. More specifically, it is used to verify if event endCA is always preceded by event beginCA, event endTPM is always preceded by event beginTPM. Also the same as version 1.0, it is used to check every trace contains at least same numbers of events endCA and endTPM as beginCA and beginTPM.

5.4 Procession of PCAS #1.1

In the second version of the model, there are also two oracles for the Privacy-CA and the TPM. Both of the two oracles is controlled by the main process. This is the same as version 1.0. However, the processions of them are more complex than version 1.0.

By adding two random nonce n and k, there is two more message exchange between the Privacy-CA and the TPM. During this procession, the Privacy-CA and the TPM need to use a space to save the nonce they generated, and they have to make more encrypt and decrypt operation.

Figure 18, 19, 20 show them respectively.

5.4.1 Process of the Privacy-CA oracle

Figure 18 shows the process of Privacy-CA oracle.

```
let processCA =
    in(c, (epk1, pk1));
    event beginCA(epk1, pk1);
    let epk = getkey(pk1) in
    if epk=epk1 then
    new N;
    out(c, (encrypt((N, puk(pk1)), epk)));

    in(c, n);
    if N=n then
    new K;
    new csk;
    out(c, (encrypt((K, puk(pk)), epk))));

    out(ch, (sencrypt(sign(pk, csk), K)));
    event endCA(epk1, pk1).
```

Fig.18. Process of the CA oracle.

First part of the procession is the same as version 1.0. The processCA is beginning with receiving a message (epk1, pk1). Then Privacy-CA will start the event beginCA, which means the Privacy-CA starts the authorization work. By using the getkey function and verifying whether epk is equal to epk1, the CA gets the validity of the TPM. Then it generates a random value N, and encrypts the concatenation of N and pk with epk.

Then the Privacy-CA gets a value n from channel c. The CA will check this n with N it generated on the last step. When they equal, the CA will generate a new random nonce K. CA uses his private key csk sign the pk as a certificate, and use this K to encrypt the certificate. Then the CA sends out ciphertext $\{K||pk\}_{epk}$ on the channel c and $\{\{pk\}_{csk}\}_K$ on the channel ch. Finally, there is the event endCA, which means the authorization work is finished.

5.4.2 Procession of the TPM oracle

Figure 19 shows the process of TPM oracle.

```
let processTPM =
    in(cs,(epk, pk));
    event beginTPM(epk, pk);
    out(c, (epk, pk));
    in(c, ms1);
    let (=(n,pk2)) = decrypt(ms1, epk1) in
    if pk=pk2 then
    out(c, n);

    in(c, ms2);
    new key;
    let (=(key,pk3)) = decrypt(ms2, epk1) in
    in(ch, ms3);
    new cer;
    let (=cer) = sdecrypt(ms3, key) in
    event endTPM(epk, pk).
```

Fig.19. Process of the TPM oracle.

The processTPM is similar but much complex then the version 1.0. The beginning part is the same as version 1.0, with receiving a message (epk, pk) from secret channel cs. Then there is the event beginTPM. The TPM sends the message (epk, pk) on channel c to the Privacy-CA. Then it receives a message ms1 back from the Privacy-CA.

The TPM needs to decrypt the ciphertext with its esk and get the message n||pk2. It will check pk2 with its pk, and make sure n is from a valid Privacy-CA. Then it sends n back. After that, the TPM will receive a ciphertext on channel c and another on channel ch. One of them is the encrypted certificate; the other is the encrypted key for the certificate. The TPM have to firstly decrypt the cipher to obtain the key, then decrypt the other cipher to get the certificate. Here using both the public key cryptographic system and symmetric key cryptographic system.

Finally, there is the event endTPM, which means the authorization work is finished.

5.4.3 Procession of the main process

Figure 20 shows the procession of the main process.

```
Process
    (!processCA | !processTPM |
    new esk;
        let epk = puk(esk) in
    new sk;
        let pk = puk(sk) in
    out (cs, (epk, pk)))
```

Fig.20. Procession of the main process.

The main procession of version 1.1 is the same as version 1.0. It is not necessary to change anything.

5.5 Analysis of result

Figure 21 shows the main part of the results. Firstly, here lists the four query properties:

- query attacker : cer.
- query attacker : k.
- query evinj: endCA(x, y) ==> evinj: beginCA(x, y).
- query evinj: endTPM(x, y) ==> evinj: beginTPM(x, y).

In the real model, sign(pk1, csk) is used to be in place of cer. The reason is that sign(pk1, csk) is the real value of cer, and cer is defined in the procession processTPM, it is not sent out. So attacker cannot obtain this value. However, sign(pk1, csk) is produced by the Privacy-CA, and is sent to the TPM on channel ch, so it may be obtain by the adversary.

"*RESULT not attacker: K_18[n = v_305,pk1 = v_306,epk1 = v_307,!1 = v_308] is true.*" and "*RESULT not attacker: sign(pk1_435,csk_19[n = v_436,pk1 = v_437,epk1 = v_438,!1 = v_439]) is true.*" show that the attacker cannot obtain neither the certificate nor the k.

"*RESULT evinj: endTPM(x_31,y_32) ==> evinj: beginTPM(x_31,y_32) is true.*" and "*RESULT evinj: endCA(x_177,y_178) ==> evinj: beginCA(x_177,y_178) is true.*" show that the one-one correspondence has been verified by the ProVerif.

From the result, we can see that all the four aims passed the verification, which means that the Privacy-CA Solution is a security protocol. However, the model still can be improved. By some researches support [11], version 1.2 of the PCAS model is founded in the next section.

```
-- Query evinj:endTPM(x_31,y_32) ==> evinj:beginTPM(x_31,y_32)
Completing...
nounif attacker:puk(y_162)/-5000
Starting query evinj:endTPM(x_31,y_32) ==> evinj:beginTPM(x_31,y_32)
RESULT evinj:endTPM(x_31,y_32) ==> evinj:beginTPM(x_31,y_32) is true.
-- Query evinj:endCA(x_177,y_178) ==> evinj:beginCA(x_177,y_178)
Completing...
nounif attacker:puk(y_289)/-5000
Starting query evinj:endCA(x_177,y_178) ==> evinj:beginCA(x_177,y_178)
RESULT evinj:endCA(x_177,y_178) ==> evinj:beginCA(x_177,y_178) is true.
-- Query not attacker:K_18[n = v_305,pk1 = v_306,epk1 = v_307,!1 = v_308]
Completing...
nounif attacker:puk(y_418)/-5000
Starting query not attacker:K_18[n = v_305,pk1 = v_306,epk1 = v_307,!1 =
v_308]
RESULT not attacker:K_18[n = v_305,pk1 = v_306,epk1 = v_307,!1 = v_308] is
true.
-- Query not attacker:sign(pk1_435,csk_19[n = v_436,pk1 = v_437,epk1 =
v_438,!1 = v_439])
Completing...
nounif attacker:puk(y_549)/-5000
Starting query not attacker:sign(pk1_435,csk_19[n = v_436,pk1 = v_437,epk1 =
v_438,!1 = v_439])
RESULT not attacker:sign(pk1_435,csk_19[n = v_436,pk1 = v_437,epk1 =
v_438,!1 = v_439]) is true.
```

Fig.21. Results of the PCAS model #1.1.

6 Model PCAS #1.2

Some more specific details are listed in the model 1.2. They are all conditions for judging the security of the Privacy-CA Solution protocol. In next section, the theory support of the modification will be discussed.

6.1 Conditions for security evaluation

Firstly, denote an adversary as attacker. Then for a CA c_i, (cpk_i, csk_i) is the public key pair of c_i. Also, for a TPM t_j, (epk_j, $eskj_i$) is the public key pair of t_j. The adversary can hold a list of the public keys of legal CAs attacker(cpk_i) and a list of the public keys of legal TPMs attacker(epk_j). According to Chen and Warinschi [11], the core idea of this theory is using a list to save four important values, which is (epk_i, cpk_j, pk, cer). Define regList as a list to save several group of (epk_i, cpk_j, pk, cer). Then there are the conditions:

Attacker wins:

- If $\{\{cer\}_{pk}\}_{cpkj}$ is corrupt
- If csk_j is corrupt

Attacker loses:

- If $(*, cpk_j, pk, cer) \notin RegList$
- If $(epk_i, cpk_j, pk, cer) \in RegList$ and $ValidKey(epk_i, pk) = 0$

There are three conditions to control the result of the experiment. When the result of the experiment is 0, then the adversary wins. When the result of the experiment is 1, the adversary loses. Firstly, verify cer and csk_j. If is csk_j corrupt, then return 0. If cer is not a valid signature certificate on pk under cpk_j, also return 0. Secondly, check the output (epk_i, cpk_j, pk, cer). If (epk_i, cpk_j, pk, cer) (the epk on the first position is negligible) does not occur in the regList then return 1. Thirdly, for some TPM public keys, if (epk_i, cpk_j, pk, cer) does not occur in the regList and the key pk does not fit for these TPMs, there is $ValidKey(esk_i, pk) = 0$. Then the experiment also returns 1. If all the conditions are satisfied, the experiment returns 0.

Then Chen and Warinschi [11] prove the protocol is security under the conditions. Firstly, define the advantage of any attacker as the probability 1 outputted by the experiment. Because there may be plenty of different TPMs, but there must be very few valid CAs, so there is a limitation on the number of entities who join the experiment and the number of the sessions of each entity. It means that the adversary cannot try with a same (cpk, csk) or (epk, esk) so many times. According to this point,

the chance of adversary win is a negligible function of the security parameter. So we said that the protocol is secure under the model assuming standard security notions for the underlying asymmetric encryption and signature schemes. However, the model is secure but weak, because the experiment just considers the situation of corrupt oracle of the CA, but does not consider the situation of the corrupt oracle of the TPM.

When consider the experiment to the model, it is necessary to add another oracle and another entity, which is used to save the regList and make the judgments on the parameters' checking. Beside this, there are also some added processes to the CA oracle and the TPM oracle.

The first one is the TPM oracle.

1) TPM sends epk and AIK;
2) TPM receives $\{n||AIK\}_{epk}$.
3) TPM decrypt $\{n||AIK\}_{epk}$ with esk.
4) TPM sends n back.
5) TPM receives $\{k||AIK\}_{epk}$ and $\{\{pk\}_{csk}\}_{k.}$
6) TPM sends {epk, cpk, pk, cer}.

The second one is Privacy-CA oracle.

1) PCA receives epk and AIK.
2) PCA checks epk. If ok then
3) PCA chooses a random nonce n
4) PCA signs n||AIK with epk
5) PCA sends $\{n||AIK\}_{epk}$
6) PCA receives n
7) PCA check n if ok then
8) PCA generates a random nonce k
9) PCA encrypts k||AIK with epk, signs pk with csk, then encrypts $\{\{pk\}_{csk}\}_{k}$
10) PCA sends $\{k||AIK\}_{epk}$ and $\{\{pk\}_{csk}\}_{k}$
11) PCA sends {epk, cpk, pk, cer}.

And here is the third oracle.

1) Third receives {epk, cpk, pk, cer} from the CA then
2) Third saves {epk, cpk, pk, cer} in regList.
3) Third receives{epk1, cpk1, pk1, cer1} from the TPM then
4) Third checks {epk1, cpk1, pk1, cer1} with the regList.

Figure 22 shows procession of Privacy-CA Solution version 1.2.

```
Message 1: TPM -> CA : (epk, pk)
Message 2: CA -> TPM : { N||pk }epk
Message 3: TPM -> CA : N(fresh)
Message 4: CA -> TPM : { k||pk }epk, {{pk}csk}k
Message 5: CA -> Third : {epk, cpk, pk, {{pk}csk}k}
message 6: TPM -> Third : {epk, cpk, pk, {{pk}csk}k}
```

Fig.22. Procession of Privacy-CA Solution #1.2.

6.2 Definitions of Model #1.2

Figure 23 shows the first part of ProVerif syntaxes of PCAS #1.2.

```
free c. free ch.
private free cct.
private free ctt.                                              (1)
private free cs.
private free cc.

fun pk/1.
fun encrypt/2.                                                 (2)
reduc decrypt(encrypt(x, pk(y)),y) = x.

fun sign/2.
reduc checksign(sign(x, y),pk(y)) = x.                         (3)
reduc getmess(sign(x, y)) = x.

fun host/1.
private reduc getkey(host(x)) = x.                             (4)

fun sencrypt/2.
reduc sdecrypt(sencrypt(x, y), y) = x.                         (5)

type bset.
fun consset(bitstring, bset): bset[data].
const emptyset: bset[data].
pred mem(bitstring, bset).
clauses
      forall x:bitstring, y:bset; mem(x, consset(x, y));       (6)
      forall x:bitstring, y:bset; z:bitstring;
            mem(x, y)==> mem(x, consset(z, y));
```

Fig.23. Definitions given by PCAS #1.2.

Here I defined six channels. Two free channels is used to transfer message between the Privacy-CA and the TPM. Four private channels are use to transfer message must not obtain by the adversary. Two of them are used by the main process generates (epk, esk) for the TPM and (cpk, csk) for the Pricacy-CA. The other two are used to transfer data from the Privacy-CA and the TPM to the third entity. The third entity save data from CA to a regList, then determine data from TPM with the regList.

The added syntax part (6) is used to define a list for the third entity to save the four data from the Privacy-CA. It is a complex function. Consset is a function, put a string at the head of a list. Mem is a function used to check the relationship between two parameters. For example, there are x, y, z three string. The aim is that put the three string in a list n. Then denote a empty list n. There is

```
a = consset (z, n);
b = consset (y, a);
c = consset (z, b);
```

Then c is the list which hold three parameters x, y, z in the form of (x||y||z) (|| means concatenation of two values). Then we use mem (z, c) to check if z is at the head of the list c or in the tail of the list.

6.3 Query property

In this part, the query properties are more complex than the previous versions. Define "attacker" as the adversary. Then

```
query attacker : cer.
query attacker : k.
query evinj: endCA(x, y) ==> evinj: beginCA(x, y).
query evinj: endTPM(x, y) ==> evinj: beginTPM(x, y).
```

This four aims are still very important properties need to check. Besides this, there are two conditions:

- If $(*, \mathrm{cpk_j}, \mathrm{pk}, \mathrm{cer}) \notin \mathrm{RegList}$
- If $(\mathrm{epk_i}, \mathrm{cpk_j}, \mathrm{pk}, \mathrm{cer}) \in \mathrm{RegList}$ and $\mathrm{ValidKey}(\mathrm{epk_i}, \mathrm{pk}) = 0$

Another two query properties are added corresponding to these conditions, by the form of:

```
query attacker : con1.
query attacker : con2.
```

The parameter con1 represent the first condition, con2 represent the second condition. Con1 and con2 are set to perform the results of the two conditions above. If the result is accordance with any of the conditions, then using the form of

new con1;
if $(*, cpk_j, pk, cer) \notin RegList$ then
out (c, con1);
new con2;
if $(epk_i, cpk_j, pk, cer) \in RegList$ and $ValidKey(epk_i, pk) = 0$ then
out (c, con2);

Because c is a free channel, so when the results satisfy the condition 1 or condition 2, the parameter con1 or con2 will be sent on the channel c, and the adversary will obtain the parameter. It means the result of the query property will be false. Because the judgment is happened in the oracle of the third entity, the script above is in the process of the third entity.

6.4 Procession of PCAS #1.2

There is almost no change to the Privacy-CA and the TPM, they are almost the same as the version 1.1. Figure 24, 25 show them respectively. Figure 26 and 27 show the processions of the Third and main process.

6.4.1 Process of the Privacy-CA oracle

Figure 24 shows process of the Privacy-CA oracle.

```
let processCA =
    in(c, (epk1, pk1));
    in(cc,(cpk, csk));
    event beginCA(epk1, pk1);
    let epk = getkey(pk1) in
    if epk=epk1 then
    new N;
    out(c, (encrypt((N, puk(pk1)), epk)));

    in(c, n);
    if N=n then
    new K;
    out(c, (encrypt((K, puk(pk1)), epk)));
    out(ch, (sencrypt(sign(pk1, csk), K)));

    out(cct, (epk, cpk, pk, sign(pk1, csk)));
    event endCA(epk1, pk1).
```

Fig.24. Process of the CA oracle.

There are two added syntaxes. The first one in the final part of a CA oracle, the CA sends a group of values (epk, cpk, pk, sign(pk, csk)) on a private channel cct. Channel cct is only used to transfer data between the CA and the third entity. The second one is in the beginning part of the CA oracle, there is a (cpk, csk) input from a private channel cc. The public key pair (cpk, csk) is generated by the main process.

6.4.2 Process of the TPM oracle

Figure 24 shows the process of the TPM oracl.

```
let processTPM =
    in(cs,(epk, pk));
    event beginTPM(epk, pk);
    out(c, (epk, pk));
    in(c, ms1);
    let (=(n,pk2)) = decrypt(ms1, epk1) in
    if pk=pk2 then
    out(c, n);

    in(c, ms2);
    new key;
    let (=(key,pk3)) = decrypt(ms2, epk1) in
    in(ch, ms3);
    new cer;
    let (=cer) = sdecrypt(ms1, key) in

    out(ctt, (epk, cpk, pk2, cer));
    event endTPM(epk, pk).
```

Fig.25. Process of the TPM oracle.

The meaning of the only added syntax is that, in the final part of a CA oracle, the CA sends a group of values (epk, cpk, pk, cer) on a private channel ctt. Channel ctt is only used to transfer data between the TPM and the third entity.

6.4.3 Process of the Third oracle

Figure 26 shows part of the scripts of the procession of the Third oracle.

```
let processThird =
    in(cct, (epk, cpk, pk, sign(pk, csk)));
    in(ctt, (epk1, cpk1, pk2, cer));
    let x = consset(sign(pk, csk), emptyset) in
    let y = consset(pk, x) in
    let z = consset(cpk, y) in
    let w = consset(epk, z) in

    if mem(cer, x)&&mem(pk2, y)&&mem(cpk1, z) then
    if mem(cer, x)&&mem(pk2, y)&&mem(cpk1, z)&&mem(epk1, z)
    ……
```

Fig.26. Procession of the Third oracle.

6.4.4 Procession of the main process

```
Process
    (!processCA | !processTPM | !processThird |
    new esk;
        let epk = puk (esk) in
    new sk;
        let pk = puk (sk) in
    out (cs, (epk, pk));
    new esk;
        let epk = puk (esk) in
    out (cc, (cpk, csk)))
```

Fig.27. Procession of the main process.

In version 1.2, the main process also generates a public key pair of (cpk, csk), and sends them on the private channel cc to the CA oracle.

6.5 Analysis of result

The analysis shows that the Privacy-CA Solution protocol is security. In model #1.2, the results show that four of the six aims are security. They are

query attacker : cer.
query attacker : k.
query evinj: endCA(x, y) ==> evinj: beginCA(x, y).
query evinj: endTPM(x, y) ==> evinj: beginTPM(x, y).

All the four queries are the same as four in the model #1.1. This has been proof as the result of figure 21. The trace here is almost the same as model #1.1. However, the other two queries are more important in model #1.2.

The two query properties

query attacker : con1;
query attacker : con2;

for the two conditions

- If $(*, \mathrm{cpk_j}, \mathrm{pk}, \mathrm{cer}) \notin \mathrm{RegList}$
- If $(\mathrm{epk_i}, \mathrm{cpk_j}, \mathrm{pk}, \mathrm{cer}) \in \mathrm{RegList}$ and $\mathrm{ValidKey}(\mathrm{epk_i}, \mathrm{pk}) = 0$

cannot be verified.

There may be many reasons. Firstly, model a regList in ProVerif is very complex and difficult. Secondly, it is very hard to perform the conditions correctly in ProVerif. Thirdly, ProVerif does not support such syntax. By analyzing the problem occurred in the verification, I found that the reason why it did not go though the verifier. It is because I did not perform the two conditions correctly.

Although in model #1.2, there are targets not achieved, model #1.1 has shown that the Privacy-CA Solution protocol is security.

7 Conclusion

In the first part, this dissertation introduces background information of TCG and TPM, and shows specific principle of the Privacy-CA Solution protocol. Then the dissertation gives some introduction on the automatic cryptographic verification tool ProVerif.

In the second part, this dissertation focuses on the PCAS protocol, uses pi logic clauses build three versions of the PCAS protocol. The later version is the improvement of the previous one. Each version of the model has been introduced specific. Each model is verified by the ProVerif. The dissertation gives the results and analysis them separately.

Although many works have been done, the project is not finished. In the third version of the model, there are two queries cannot be verified. This problem needs to be solved in the further work.

In the TPM area, there are many researches focus on the Direct Anonymous Attestation (DAA) solution. ProVerif also used widely on verification of cryptographic protocol. However, there is less research on verifying the PCAS protocol with protocol. This dissertation gives examples on modeling the PCAS protocol in ProVerif, which will help to study and improve the PCAS protocol.

8 Reference

[1] D.M. Rousseau, S.B. Sitkin, R.S. Burt, and C. Camerer, Not So Different After All: A Cross-discipline View of Trust, Academy of Management Review, 23, 1998, p. 395.

[2] ISO/IEC. Information technology–Open Systems Interconnection-Evaluation criteria for information technology, 1999. Standard ISO/IEC 15408.

[3] R. Anderson, M. Bond, J. Clulow, S. Skorobogatov, Cryptographic Processors-a survey, IEEE Proceedings, Volume: 94, Issue: 2, page: 357-369, Feb. 2006

[4] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. Smith and S. Weingart, Building the IBM 4758 Secure Coprocessor, IEEE Computer, Oct. 2001

[5] Trusted Computing Group, TCPA Main Specification Version 1.1b, Available at www.trustedcomputinggroup.org, 2011

[6] Trusted Computing Group, TPM Main Specification Level 2 Version 1.2, Available at www.trustedcomputinggroup.org, 2011

[7] Trusted Computing Group, TPM Main Part 1 Design Principles Specification Version 1.2 Revision 116, Available at www.trustedcomputinggroup.org, 2011

[8] Eusebius, TPM English, Available at:
http://commons.wikimedia.org/wiki/File:TPM_english.svg, 2011

[9] Camenisch, J. Better. Privacy for Trusted Computing Platforms, Computer Security ESORICS, 9th European Symposium on Research in Computer Security. Page: 73-88, 2004.

[10] Brickell E, Camenisch J, Chen L. Direct Anonymous Attestation, Proceedings of the 11th ACM Conference on Computer and Communications Security. Page: 132-145, ACM Press, 2004.

[11] L. Chen, B. Warinschi, "Security of the TCG Privacy-CA Solution," Embedded and Ubiquitous Computing, IEEE/IFIP International Conference on, pp. 609-616, 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2010

[12] J. Millen, S. Clark, S. Freedman. The Interrogator: protocol security analysis. IEEE Trans. Software Engineering, SE-13(2), Feb 1987

[13] B. Blanchet. An Efficient Protocol Verifier Based on Prolog Rules. In: Proceedings of 14th IEEE computer security Foundations Work shop, IEEE computer Society Press, pages: 82-96, 2001

[14] D. Dolev, A.C. Yao. On the Security of Public Key Protocols. IEEE Transactions on Information Theory, pages: 198-208, 29(2), 1983

[15] M. Burrows, M. Abadi, R.Needham. A Logic of Authentication. ACM Transactions in Computer System, pages: 18-36, 8(1), 1990.

[16] B. Blanchet, B. Smyth. ProVerif 1.85: Automatic Cryptographic Protocol Verifier User Manual and Tutorial. Available at: http://www.proverif.ens.fr/, 2011

[17] Alfred Horn. On sentences which are true of direct unions of algebras, Journal of Symbolic Logic, 16, 14–21.

[18] MPRI Lecture Notes, Available at: http://www.proverif.ens.fr/, 2011

[19] M. Abadi, C. Fournet. Mobile Values, New Names and Secure Communication. In 28th ACM Symposium on Principles of Programming Languages (POPL'01), pages 104-115, Jan. 2001.

[20] M. Peters, P. Rogaar. A review of ProVerif as an automatic security protocol verifier, Available at: http://agoraproject.eu/app/webroot/papers/, 2011

[21] Chris W. Hoffmeister. An Evaluation Methodology for Protocol Analysis Systems, Available at: http://edocs.nps.edu/npspubs/scholarly/theses/2007/Mar/07Mar_Hoffmeister.pdf, 2011

[22] J. Thayer, J. Herzog, J. Guttman. Strand Spaces: Proving Security Protocols Correct, Journal of Computer Security, vol. 7, no. 2-3, pages: 191-230, Jan. 1999

[23] R. Kusters, T. Truderung. Reducing protocol analysis with XOR to the XOR-free case in the Horn theory based approach. In CCS '08: Proceedings of the 15th ACM conference on Computer and communications security, pages: 129–138, New York, 2008. .

[24] R. Kusters, T. Truderung. Using ProVerif to Analyse Protocols with Diffie-Hellman Exponentiation. Computer Security Foundations Symposium, IEEE, 0:157–171, 2009.

[25] P. England, B. Lampson, J. Manferdelli, B. Willman, IEEE Computer, Vol. 36, No. 7. (2003), pp. 55-62, 2003

[26] Microsoft Corporation. Microsoft Palladium: A Business Overview, Available at: http://www.microsoft.com/australia/resources/palladium+white+paper+public.pdf, 2011

[27] B. Brian: Guide to Trusted Computing, Available at: http://www.wwpi.com/index.php?option=com_content&view=article&id=4363:guide-to-trusted-computing&catid=99:cover-story&Itemid=2701018, 2011

[28] B. Blanchet. Using Horn clauses for analyzing security protocols. In Veronique Cortier and

Steve Kremer, editor, Formal Models and Techniques for Analyzing Security Protocols, chapter 5. IOS Press, 2010. Available at: http://www.di.ens.fr/~blanchet/publications/BlanchetBook09.html.

Appendix

1. File PACS model #1.0

```
(************************************************************
 *                                                          *
 *           Cryptographic protocol verifier                *
 *                                                          *
 *                                                          *
 ************************************************************)

(*

      This program is free software; you can redistribute it and/or modify
      it under the terms of the GNU General Public License as published by
      the Free Software Foundation; either version 2 of the License, or
      (at your option) any later version.

*)
free c.
free ch.
private free cs.
(*

PCAS protocol #1.0

Message 1: TPM -> CA : (epk, pk)
Message 2: CA -> TPM : { { pk }csk}epk
Message 3: TPM -> Verifier : { pk }csk

*)

(* Public key cryptography *)

fun puk/1.
fun encrypt/2.
reduc decrypt(encrypt(x,puk(y)),y) = x.

(* synmetric key cryptography *)

fun sencrypt/2.
reduc sdecrypt(sencrypt(x,y),y) = x.
```

```
(* cpks and epks table.
     The CA has a table (epk(TPM), cpk), which we
     represent by the function getkey. *)

fun host/1.
private reduc getkey(host(x)) = x.

(* Signatures *)

fun sign/2.
reduc checksign(sign(x,y),puk(y)) = x.
reduc getmess(sign(x,y)) = x.

(* Secrecy assumptions *)
query attacker : sign(pk1, csk).

query evinj: endCA(x, y) ==> evinj: beginCA(x, y).
query evinj: endTPM(x, y) ==> evinj: beginTPM(x, y).


let processTPM =

      in(cs,(epk, pk));
      event beginTPM(epk, pk);
      out(c, (epk, pk));
      in(c, ms1);
      new cer;
      let (=cer) = decrypt(ms1, epk) in
      event endTPM(epk, pk).


let processCA =

      in(c, (epk1, pk1));
      event beginCA(epk1, pk1);
      let epk = getkey(pk1) in
      if epk=epk1 then
      new csk;
      out(c,(encrypt (sign(pk1, csk), puk(epk))));
      event endCA(epk1, pk1).

process
      (!processCA | !processTPM |
            new esk;
```

```
        let epk = puk(esk) in
        new sk;
        let pk = puk(sk) in
        out (cs, (epk, pk)))
```

2. File PACS model #1.1

```
(************************************************************
 *                                                          *
 *           Cryptographic protocol verifier                *
 *                                                          *
 *                                                          *
 ************************************************************)

(*

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

*)
free c.
free ch.
private free cs.
(*

PCAS protocol #1.1

Message 1: TPM -> CA : (epk, pk)
Message 2: CA -> TPM : { N||pk }epk
Message 3: TPM -> CA : N(fresh)
Message 4: CA -> TPM : { k||pk }epk, {{pk}csk}k



*)

(* Public key cryptography *)

fun puk/1.
fun encrypt/2.
reduc decrypt(encrypt(x,puk(y)),y) = x.
```

```
(* synmetric key cryptography *)

fun sencrypt/2.
reduc sdecrypt(sencrypt(x,y),y) = x.

(* cpks and epks table.
     The CA has a table (epk(TPM), cpk), which we
     represent by the function getkey. *)

fun host/1.
private reduc getkey(host(x)) = x.

(* Signatures *)

fun sign/2.
reduc checksign(sign(x,y),puk(y)) = x.
reduc getmess(sign(x,y)) = x.

(* Secrecy assumptions *)
query attacker : sign(pk1, csk).
query attacker : K.

query evinj: endCA(x, y) ==> evinj: beginCA(x, y).
query evinj: endTPM(x, y) ==> evinj: beginTPM(x, y).


let processTPM =

      in(cs,(epk, pk));
      event beginTPM(epk, pk);
      out(c, (epk, pk));
      in(c, ms1);
      let (=(N1,pk2)) = decrypt(ms1, epk1) in
      if pk=pk2 then
      out(c, N1);

      in(c, ms2);
      new key;
      let (=(key,pk3)) = decrypt(ms2, epk1) in
   in(ch, ms3);
      new cer;
      let (=cer) = sdecrypt(ms3, key) in
      event endTPM(epk, pk).
```

```
let processCA =

    in(c, (epk1, pk1));
    event beginCA(epk1, pk1);
    let epk = getkey(pk1) in
    if epk=epk1 then
    new N;
    out(c, (encrypt((N, puk(pk1)), epk)));

    in(c, n);
    if N=N1 then
    new K;
    new csk;
    out(c, (encrypt((K, puk(pk1)), epk)));
    out(ch, (sencrypt(sign(pk1, csk), K)));
    event endCA(epk1, pk1).


process
    (!processCA | !processTPM |
        new esk;
        let epk = puk(esk) in
        new sk;
        let pk = puk(sk) in
        out (cs, (epk, pk)))
```

3. File PACS model #1.2

```
(************************************************************
 *                                                          *
 *           Cryptographic protocol verifier                *
 *                                                          *
 *                                                          *
 ************************************************************)

(*

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.
```

```
*)
free c.
free ch.
private free cct.
private free ctt.
private free cs.
private free cc.

(*

PCAS protocol #1.2

Message 1: TPM -> CA : (epk, pk)
Message 2: CA -> TPM : { N||pk }epk
Message 3: TPM -> CA : N(fresh)
Message 4: CA -> TPM : { k||pk }epk, {{pk}csk}k
Message 5: CA -> Third : {epk, cpk, pk, {{pk}csk}k}
message 6: TPM -> Third : {epk, cpk, pk, {{pk}csk}k}

*)

(* Public key cryptography *)

fun puk/1.
fun encrypt/2.
reduc decrypt(encrypt(x,puk(y)),y) = x.

(* synmetric key cryptography *)

fun sencrypt/2.
reduc sdecrypt(sencrypt(x,y),y) = x.

(* cpks and epks table.
     The CA has a table (epk(TPM), cpk), which we
     represent by the function getkey. *)

fun host/1.
private reduc getkey(host(x)) = x.

(* Signatures *)

fun sign/2.
reduc checksign(sign(x,y),puk(y)) = x.
```

```
reduc getmess(sign(x,y)) = x.

(* Secrecy assumptions *)

query attacker : sign(pk1, csk).
query attacker : K.
query attacker : con1.
query attacker : con2.

query evinj: endCA(x, y) ==> evinj: beginCA(x, y).
query evinj: endTPM(x, y) ==> evinj: beginTPM(x, y).

let processTPM =

      in(cs,(epk, pk));
      event beginTPM(epk, pk);
      out(c, (epk, pk));
      in(c, ms1);
      let (=(n,pk2)) = decrypt(ms1, epk1) in
      if pk=pk2 then
      out(c, n);

      in(c, ms2);
      new key;
      let (=(key,pk3)) = decrypt(ms2, epk1) in
   in(ch, ms3);
      new cer;
      let (=cer) = sdecrypt(ms1, key) in

      out(ctt, (epk, cpk, pk3, cer));
      event endTPM(epk, pk).

let processCA =

      in(c, (epk1, pk1));
      in(cc,(cpk, csk));
      event beginCA(epk1, pk1);
      let epk = getkey(pk1) in
      if epk=epk1 then
      new N;
      out(c, (encrypt((N, puk(pk1)), epk)));

      in(c, n);
      if N=N1 then
```

```
        new K;
        out(c, (encrypt((K, puk(pk1)), epk)));
        out(ch, (sencrypt(sign(pk1, csk), K)));

        out(cct, (epk, cpk, pk, sign(pk1, csk)));
        event endCA(epk1, pk1).

let processThird =
        in(cct, (epk, cpk, pk1, sign(pk1, csk)));
        in(ctt, (epk, cpk, pk3, cer));
        let x = consset(sign(pk1, csk), emptyset) in
        let y = consset(pk1, x) in
        let z = consset(cpk, y) in
        let w = consset(epk, z) in

(*      if mem(decrypt(b1,K), x)&&mem(pk3, y)&&mem(cpk1, z) then
        if mem(decrypt(b1,K), x)&&mem(pk3, y)&&mem(cpk1, z)&&mem(epk1, z) then    *)

        new con1;
        new con2;
        if pk1=pk3 then
        out(cc, con1) else
        out(c, con1);
        if cer=sign(pk1, csk) then
        out(cc, con2) else
        out(c, con2).

process
        (!processCA | !processTPM | !processThird |
                new esk;
                        let epk = puk (esk) in
                new sk;
                        let pk = puk (sk) in
                out (cs, (epk, pk));
                new esk;
                        let epk = puk (esk) in
                out (cc, (cpk, csk)))
```