PAPER

A New Cloud Architecture of Virtual Trusted Platform Modules

Dongxi LIU[†a)], ***Member***, **Jack LEE**[††*], **Julian JANG**[†], **Surya NEPAL**[†], ***and*** **John ZIC**[†], ***Nonmembers***

SUMMARY We propose and implement a cloud architecture of virtual Trusted Platform Modules (TPMs) to improve the usability of TPMs. In this architecture, virtual TPMs can be obtained from the TPM cloud on demand. Hence, the TPM functionality is available for applications that do not have physical TPMs in their local platforms. Moreover, the TPM cloud allows users to access their keys and data in the same virtual TPM even if they move to untrusted platforms. The TPM cloud is easy to access for applications in different languages since cloud computing delivers services in standard protocols. The functionality of the TPM cloud is demonstrated by applying it to implement the Needham-Schroeder public-key protocol for web authentications, such that the strong security provided by TPMs is integrated into high level applications. The chain of trust based on the TPM cloud is discussed and the security properties of the virtual TPMs in the cloud is analyzed.

key words: *TPM, cloud, virtualization, trust service*

1. Introduction

Trusted computing is a category of technology developed by the Trusted Computing Group (TCG) [1] to facilitate the development of trusted systems. The standards of trusted computing specify the hardware and software components needed to build trusted systems. In particular, the hardware component is a chip called Trusted Platform Module (TPM), which is used as the hardware root in system trust.

The trust of TPM lies in its capabilities of secure key management (i.e., key generation, storage and use), and secure storage and reporting of platform configuration measurements. A TPM can generate RSA key pairs. The private RSA keys are always used within the TPM, never leaving it without encryption. A TPM stores measurements in a set of Platform Configuration Registers (PCRs), which are physically protected. The platform measurements stored in PCRs are signed with a key in the TPM when reporting the integrity of a system for remote attestation [2].

TPMs are currently provided by embedding them into computer motherboards. For an application to benefit from the TPM functionality, there are several requirements. First, the computer running the application must have a TPM. Second, the supporting softwares (e.g., TPM drivers and TCG Software Stack (TSS) [3]) must be installed for the application to access the TPM. Third, the application users must be willing to use the TPM since the computer being used might not be owned by them.

However, the above requirements are sometimes not easy to satisfy, thus making TPMs not so usable and hampering the wide acceptance of TPMs by secure applications [4]. A lot of computers (new or legacy) are not manufactured with TPMs. Most IBM blade servers [5] do not contain TPMs, nor do many resource-constrained embedded systems due to the size and cost overheads of a separate TPM [6]. There is the software-based TPM emulator [7]. However, it is only developed for Unix and not as secure as physical TPMs. For the requirement of supporting software, the current TSS is mainly implemented in C language such as TrouSerS [8] or in Java such as jTSS [9]. Hence, it is hard for the applications developed in other languages to access the TPM functionality. For example, Javascript code in web applications cannot access the TPM functionality easily. At last, application users may be reluctant to use TPMs in the computers they use but do not own (e.g., a public computer in an Internet bar) since the TPMs there may not have proper keys and PCRs.

In this paper, we propose a cloud architecture of virtual TPMs (the TPM cloud), embodying the concept of infrastructure as a service in cloud computing [10]. Our motivation is to improve the usability of TPMs. A usable security mechanism is more likely to be widely and effectively used [4], [11], [12]. The TPM cloud will help applications to benefit from the strong security provided by TPMs.

From the TPM cloud, users can apply for their own TPM instances (or virtual TPMs) on demand, as exemplified in Fig. 1. The TPM cloud contains a cluster of physical TPMs and virtualizes them to provide TPM instances for a large number of users. Consequently, applications can access the TPM functionality irrespective of the availability of TPMs in the underlying computers. Moreover, even if users run their applications in different computers, they still can access the same TPM instance since it is provided as a service in the cloud. Hence, there is no need to migrate private keys among computers, avoiding inconvenience and potential security problems. Cloud computing advocates the de-

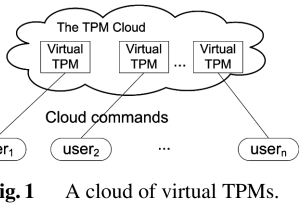


Fig. 1 A cloud of virtual TPMs.

Manuscript received July 13, 2011.
Manuscript revised December 7, 2011.

[†]The authors are with the CSIRO ICT Centre, Australia.
[††]The author was with Sydney University, Australia.
[*]Presently, with the Defense Signals Directorate, Australia.
a) E-mail: dongxi.liu@csiro.au

DOI: 10.1587/transinf.E95.D.1577

livery of services in standard protocols, such as Simple Object Access Protocol (SOAP), so there is no interoperability problem between the TPM cloud and its applications. Our contributions of this work are summarized as follows:

- We propose an architecture of the TPM cloud (Sect. 2). The TPM cloud facilitates the usability of TPMs since it addresses the requirement problems as discussed above. The TPM functionalities in the cloud are provided as Web service operations, so they can be easily integrated into application design and implementation. Easy access to TPM functionalities is also realized as a key step to promote the wide acceptance of TPMs and regarded as a future working direction in the report [4][†].
- We formalize the functionality of the TPM cloud (Sect. 3). The formal functionality specification lays a foundation for analyzing and implementing the TPM cloud. To support a large number of users, the TPM cloud virtualizes a cluster of physical TPMs. Based on the formalization, we suggest some improvement to the TPM specification, which can make TPMs more flexible to be used in various contexts.
- We implement a prototype of the TPM cloud and apply it to implement the Needham-Schroeder public-key protocol [14] for Web authentication (Sect. 4). In the implementation of this protocol, both the user and the Web server depend on the TPM cloud to decrypt encrypted messages. That is, their private keys are manipulated only by the TPMs in the cloud. Hence, even if the user logs onto the server through a public computer, the private key is still secure and not released to the public computer. The TPM cloud brings strong security to Web applications without sacrificing users flexibility of using different computer platforms.
- We analyze the security properties of the virtual TPMs in the TPM cloud by comparing them with physical TPMs and software-based virtual TPMs. We conclude that the virtual TPMs in the TPM cloud are as secure as physical TPMs in practical applications (Sect. 5).

A preliminary version of this paper was presented in [13]. In this new version, we revised the key hierarchy in the TPM cloud, and formalized all components of the cloud architecture and more cloud commands to cover the typical use of the TPM cloud. The new key hierarchy makes it easier to migrate keys among physical TPMs, hence making the cloud architecture simpler as discussed in Sect. 3. We also discussed the establishment of trust chains based on the TPM cloud and analyzed the security properties in this version.

2. Architecture

The architecture of TPM cloud is shown in Fig. 2. This architecture includes a virtual TPM service, a user management component, a cryptographic service, and a number of physical TPM services and their management.

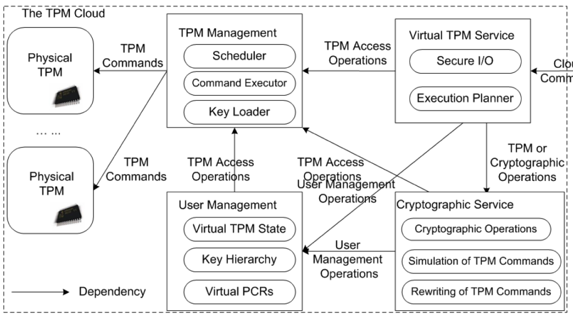


Fig. 2 The cloud architecture of virtual TPMs.

2.1 Virtual TPM Service

The virtual TPM service has the cloud port as the interface to users or applications, such as the Javascript Web pages and the Web server in the implementation of the Needham-Schroeder public-key protocol. Through this port, users or applications send cloud commands to consume TPM services, such as registering new TPM instances, creating keys and signing data.

The secure I/O module in this component protects the communication between the TPM cloud and its users. The TPM cloud is designed to have a Cloud Key (CK), which is a RSA key. We denote the public part of CK as $\mathtt{PK_c}$. Every cloud command received should be encrypted with $\mathtt{PK_c}$. In the TPM cloud, all operations involving private keys must be performed within physical TPMs (a security feature of TPM), so the secure I/O module relies on physical TPMs to decrypt the encrypted cloud commands. In other words, the secure I/O module needs not to know the private CK.

The result of virtual TPM service should also be protected, since the result (such as a decryption result) might contain valuable data. If needed, the result of cloud commands is encrypted with symmetric encryption algorithms before sending back. By using symmetric encryption, users do not keep and manage private keys on their platforms. The symmetric key for encrypting the result is generated by users and provided as an argument in the cloud command.

The execution planner determines the execution of cloud commands after being decrypted by the secure I/O module. It needs to select the component (the TPM management, the cryptographic service or the user management) and may need to divide a cloud command into a sequence of steps, suitable for other components to execute. As an example, to execute a cloud command for data decryption, the execution planner needs to load the key into a physical TPM and then call the physical TPM_UnBind command.

2.2 User Management

The user management component manages the state of users and maintains a TPM instance for each user. A TPM in-

[†]This report appears after our paper [13] is presented.

stance includes the virtual TPM state, a key hierarchy and a set of virtual PCRs.

The virtual TPM state indicates the state of a TPM instance by simulating state flags in the physical TPM, such as the flags indicating whether a TPM instance is enabled or disabled, activated or deactivated. The virtual state does not need to support all state flags in the physical TPM since some flags might not make sense for TPM instances in cloud, such as the state flag indicating the physical presence of a human since virtual TPMs in the cloud are accessed as a network service. On the other hand, TPM instances may have flags not included in physical TPMs. As a future work, a `delegated` flag could be used to indicate whether a TPM instance is delegated by its owner to other users.

Before introducing the key hierarchy for a virtual TPM, we describe briefly the key management in a physical TPM. Each TPM has a unique endorsement key (EK), which is generated by the chip manufacturer. Before using a TPM, users need to take the ownership of the TPM and create a storage root key (SRK). Both EK and SRK are RSA key pairs. Other RSA keys in the TPM are created under a parent key. Their private keys are always used within TPM and when released outside TPM they are encrypted with their parent keys. The SRK can be used as a parent key.

The key hierarchy for a virtual TPM is shown in Fig. 3. A virtual TPM has a virtual EK and a virtual SRK, which are both generated by a physical TPM with a Virtual TPM Root Key as their parent key. As a migratable storage key created under SRK, the Virtual TPM Root Key is created on one physical TPM, and then migrated and loaded into every physical TPM. Therefore, the virtual EK and SRK can be loaded into every physical TPM since the Virtual TPM Root Key is already there. Using Virtual TPM Root Key is an important improvement over the old cloud architecture in [13]. The Virtual TPM Root Key facilitates the management of physical TPMs and the migration of keys among them, with more details discussed in Sect. 3.1. In addition, the Cloud Key is also created on one physical TPM, and then migrated and loaded into every physical TPM.

The virtual EK is created as a binding key since its public key is used to encrypt data, for instance, by Privacy CA when activating an identity key. The virtual SRK is a storage key and used as the parent key to create other keys. All keys in the virtual TPM are migratable, such that they are not bound to a physical TPM. This feature is useful to maintain the availability of virtual TPMs in case of physical TPM

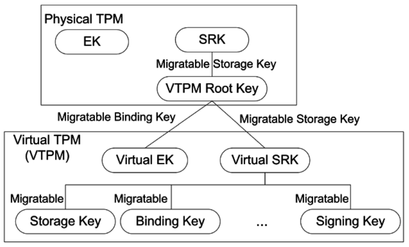


Fig. 3 The key hierarchy for virtual TPM.

failure and balance the load among physical TPMs. A key can move from one physical TPM to another for unbinding data if the target TPM is not busy.

A virtual PCR is a sequence of bytes with the same length as a physical PCR. Virtual PCRs stay outside physical TPMs. To protect their contents, we encrypt virtual PCRs with the virtual public EK. Before executing a PCR dependent TPM command, the virtual PCRs need to be decrypted with the corresponding virtual private EK. Since virtual PCRs are not in physical TPMs, the TPM commands dependent on PCRs need to be specially treated, as discussed below.

2.3 Cryptographic Service

The cryptographic service implements some cryptographic operations (e.g., symmetric encryption algorithms) that are not provided by physical TPMs. The cryptographic operations are similar to vendor-specific commands in some physical TPMs. For example, an Atmel TPM has the specific commands `TPM_BindV20` and `TPM_VerifySignature` for public-key encryption and verification, respectively.

The cryptographic service also deals with the TPM commands relying on PCRs. A critical feature for physical PCRs is that they can only be extended and not all of them are resettable. Hence, it is hard to swap a set of virtual PCRs into the physical PCRs as in classic virtual memory management systems. However, we realize that some PCR dependent TPM commands can still be implemented in physical TPMs after rewriting, while others have to be simulated in software. Our solution is that if a command only refers to the state of PCRs, then it is rewritten for execution on a physical TPM; if a command needs the actual values of PCRs, then it is simulated. A command is said to refer to the state of PCRs if it does not change PCRs and only use them to affect the execution of subsequent TPM commands; otherwise we say it needs the actual values of PCRs. For example, the command `TPM_Extend` needs the actual values of the argument PCRs since it changes the values of PCRs; the command `TPM_CreateWrapKey` refers to the state of the argument PCRs since the resulting key can only be loaded into a TPM by a subsequent command `TPM_LoadKey` if the PCRs are not changed. That is, the commands `TPM_CreateWrapKey` and `TPM_LoadKey` do not care about the actual values of PCRs, just requiring them to have the same values. Moreover, the command `TPM_CreateWrapKey` has to be executed by a physical TPM since it generates RSA keys. The method of rewriting PCR dependent TPM commands is described in Sect. 3.3.

2.4 Physical TPMs and Their Management

The TPM cloud depends on physical TPMs to manipulate private keys and execute TPM commands (except some commands replying on PCRs). The TPM cloud might have a large number of users. For the scalability of the TPM cloud, the cloud architectures supports a number of physical TPMs.

The physical TPM management component coordinates all physical TPMs to execute multiple TPM commands in parallel. The scheduler in this component determines which physical TPM is selected to execute a new command, such that the workload of physical TPMs is balanced. The executor accepts the requests to execute TPM commands and issues the commands to physical TPMs. The key loader loads a key into a physical TPM to be used by other TPM commands. The key is not necessarily created on the same TPM.

3. Formal Functionality of the TPM Cloud

We describe formally the functionality of each TPM cloud component and their interactions to explain the processing of cloud commands. The formal functionality specification lays a foundation for the analysis and implementation of TPM cloud. We start with the description of several simplified physical TPM commands to be used. Details of them can be found in the TCG TPM specification [1].

TPM_LoadKey(*pHandle*, *pUsageAuth*, *key***):** Loads *key* into a physical TPM and returns a key handle. The parent of *key* is specified by the handle *pHandle* and has the usage authorization (or password) *pUsageAuth*.

TPM_BindV20(*key*, *data***):** Encrypts *data* with *key*.

TPM_UnBind(*handle*, *usageAuth*, *encdata***):** Decrypts the encrypted data *encdata* with the key specified by *handle*. The usage authorization of the key is *useageAuth*.

TPM_Extend(*index*, *data***):** Extends the PCR *index* with *data*.

TPM_PCR_Reset(*index***):** Resets the PCR *index* to its default initial value.

TPM_CreateWrapKey(*pHandle*, *pUsageAuth*, *usageAuth*, *indices***):** Creates a new key having the usage authorization *usageAuth* under the parent key *pHandle* with the usage authorization *pUsageAuth*. The new key is locked by PCRs *indices*.

TPM_Sign(*handle*, *usageAuth*, *data***):** Signs *data* with the key specified by *handle*. The key has the usage authorization *useageAuth*.

The Cloud Key and Virtual TPM Root Key are only used inside the cloud, so in the following we give them a default usage authorization, represented by a underscore _.

3.1 Physical TPM Representation and Access Operations

In the TPM management component, a physical TPM is represented as a tuple (*id*, *ckhandle*, *rkhandle*), where *id* is an identifier, *ckhandle* the handle for the Cloud Key, and *rkhandle* the handle for the Virtual TPM Root Key. Recall that the Cloud Key and Virtual TPM Root Key are loaded into every physical TPM. Let `pTPM` be a set of physical TPMs managed by the TPM management component. Suppose `pTPM` includes a TPM (*id*, *ckhandle*, *rkhandle*). Then, we refer to this TPM by `pTPM`[*id*], its components *ckhandle* and *rkhandle* by `pTPM`[*id*].ckhandle and `pTPM`[*id*].rkhandle, respectively.

Due to the use of the Virtual TPM Root Key, the management of physical TPMs in this new cloud architecture is simpler. Unlike the representation of physical TPMs in [13], users are not linked to physical TPMs any more in the above TPM representation, since the Virtual TPM Root Key is used as the parent of all virtual EKs and virtual SRKs. That is, by using the Virtual TPM Root Key, a user is no longer bound to a particular physical TPM in this new cloud architecture because the SRK of a physical TPM is not used as the parent key for creating his virtual EK and SRK.

The TPM management component provides the following operations for other components to access physical TPMs. The operation `schedule`() returns the identifer of a physical TPM, which is scheduled to execute a command. The operation `execute`(*id*, *tpmcmd*) executes the TPM command *tpmcmd* on the physical TPM *id*. For example, `execute`(*id*, `TPM_Extend`(*i*, *data*)) sends the command `TPM_Extend`(*i*, *data*) to physical TPM *id* for executing.

The operation `load`(*name*, *id*, *vsrkpwd*, *key*) defined in Fig. 4 loads *key* into the TPM *id*, and returns a handle. *key* is supposed to be created under the virtual SRK of user *name*, which has the usage authorization *vsrkpwd*. Briefly, this operation first loads the virtual SRK of user *name* into the TPM *id*, and then loads *key* into the same TPM. When loading the virtual SRK, the TPM *id* uses the Virtual TPM Root Key on it as the parent key. The virtual SRK is then used as the parent key to load *key*. The virtual SRK *vsrk* is retrieved through the operation getVTPM(*name*).keyhrk.srk, which is described later. Compared with the `load` operation in [13], the operation in Fig. 4 is simpler since we do not need to rewrap *key* on the source physical TPM. The Virtual TPM Root Key is loaded into every physical TPMs, so we can directly load *key* and its parent virtual SRK into the target physical TPM.

3.2 Representation of Users and Virtual TPMs

A user is described by a pair (*name*, *vtpm*), meaning that the user *name* has the TPM instance *vtpm*. A TPM instance *vtpm* is represented by a tuple (*ownerauth*, *state*, *keyhrk*, $\{pcrs\}_{pubek}$), where *ownerauth* is the owner authorization of the TPM instance, *state* its state, *keyhrk* the key hierarchy created for this TPM instance, and *pcrs* a set of PCRs encrypted with the public virtual EK *pubek*.

For a TPM instance, we consider only the ownership state for simplicity. That is, the *state* field in *vtpm* is a boolean to indicate whether the command `TPM_TakeOwnership` is executed or not. Other state flags

```
load(name, id, vsrkpwd, key){
  1. rkhdl = pTPM[id].rkhandle;
  2. vsrk = getVTPM(name).keyhrk.srk;
  3. vsrkhdl =execute(id, TPM_LoadKey(rkhdl, _, vsrk);
  4. keyhdl =execute(id, TPM_LoadKey(vsrkhdl, vsrkpwd, key);
}
```

Fig. 4 Loading key into TPM.

can be supported similarly. For example, we can add other boolean-valued fields to indicate whether a TPM instance is enabled or activated.

The key hierarchy in a TPM instance is described by the tuple $(ek, srk, \{(key_1, handle_1), \ldots, (key_n, handle_n)\})$, including the virtual EK *ek*, the virtual SRK *srk*, and a set of pairs of key_i and $handle_i$. The key key_i is created under *srk*, as shown in Fig. 3. A key is a pair $(\{usageauth, privkey\}_{srk.\texttt{pubkey}}, pubkey)$, consisting of the public key *pubkey* and a blob of private key *privkey* and its usage authorization *usageauth* encrypted with the public key *srk*.pubkey. Hence, the private key is protected when stored in a virtual TPM.

Let $\mathcal{U}$ be a list of users. The operation getVTPM(*name*) returns the TPM instance owned by the user *name*. That is, if $(name, vtpm) \in \mathcal{U}$, then getVTPM(*name*) = *vtpm*. Given a TPM instance *vtpm*, its four fields are referred to by the notations *vtpm*.ownerauth, *vtpm*.state, *vtpm*.keyhrk and *vtpm*.encpcrs. For the key hierarchy in *vtpm*, the virtual EK and the virtual SRK are accessed by using the notations *vtpm*.keyhrk.ek and *vtpm*.keyhrk.srk, respectively. Other keys in the hierarchy are referred to by *vtpm*.keyhrk.keys. We use *vtpm*.keyhrk.keys[*handle*] for the key indexed by the handle *handle*, and *vtpm*.keyhrk.keys[*key*] for the handle of *key*. That is, if $(key, handle) \in$ *vtpm*.keyhrk.keys, then *vtpm*.keyhrk.keys[*handle*] = *key* and *vtpm*.keyhrk.keys[*key*] = *handle*. For a key *key*, its public key and private key are accessed by *key*.pubkey and *key*.privkey, respectively.

A new user or a new TPM instance is created by using the registration operation register(*name*, *passwd*) in Fig. 5. The operation starts by creating a virtual EK *ek* under the Virtual TPM Root Key on TPM *id*. We do not require PCRs to lock *ek*, so the last argument of TPM_CreateWrapKey is an empty PCR index list []. Next, the initial PCRs initpcrs (an array with each entry initialized as 20 bytes of 0 s) is encrypted by the public virtual EK *ek*.pubkey. At last, a tuple for the new user is added into $\mathcal{U}$, where the password is stored after hashing. Since the ownership of the new TPM instance is not taken, the ownership state is false and accordingly the virtual SRK is not existing (represented by an underscore _).

3.3 Simulation and Rewriting of TPM Commands

The implementation of cryptographic operations (e.g., symmetric encryption) is straightforward. In this section, we describe how the TPM commands that are dependent on PCRs are specially treated. The commands TPM_Extend and TPM_CreateWrapKey are taken as examples.

The command TPM_Extend is simulated since it needs the actual PCRs values. The simulation in Fig. 6 follows firmly the semantics of the physical TPM_Extend command. In the simulation, the virtual PCRs of user *name* is first decrypted with the virtual EK by calling the command TPM_UnBind. Next, the concatenation of virtual PCR *index* and *data* (i.e., pcrs[*index*]||*data*) is hashed, resulting in the new (or extended) virtual PCRs. At last, the extended virtual PCRs are encrypted with the public virtual EK of user *name* and put back to the TPM instance.

The operation createkey in Fig. 7 creates a key under the virtual SRK of user *name* and returns a handle *khdl*. The usage of virtual SRK is protected by *passwd*, and the new key has the usage authorization *usageauth*. Different from pcrextend, the operation createkey relies on the corresponding physical TPM command TPM_CreateWrapKey to create keys within a physical TPM, such that the private keys are not released ourside physical TPMs during generation. There are two cases of invoking TPM_CreateWrapKey. If the argument *indices* is an empty list [], meaning that the new key is not locked by any PCR, then the arguments of createkey are passed to TPM_CreateWrapKey directly. Otherwise, the argument *indices* is rewritten (as described below) before passed.

```
register(name, passwd){
  1. id = schedule(); rkhdl =pTPM[id].rkhandle;
  2. ek = execute(id,TPM_CreateWrapKey(rkhdl, _, passwd,[]));
  3. encpcrs =execute(id,TPM_BindV20(ek.pubkey,initpcrs));
  4. newuser = (name, (hash(passwd), false, (ek, _, ∅), encpcrs));
  5. 𝒰 = 𝒰 ∪ {newuser};
}
```

Fig. 5 Registration of new users.

```
pcrextend(name, passwd, index, data){
  1. id = schedule(); rkhdl =pTPM[id].rkhandle;
  2. vek =getVTPM(name).keyhrk.ek;
  3. vekhdl =execute(id, TPM_LoadKey(rkhdl, _, vek));
  4. epcrs =getVTPM(name).encpcrs;
  5. pcrs =execute(id, TPM_UnBind(vekhdl, passwd, epcrs));
  6. pcrs[index] = hash(pcrs[index]||data);
  7. getVTPM(name).encpcrs = execute(id,
                              TPM_BindV20(vek.pubkey,pcrs));
}
```

Fig. 6 Simulation of TPM_Extend.

```
createkey(name, passwd, usageauth, indices){
  1. id = schedule(); rkhdl =pTPM[id].rkhandle;
  2. vsrk =getVTPM(name).keyhrk.srk;
  3. vsrkhdl =execute(id, TPM_LoadKey(rkhdl, _, vsrk));
  4. if indices=[] then
  5.   khdl = execute(id, TPM_CreateWrapKey(vsrkhdl,
                                passwd, usageauth, []));
  6. else
  7.   vek =getVTPM(name).keyhrk.ek;
  8.   vekhdl =execute(id, TPM_LoadKey(rkhdl, _, vek));
  9.   epcrs =getVTPM(name).encpcrs;
  10.  pcrs =execute(id, TPM_UnBind(vekhdl, passwd, epcrs));
  11.  execute(id, TPM_PCR_Reset(rindex));
  12.  for each index in indices do
  13.      execute(id, TPM_Extend(rindex,pcrs[index]));
  14.  khdl = execute(id, TPM_CreateWrapKey(vsrkhdl,
                               passwd, usageauth, rindex));
}
```

Fig. 7 Rewriting of TPM_CreateWrapKey.

We cannot simply put the virtual PCRs specified by *indices* into the corresponding physical PCRs since PCRs can only be extended. Our solution is to use a resettable physical PCR `rindex` to record the current state of virtual PCRs *indices*. To this end, we reset the PCR `rindex` and extend this PCR with each specified virtual PCR. Then, we invoke the physical command TPM_CreateWrapKey with `rindex` as its argument for specifying a physical PCR. For the command TPM_Load, it can also refer to PCRs when loading a key (the argument of TPM_Load for specifying PCRs is not given in this paper). The virtual PCRs for TPM_LoadKey are also treated in the same way. Hence, when the same virtual PCRs are specified for TPM_CreadWrapKey and TPM_LoadKey, the resettable physical PCR `rindex` would have the same value, allowing a key locked by the specified PCRs to be successfully loaded. Our solution shows an indirect way of using virtual PCRs when considering the restriction of physical PCRs.

3.3.1 Limitations of TPM Specification

From the formalization of `createkey`, we find that the TPM specification should allow an extra argument for the command TPM_TakeOwnership, which can configure whether PCRs are or are not resettable when taking the ownership of a TPM. This will bring much flexibility for the usage of PCRs in various scenarios. For example, a user of a PC might want all PCRs in its TPM not to be resettable, while the TPM cloud wants all PCRs to be resettable. If all PCRs are resettable, the virtual TPM commands using PCRs can be implemented more easily.

In addition, the TPM specification requires the commands TPM_Seal and TPM_UnSeal use nonmigratable keys. Since the TPM cloud is only supposed to support migratable keys for better load balancing, such commands provided by physical TPMs become useless. On the other hand, the TPM_Seal command binds the sealed data with the secret value tpmProof, which is only known within a TPM. That is, a sealed data cannot be unsealed by another physical TPM even if the two TPMs have the same PCRs. If the TPM specification could allow TPM_Seal to use migratable keys to seal data only with the values of PCRs, then the physical commands TPM_Seal and TPM_UnSeal can be used in the TPM cloud. Moreover, it also makes it possible to unseal data when it is sealed to a TPM that is broken.

3.4 Cloud Commands Processing

Cloud commands are encrypted with the public Cloud Key $\mathrm{PK_C}$. To decrypt a cloud command *ecmd*, the secure I/O module executes the following two steps: selects a physical TPM *id* and then executes the command TPM_UnBind.

```
id = schedule();
cmd =execute(id, TPM_UnBind(pTPM[id].ckhandle,_, ecmd);
```

After a cloud command is decrypted, it is passed to the execution planner, where the steps of processing cloud commands are defined. In the following, we take several cloud commands as examples to describe their processing. These commands cover the typical uses of TPM cloud, ranging from registration of new users, generation and use of keys to extension of PCRs.

3.4.1 Preparation of New Virtual TPMs

A new virtual TPM is created when users send a registration command `cTPM_Register`(*name*, *passwd*) (i.e., the cloud command after decryption). The prefix “`cTPM_`” is used to indicate cloud commands. The implementation of `cTPM_Register` is straightforward. The execution planner just needs to call the operation `register` in the user management component with the same arguments.

After a new virtual TPM is created, users need to take its ownership, like using a physical TPM. The cloud command `cTPM_TakeOwnership`(*name*, *passwd*) is for this purpose. Figure 8 gives the steps to take ownership. The execution planner first checks the user password and makes sure the ownership of TPM instance for user *name* has not been taken. After these checks, a new virtual SRK is created and recorded into the TPM instance. Like the virtual EK, the virtual SRK is also protected with the usage authorization *passwd*. Hence, when a user is authenticated to own a TPM instance, he can access the virtual EK and SRK without providing extra passwords. This design makes the cloud commands more convenient to use. At last, the state of the TPM instance is changed accordingly to indicate that the ownership has been taken.

3.4.2 Key Creation and Loading in TPM Cloud

The cloud command `cTPM_CreateWrapKey`(*name*, *passwd*, *usageAuth*, *indices*) is used to create a key for user *name* under his virtual SRK. The new key is protected by the usage authorization *usageAuth* and locked by virtual PCRs *indices*. The steps implementing this command is shown in Fig. 9. First, the user *name* is authenticated by checking the password *passwd*, and then the `createkey` operation in the cryptographic service is invoked with the same arguments. At last, the new key is added into the key hierarchy. There is no key handle yet (indicated by a underscore _), since the key has not been loaded into a TPM.

A key is loaded into a virtual TPM by the cloud command `cTPM_LoadKey`(*name*, *passwd*, *key*), defined in

```
cTPM_TakeOwnership(name, passwd){
  1. vTPM= getVTPM(name);
  2. assert(hash(passwd)=vTPM.ownerauth);
  3. assert(vTPM.state = false);
  4. id = schedule(); rkhdl = pTPM[id].rkhandle;
  5. srk =execute(id, TPM_CreateWrapKey(rkhdl, _, passwd,[]));
  6. vTPM.keyhrk.srk = srk;
  7. vTPM.state = true;
}
```

Fig. 8 Steps for taking TPM ownership.

```
cTPM_CreateWrapKey(name, passwd, usageAuth, indices){
  1. vTPM= getVTPM(name);
  2. assert(hash(passwd)=vTPM.ownerauth);
  3. key = createkey(name, passwd, usageAuth,indices);
  4. vTPM.keyhrk.keys = vTPM.keyhrk.keys∪{(key,_)};
}
```

Fig. 9 Steps for creating keys.

```
cTPM_LoadKey(name, passwd, key){
  1. vTPM= getVTPM(name);
  2. assert(hash(passwd)=vTPM.ownerauth);
  3. id = schedule();
  4. hdl = load(name, id, passwd, key);
  5. vTPM.keyhrk.keys[key] = hdl;
}
```

Fig. 10 Steps for loading keys.

```
cTPM_UnBind(name, passwd, hdl, usgAuth, encdata, sk){
  1. vTPM = getVTPM(name);
  2. assert(hash(passwd)=vTPM.ownerauth);
  3. assert(vTPM.state = true);
  4. id = schedule();
  5. newhdl = load(name, id, passwd,vTPM.keyhrk.keys[hdl]);
  6. data =execute(id, TPM_UnBind(newhdl, usgAuth, encdata));
  7. result =encrypt(data, sk);
}
```

Fig. 11 Steps for unbinding data.

```
cTPM_Sign(name, passwd, hdl, usgAuth, data){
  1. vTPM = getVTPM(name);
  2. assert(hash(passwd)=vTPM.ownerauth);
  3. assert(vTPM.state = true);
  4. id = schedule();
  5. newhdl = load(name, id, passwd,vTPM.keyhrk.keys[hdl]);
  6. result =execute(id, TPM_Sign(newhdl, usgAuth, data));
}
```

Fig. 12 Steps for signing data.

Fig. 10. This command depends on the `load` operation provided by the physical TPM management component. When a handle for *key* is generated, it will be put into the key hierarchy together with *key*.

3.4.3 Data Decryption and Signing

The cloud commands using private keys for data decryption and signing are defined in Fig. 11 and Fig. 12, respectively. The command `cTPM_UnBind`(*name*, *passwd*, *hdl*, *usgAuth*, *encdata*, *sk*) decrypts *encdata* for user *name* by using the key *hdl*, which is protected by *usgAuth*. The decrypted result will be encrypted with the symmetric key *sk*. The symmetric encryption is done by using the `encrypt` operation provided by the cryptographic service. Similarly, the command `cTPM_Sign`(*name*, *passwd*, *hdl*, *usgAuth*, *data*) signs *data* for user *name*.

These two commands have the same first five steps, in which the key with handle *hdl* is reloaded into a physical TPM, resulting in a new handle. The reloading is needed because a physical TPM has limited slots to accommodate loaded keys and thus a previously loaded key might have been unloaded by the physical TPM management component. Hence, we need to reload it without users noticing their key have been unloaded. After a new handle is generated, the physical TPM command TPM_UnBind and TPM_Sign are used to do the actual decryption and signing since private keys must be used within physical TPMs.

3.4.4 Extension of Virtual PCRs

The cloud command `cTPM_Extend`(*name*, *passwd*, *index*, *data*) extends the virtual PCR *index* with *data* for user *name*. The implementation of this command is straightforward, as shown below. First, the user *name* is authenticated. Then, the *pcrextend* operation provided by the cryptographic service is invoked with the same arguments.

```
assert(hash(passwd)=getVTPM(name).ownerauth);
pcrextend(name, passwd, index, data);
```

4. Applications of The TPM Cloud

We have implemented a prototype of the TPM cloud to demonstrate its functionality. Currently, physical TPMs are usually embedded into computer motherboards, with one computer having at most one physical TPM. Our prototype uses a cluster of Dell Optiplex GX620 to provide physical TPMs. The TPM functionalities on these computers are exposed as web services, and they are accessed by the TPM management component with SOAP. The cloud commands are also exposed as web services, such that the TPM functionality provided by the TPM cloud can be easily accessed. The web service engine we used is Apache Axis2 [15].

TCG advocates that trusted computing technology can be used to improve the security of computer systems (e.g., improved authentication and network access control) [1]. In this paper, we apply the TPM cloud to the implementation of the Needham-Schroeder public-key protocol (NS protocol) for web authentication, as it is a simple and well-analyzed authentication protocol. As a result, the TPM functionality is incorporated into high-level applications, useful to improve the acceptance of TPM functionalities by users.

To use the TPM service, users need to provide their user names and passwords, as shown by the cloud commands. When using the TPM service on a public computer, the malware there might steal the passwords and thus can access the secrets of users in the cloud. This problem can be solved for instance by authenticating cloud commands with some second factor (e.g., smartcard issued by the TPM cloud provider). Hence, when just keeping one second factor for the cloud rather than many second factors for different web servers, a user can securely exploit the cloud to run strong protocols to authenticate him to many different web servers.

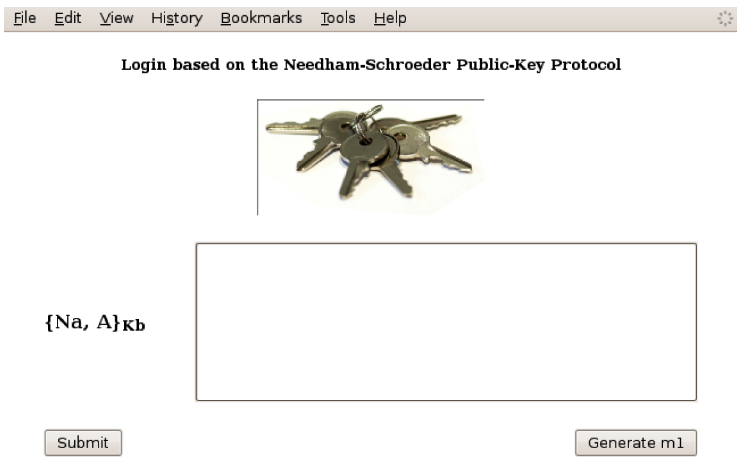
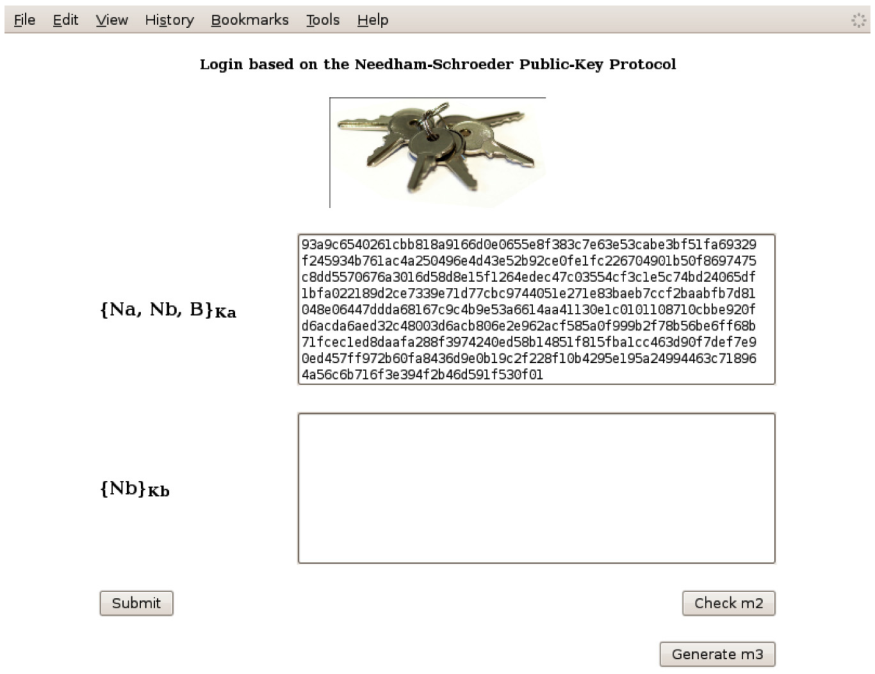

File Edit View History Bookmarks Tools Help
Login based on the Needham-Schroeder Public-Key Protocol
{Na, A}Kb
Submit
Generate m1

Fig. 13 The first login page.


File Edit View History Bookmarks Tools Help
Login based on the Needham-Schroeder Public-Key Protocol
{Na, Nb, B}Ka
{Nb}Kb
Submit
Check m2
Generate m3

```
93a9c6540261cbb818a9166d0e0655e8f383c7e63e53cabe3bf51fa69329
f245934b761ac4a250496e4d43e52b92ce0fe1fc226704901b50f8697475
c8dd5570676a3016d58d8e15f1264edec47c03554cf3c1e5c74bd24065df
1bfa022189d2ce7339e71d77cbc9744051e271e83baeb7ccf2baabfb7d81
048e06447ddda68167c9c4b9e53a6614aa41130e1c0101108710cbbe920f
d6acda6aed32c48003d6acb806e2e962acf585a0f999b2f78b56be6ff68b
71fcec1ed8daafa288f3974240ed58b14851f815fba1cc463d90f7def7e9
0ed457ff972b60fa8436d9e0b19c2f228f10b4295e195a24994463c71896
4a56c6b716f3e394f2b46d591f530f01
```

Fig. 14 The second login page.

In the following, we introduce the implementation of NS protocol for web authentication based on the TPM cloud. Suppose that user A and the web server B have registered TPM instances in the cloud and created their RSA keys there. Let $\mathrm{PK_A}$ and $\mathrm{PK_B}$ be the public keys of A and B, respectively. Then, the NS protocol is run by exchanging the following three messages m1, m2 and m3. The two nonces Na and Nb are random numbers generated by A and B, respectively.

$$\begin{array}{lll} m1 & A \rightarrow B: & \{Na, A\}_{PK_B} \\ m2 & B \rightarrow A: & \{Na, Nb, B\}_{PK_A} \\ m3 & A \rightarrow B: & \{Nb\}_{PK_B} \end{array}$$

We show the web pages used by the user to run the NS protocol. The pages are embedded with the Javascript code that implements the RSA and DES encryption algorithms. Recall that the cloud commands are encrypted with the public Cloud Key, and the result of the cloud command might be encrypted with some symmetric encryption algorithm (e.g., DES algorithm). In this application, the user and the server use the TPM cloud to decrypt the messages in the NS protocol, so that their private keys are always kept inside the TPM cloud.

The first page for logging on the web server is shown in Fig. 13, where the user can click the “Generate m1” button to generate the message m1 and click the Submit button to send this message to the server. The TPM cloud is not used for generating the first message since the public key $\mathrm{PK_B}$ is known and the encryption algorithm is provided in the page. However, the server needs the cloud to unbind the received message since its private key is stored there.

After the server receives the decrypted message, it sends back the second page, shown in Fig. 14. This page shows the second message sent by the server. The user now checks whether the second message is valid. If valid, the user then sends the third message. The “Check m2” button is to check the second message. Clicking on this button will send a cloud command to decrypt the second message after prompting the user to input some necessary information such as the user name and password. If the decrypted message contains correct nonce Na, a popup window will say the server is authenticated. And then, the user generates the third message by clicking the “Generate m3” button and sends it back by clicking the submit button. Upon receiving the third message, the web server uses the cloud to decrypt it and if the decrypted message contains correct Nb, then it replies a page to tell the successful authentication of the user.

This application demonstrated that the TPM functionality can be extended to high-level applications by using the TPM cloud, which provides a way of improving the usability of TPMs. To the best of our knowledge, our work shows the first application of TPM functionality to the Javascript programs in Web pages.

5. The Trust Chain Based on The TPM Cloud

In this section, we discuss the chain of trust that can be established based on the TPM cloud. The TPM cloud has the same functionality as a physical TPM. To implement remote attestation, a platform can store its platform measurements remotely into the cloud and then depend on the cloud to sign and report the measurements. In the following, we discuss how to establish the trust of chain based on the TPM cloud.

Suppose there is a platform that executes a sequence of programs $\mathtt{P}_0$, $\mathtt{P}_1$, $\mathtt{P}_2$,..., $\mathtt{P}_n$ after its power is switched on. The program $\mathtt{P}_0$ is firstly executed, and then it starts the second program $\mathtt{P}_1$, which launches the next program $\mathtt{P}_2$, until the $\mathtt{P}_n$ is started by $\mathtt{P}_{n-1}$. Usually, the program $\mathtt{P}_0$ is a boot loader. We assume there is a program $\mathtt{P}_{\mathtt{net}}$ ($0 \leq \mathtt{net} \leq n$), which enables the platform to access network (hence the TPM cloud) after being executed. The establishment of the trust chain is discussed in two cases.

In the first case, we suppose a platform that does not have a physical TPM. For this platform, the trust establishment technique in [2] is not applicable since there is no physical TPM to store and sign the program measurements. By using the TPM cloud, the platform gets the opportunity to build the trust chain. The method is that before $\mathtt{P}_{\mathtt{net}}$ launches $\mathtt{P}_{\mathtt{net+1}}$, it measures the program $\mathtt{P}_{\mathtt{net+1}}$ and the measurement (i.e., the hash of $\mathtt{P}_{\mathtt{net+1}}$) is extended into the PCRs in the TPM cloud by using the cloud command `cTPM_Extend`; the chain of trust is established until $\mathtt{P}_n$ is measured and extended into the PCRs in the cloud by $\mathtt{P}_{n-1}$.

In this case, the programs from $\mathtt{P}_0$ to $\mathtt{P}_{\mathtt{net}}$ comprise the trusted computing base. By comparison, in [2], only $\mathtt{P}_0$ (the boot loader) is included in the trusted computing base if there is a physical TPM on the platform. To make the trusted computing base smaller, the program $\mathtt{P}_{\mathtt{net}}$ should be executed after $\mathtt{P}_0$ as closely as possible. For example, the program $\mathtt{P}_1$ can be designed to enable network access and then launch the operating system or virtual machine monitor, such that the TPM cloud can be used to build the trust chain by trusting only $\mathtt{P}_0$ and $\mathtt{P}_1$.

In the second case, the platform is assumed to have a physical TPM. For this platform, the trust chain from $\mathtt{P}_0$ to $\mathtt{P}_{\mathtt{net}}$ is built into PCRs in the physical TPM, as described in [2]. After the program $\mathtt{P}_{\mathtt{net}}$ is executed, it sends to the TPM cloud the measurements of programs from $\mathtt{P}_1$ to $\mathtt{P}_{\mathtt{net}}$, together with the PCR values quoted from the physical TPM. The TPM cloud then checks the integrity of the measurements based on the signed PCR values. If valid, then they are extended into the PCRs in the TPM cloud. The programs from $\mathtt{P}_{\mathtt{net+1}}$ to $\mathtt{P}_n$ are dealt with similarly as in the first case. In this case, the trusted computing based is only $\mathtt{P}_0$ (the boot loader).

In this case, if $\mathtt{P}_1$ is a virtual machine monitor, each virtual machine on the monitor is allowed to have their own trust chain even if the platform is shared by all virtual machines. That is, the trust chain for each virtual machine includes the measurements of the virtual machine monitor, the virtual machine, and other programs managed by the virtual machine. This is important for the cloud computing platforms such as Amazon Elastic Compute Cloud (Amazon EC2), where virtual machines usually belong to different users and their trust chains should be separated.

6. Security Analysis of The TPM Cloud

The TPM cloud improves the usability of physical TPMs by virtualizing the physical TPMs in a cloud architecture. In this section, we analyze the security properties of virtual TPMs in the TPM cloud, which is based on physical TPMs to execute all commands involving private keys and other commands suck as `TPM_Extend`. The analysis is done by comparing them with physical TPMs and the software-based virtual TPMs (vTPMs) [16], which implement TPM functionality as user-space software.

To compare the security property of the various TPM implementations, we first need to determine an attack model. That is, what information an attacker is allowed to know and what operations an attacker is allowed to do. In different attack models, a system may have different security properties. For example, if an attack model allows attackers to tamper hardware rather than software, then the software-based TPM can provide more secure TPM functionality than physical TPMs and virtual TPMs based on physical TPMs in the TPM cloud.

In this section, we use an attack model that is more possible. In this model, we suppose the attacker has installed some malicious software on a platform and thus the attacker can know all contents semantically in the platform memory and change those contents if he wants. To do our analysis, we assume a platform is a machine that has a phys-

Table 1 Security analysis of three TPM implementations.

	Physical TPMs (pTPMs)	Software-based virtual TPMs (vTPMs)	Virtual TPMs in TPM Cloud (vpTPMs)
Key Generation	√	X	√
Signing with Private Keys	√	X	√
Decrypting with Private Keys	√	X	√
PCR Reset	√	X	X
PCR Extend	X	X	X

ical TPM or a software-based virtual TPM, or the platform that implements the TPM cloud. For example, if a key is put into the memory, then the attacker knows it is a key and its value. This situation can happen when the platform is compromised during execution even after trusted booting, known as the vulnerability of time-of-check time-of-use (TOCTOU) [17]. In addition, we assume that the physical TPM can only be handled by using TPM commands and the attacker cannot decrypt a message if the needed key is not known by him.

After determining the attack model, we choose the security properties of TPM functionality to compare. As introduced in Sect. 1, a TPM lies in its capabilities of secure key management, and secure storage and reporting of platform configuration measurements in PCRs. Hence, the secrecy of private keys and the integrity of PCR values are the most important security properties of TPMs.

We compare the secrecy of private keys and the integrity of PCR values with respect to several kinds of TPM commands implemented in the physical TPMs, the software-based virtual TPMs, and the virtual TPMs based on physical TPMs in the cloud. These commands deal with private keys or PCRs. The comparison result is given in Table 1, where the symbol √ means the secrecy or integrity property is respected, while X means not. The comparison is explained below and for brevity we use pTPMs for physical TPMs, and vpTPMs for the virtual TPMs based on physical TPMs.

For key generation, private keys are secret in both pTPMs and vpTPMs since privates keys are generated within physical TPMs and when they are moved to the platform memory they are encrypted with the parent keys, which have their private parts only used within physical TPMs. Hence, the attacker has no way to know the private keys in pTPMs and vpTPMs. For vTPMs, the private keys are generated in the platform memory, so the attacker can read the generated private keys in our attack model.

When signing or decrypting with private keys, pTPMs and vpTPMs do not expose private keys to the attacker, since they are used within physical TPMs. vTPMs expose private keys when they are used by signing or decrypting software since they have to be first loaded into the platform memory.

For vTPMs and vpTPMs, the contents of PCRs need to be put into platform memory before processing. Hence, in both vTPMs and vpTPMs, the integrity of PCRs values cannot be guaranteed since they can tampered by the attacker in the platform memory. For pTPMs, the PCR Reset command, which initializes PCR values, can guarantee the integrity of newly initialized PCR values (i.e., they must be zero after initialization). However, the PCR Extend command in pTPMs cannot guarantee the integrity of extended PCR values since it may use an argument that has already been tampered before it is sent from the platform memory into pTPMs.

Although vpTPMs are not as secure as pTPMs with respect to the PCR Reset command, we can argue that this is not a problem for security of applications. This is because PCR values always need to be extended after initialization when they are used in applications and the PCR Extend command in vpTPMs and pTPMs are at the same security level in our attack model, as discussed above. Apparently, vTPMs are not as secure as pTPMs and vpTPMs since the secrecy of private keys are not guaranteed in our attack model.

At last, we mention that pTPMs and vTPMs do not have good usability as virtual TPM in our TPM cloud. The usability problem was discussed in Sect. 1. For example, even if a host has a physical TPM and the correct driver, a Javascript program running in a Web browser cannot access the TPM functionality on the same host due to the interoperability problem between TSS and Javascript, and the security limitation to Javascript enforced by the Web browser. Hence, the NS protocol cannot be implemented with pTPMs and software-based vTPMs.

7. Related Work

The system of vTPMs [16] provides virtual TPMs for virtual machines on a single hardware platform. The vTPMs are implemented in software that is running in the host memory as user-space processes or in a secure co-processor. A vTPM has its own virtual EK and virtual SRK. However, all keys of vTPMs are created not by the physical TPM, instead by software. On the contrary, our TPM cloud depends on physical TPMs to generate and manipulate keys. The benefit is that private keys are always protected by TPMs even when the TPM cloud is intruded by malware. More importantly, the system of vTPM and other work of TPM virtualization [18], [19] cannot address all the usability problems of TPMs discussed in the first section.

A software-based TPM (SW-TPM) [6] is developed for resource-constrained embedded systems because they cannot afford the size and cost overheads of a separate TPM. Similarly, the work [20] proposes the customizable trusted modules for mobile phones to address the resource limitation of mobile phones when using trusted modules. The

TPM cloud can be applied to the mobile embedded systems if they can access network. Using the TPM cloud is also a way to reduce the energy overheads of embedded systems caused primarily by 2,048-bit RSA operations [6].

The work [21] deals with the complexity and inflexibility of TPM functionality by proposing the μTPM architecture. In this architecture, some TPM functionality is implemented outside the physical TPMs. On the one hand, the physical μTPMs do not have to be very complex to implement too much functionality. On the other hand, the functionality outside physical μTPMs can be changed more easily to get flexibility. Similarly, in our architecture of TPM cloud, we move some TPM functionality from physical TPMs to the cryptographic service module.

The Bind attestation service [22] has been used to perform fine attestation of applications such as several distributed computation applications and the implementation of Border Gateway routing Protocol (BGP). The Bind service needs the TPM on the platform running those applications. By redirecting the TPM commands (such as TPM_Extend, TPM_PCR_Reset, TPM_Seal and TPM_UnSeal given in [22]), the Bind service can de used on the platform without TPMs. This also applies to other TPM-based applications like [23], [24]. The remote attestation is a key component to build trusted distributed systems (such as computation grids) [25]. If a host in a distributed system does not have a physical TPM, the trust among hosts can still be built by relying on the TPM cloud, which can store and sign the measurements of the host applications.

The work [26], [27] analyzes how to build trust into cloud computing (in particular, how to build trusted virtual machine monitors). A typical architecture of cloud computing is a virtual machine monitor supporting many virtual machines for many users. So users can apply for computing resources in an on-demand, pay-as-you-go manner. The TPM cloud can be used together with the mechanisms in [26], [27] to attest the integrity of virtual machines in cloud computing. To attest the integrity of a virtual machine, the underlying virtual machine monitor, whose integrity is guaranteed by the mechanisms in [26], [27], can hash the image of the virtual machine before launching it and then extend the hash value into a virtual TPM in the TPM cloud.

The virtual smart cards [28], [29] are proposed as a secure and convenient way of managing private keys. By using virtual smart cards, users store their private keys in the Trust Provider [28] or in the Virtual Smart Card Server [29], which are the only places where the private keys can be used for making signature and decryption. To use their own private keys, users need to authenticate to the the Trust Provider or the Virtual Smart Card Server by using some second authentication factor. The TPM cloud is similar to virtual smart cards in the style of managing private keys and authenticating users with second factors. However, the private keys in TPM cloud are used within the physical TPMs, rather than by software running on a server as in the virtual smart card systems. Moreover, the TPM cloud has other functionality, such as the management of platform measurement with virtual PCRs.

8. Conclusion

TPM has been applied to build trusted systems. However, it has been limited in its use and adoption as we analyzed before. We proposed the TPM cloud to promote acceptance of TPM functionality. From the TPM cloud, users can apply for their own TPM instances on demand. It allows users to benefit from the security properties of TPM without owning a physical TPM. Moreover, the TPM functionality in the cloud is easy for users to access when they move to platforms not owned by them. The TPM functionality also can be easily integrated into applications developed in various languages without the interoperability problem. The embedded devices can use the TPM functionality in the cloud to reduce the energy overheads caused by physical TPMs.

We formalized the functionality of the TPM cloud, and made it clear how cloud commands are executed. Some possible improvements to the TPM specification are proposed to make TPMs more useful in various usage contexts. We demonstrated the functionality of the TPM cloud by using it to implement Needham-Schroeder public-key protocol for web authentication. This example showed that using the TPM cloud is a viable way to incorporate strong security into high-level applications. At last, we discussed the establishment of trusted chains by using the TPM cloud and analyzed the security properties of virtual TPMs in the TPM cloud, concluding that the virtual TPMs in the TPM cloud are as secure as physical TPMs in practical applications in our attack model.

References

[1] Trusted Computing Group. http://www.trustedcomputinggroup.org

[2] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," SSYM'04: Proc. 13th Conference on USENIX Security Symposium, p.16, 2004.

[3] The TCG Software Stack (TSS) Specification. http://www.trustedcomputinggroup.org

[4] D. Fisher, J.M. McClune, and A.D.J. Andrews, "Trust and trusted computing platforms," Tech. Rep. 642, Carnegie Mellon University, 2011. http://repository.cmu.edu/sei/642

[5] IBM BladeCenter. http://www-03.ibm.com/systems/bladecenter

[6] N. Aaraj, A. Raghunathan, and N.K. Jha, "Analysis and design of a hardware/software trusted platform module for embedded systems," ACM Trans. Embed. Comput. Syst., vol.8, no.1, pp.1–31, 2008.

[7] M. Strasser, H. Stamer, and J. Molina, "Software-based TPM Emulator for Unix," http://tpm-emulator.berlios.de/

[8] The Open-Source TCG Software Stack. http://trousers.sourceforge.net

[9] IAIK at TU Graz, "Trusted Computing for the Java Platform. http://trustedjava.sourceforge.net/"

[10] Infrastructure as a service. http://en.wikipedia.org/wiki/Infrastructure_as_a_service

[11] P.C. van Oorschot, "System security, platform security and usability," Proc. Fifth ACM Workshop on Scalable Trusted Computing, STC '10, New York, NY, USA, pp.1–2, ACM, 2010.

[12] R.W. Reeder, L. Bauer, L.F. Cranor, M.K. Reiter, and K. Vaniea, "More than skin deep: Measuring effects of the underlying model on

access-control system usability," CHI 2011: Conference on Human Factors in Computing Systems, May 2011.

[13] D. Liu, J. Lee, J. Jang, S. Nepal, and J. Zic, "A cloud architecture of virtual trusted platform modules," 6th International Symposium on Trusted Computing and Communications, pp.804–841, 2010.

[14] G. Lowe, "An attack on the needham-schroeder public-key authentication protocol," Inf. Process. Lett., vol.56, no.3, pp.131–133, 1995.

[15] Apache Axis2, "http://ws.apache.org/axis2/"

[16] S. Berger, R. Cáceres, K.A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: virtualizing the trusted platform module," USENIX-SS'06: Proc. 15th Conference on USENIX Security Symposium, 2006.

[17] S. Bratus, N. D'Cunha, E. Sparks, and S.W. Smith, "TOCTOU, traps, and trusted computing," Proc. 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications, Trust '08, pp.14–32, 2008.

[18] V. Scarlata1, C. Rozas, M. Wiseman, D. Grawrock, and C. Vishik, TPM Virtualization: Building a General Framework, pp.43–56, Springer, 2008.

[19] A.R. Sadeghi, C. Stüble, and M. Winandy, "Property-based tpm virtualization," 11th International Conference on Information Security, pp.1–16, 2008.

[20] K. Dietrich and J. Winter, "Towards customizable, application specific mobile trusted modules," Proc. Fifth ACM Workshop on Scalable Trusted Computing, STC '10, pp.31–40, 2010.

[21] K. Kursawe and D. Schellekens, "Flexible μTPMs through disembedding," Proc. 4th International Symposium on Information, Computer, and Communications Security, pp.116–124, 2009.

[22] E. Shi, A. Perrig, and L.V. Doorn, "Bind: A fine-grained attestation service for secure distributed systems," SP '05: Proc. 2005 IEEE Symposium on Security and Privacy, pp.154–168, 2005.

[23] R. Sandhu and X. Zhang, "Peer-to-peer access control architecture using trusted computing technology," SACMAT '05: Proc. Tenth ACM Symposium on Access Control Models and Technologies, pp.147–158, 2005.

[24] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: Policy-reduced integrity measurement architecture," SACMAT '06: Proc. Eleventh ACM Symposium on Access Control Models and Technologies, pp.19–28, 2006.

[25] C.N.A.M. Jun Ho Huh, John Lyle, "Managing application whitelists in trusted distributed systems," Future Generation Computer Systems, vol.27, no.2, pp.211–226, Feb. 2011.

[26] N. Santos, K.P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," Proc. 2009 Conference on Hot Topics in Cloud Computing, HotCloud'09, 2009.

[27] A.M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N.C. Skalsky, "Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity," Proc. 17th ACM Conference on Computer and Communications Security, CCS '10, pp.38–49, 2010.

[28] G. Forget and A. Stervinou, "The virtual smart card," Card Technology Today, vol.19, no.7-8, pp.211–226, Aug. 2007.

[29] L. Smith and R. Levenberg, "Virtual Smart Card System and Method," US Patent: US7890767 B2. Feb. 2011.

Dongxi Liu is a Research Scientist in CSIRO ICT Centre, Australia, since March, 2008. His research interests include programming languages and formal methods, and their applications to the areas like rigorously trusted computing, information security, cloud computing, XML view update. He has an invention on homomorphic encryption scheme applied to encrypted database query. Before joining CSIRO, he worked as a researcher in the University of Tokyo from Feb., 2004, and worked as a research fellow in National University of Singapore from Dec., 2002. He got his Ph.D. in computer science and engineering from Shanghai Jiao Tong University in 2003, and both his M.E. and B.E. from Taiyuan University of Technology in 1999 and 1996, respectively.

Jack Lee graduated from the University of Sydney in 2010, with a B.Sc. (Honours) in Pure Mathematics. He is currently working for the Department of Defence, Australia.

Julian Jang has been working as a research scientist at CSIRO ICT Centre since 2002. Her research interests compasses from building robust automated business processes for large-scale distributed applications, providing hardware-based security solutions in web-based applications, and recently developing network intrusion detection system using the concept from artificial immune system. Prior to join CSIRO, she worked as a java developer and a database administrator. She has Ph.D. and Maters degrees from University of Sydney.

Surya Nepal is a Principal Research Scientist working on Service and Cloud Computing, more specifically trust and security aspects of collaboration at CSIRO ICT Centre. His main research interest is in the development and implementation of technologies in the area of Service Oriented Architectures (SOA) and Web Services. He obtained his B.E. from Regional Engineering college, Surat, India, M.E. from Asian Institute of Technology, Bangkok, Thailand and Ph.D. from RMIT University, Australia. At CSIRO Surya undertook research in the area of multimedia databases, Web services and service oriented architectures, and security and trust in collaborative environment. He has also edited a book on "Managing Multimedia Semantics". He is a program committee member of major international conferences in cloud computing and service computing such as IEEE Cloud, Cloud Computing, ICSOC, CCGRID and WISE. He has several journal and conference papers in these areas to his credit.

John Zic is currently a research team leader and science specialist for Trustworthy Systems within the ICT Centre, CSIRO and holds visiting positions the University of New South Wales and at Macquarie University. He is a co-inventor of the world's first portable USB-based trusted computing platform to incorporate a TPM cryptographic microcontroller: the patented Trust Extension Device. He has recently given invited presentations and keynotes at the EU FP7 INCO-TRUST workshop in New York, MIT Kerberos Consortium Conference, Vanguard/TTI CyberINsecurity Conference and participated in the 11th Joint EU and Australian Science and Technology Cooperation Committee. John was a member of the Australian Academy of Technological Sciences and Engineering Working Group on Cloud Computing from 2009–2010, advising on privacy, security and trust. Prior to the current position, John was the Research Director for the Networking Technologies Laboratory for a period of two years, during which time he oversaw the second Centre for Networking Technologies for the Information Economy project (CeNTIE 2). This project was funded by the Australian Government through the then Department of Communications, Information Technologies and the Arts (DCITA) that brought in some $3M p.a. into the ICT Centre. John has previously held research positions at Motorola's Australian Research Centre from 1999 to 2003, and has been a faculty member from 1982 to 1999 at both the University of New South Wales and Sydney University.