Towards a Programmable TPM

Paul England and Talha Tariq

Microsoft Corporation, 1 Microsoft Way,
Redmond WA 98052, USA
{paul.england,talhat}@microsoft.com

Abstract. We explore a new model for trusted computing in which an existing fixed-function Trusted Platform Module (TPM) is coupled with user application code running on a programmable smart card. We will show that with appropriate coupling the resulting system approximates a "field-programmable TPM." A true field-programmable TPM would provide higher levels of security for user-functions that would otherwise need to execute in host software. Our coupling architecture supports many (but not all) of the security requirements and applications scenarios that you would expect of a programmable TPM, but has the advantage that it can be deployed using existing technology.

This paper describes our TPM-smart card coupling architecture and the services that we have prototyped. The services include: (1) An implementation of count-limited objects in which keys can only be used a preset number of times. (2) More flexible versions of the TPM *Unseal* and *Unbind* primitives that allow sealing to groups of equivalent configurations. And (3) a version of *Quote* that uses alternative signature formats and cryptography available within smart cards but not in the TPM itself.

We also describe the limitations of the coupling architecture and how some of the limitations could be overcome with a true programmable TPM.

Keywords: Trusted Platforms, Trusted Platform Module, Smart Cards, Secure Execution.

1 Introduction

Trusted Platform Modules (TPMs) are fixed-function security processors built into many computer platforms [1]. When combined with Core- and Dynamic-Root-of-Trust-Measurement facilities (CRTM & DRTM) for reporting platform state, the TPM provides a basis for a secure and attestable execution environment for system software and applications.

The TPM provides a variety of services [2] that depend on the platform state. These include:

> *Attestation*: Cryptographic reporting of platform state to a remote challenger.
>
> *Sealing*: Protected storage / encryption of data that will only be released / decrypted when the platform is in a particular configuration and state.

When these services are combined with a secure software stack, the small set of TPM-provided functions can bootstrap rich and powerful execution environments running on the main processors.

L. Chen, C.J. Mitchell, and A. Martin (Eds.): Trust 2009, LNCS 5471, pp. 1–13, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Using the TPM to bootstrap trust into an execution environment like a platform hypervisor or operating system is adequate for many purposes, however data and application programs running on the main processors are much less protected from physical attack than programs and data held inside the TPM. This problem is evident from the recent hardware attacks on applications utilizing TPMs [3], [4], [5]. The problems of software robustness are even more challenging: mainstream operating systems have an ill-defined Trusted Computing Base (TCB) that is generally not secure enough for attestation to be meaningful [6].

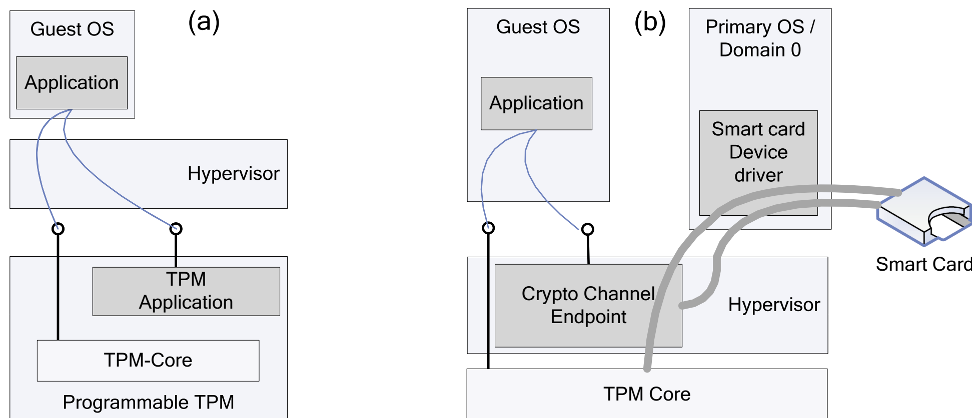

Fig. 1. (a) Schematic illustration of a programmable TPM and its use in a hypervisor setting. We assume that the TPM can load and run applications and the services implemented can be exposed to the hypervisor and guest operating systems. (b) Schematic of one instantiation of our coupled TPM smart card architecture: The TCB and TPM are coupled to the smart card using an out-of-band cryptographic marrying step. The cryptographic channels (thick lines) represent authenticated and secure connections from the smart card to the TPM and the smart card to the channel end-point.

If the host-platform-based secure environment is not secure enough, we might consider building a secure execution environment *inside* a TPM such as that illustrated in Fig 1(a). Such a system should provide much better robustness to hardware and software attack than that offered by platform macrocode. And indeed, such devices have been studied by researchers, but unfortunately they do not yet exist [7].

In this paper we propose an alternative architecture for providing high-assurance extended TPM services. Instead of making changes to the TPM hardware design, we describe and evaluate an architecture in which we *couple* a programmable smart card to a TPM to provide programmable services that are not possible with either alone. See Fig. 1 (b). This architecture has many interesting characteristics: First it is a practical way of providing enhanced security functionality for existing TPMs. Second, it provides a way of prototyping new TPM functions to assess their usefulness before committing them to silicon, and finally it allows us to explore the design and assess the usefulness of a true “programmable TPM.”

We have built several advanced security services to help us understand this architecture and demonstrate its capabilities. The services described in this paper are:

Count-Limited Objects: An implementation of TPM keys that can only be used to perform cryptographic operations a preset number of times. This capability is designed to simplify some aspects of key revocation and support rights-management.

Flexible *Seal* and *Unseal*: An implementation of the *Seal*, *Unseal,* and *Unbind* primitives that allow more complex policy expressions than the simple Platform Configuration Register (PCR) equality checks supported by the current TPM specifications[1]. One policy expression allows sealing to a software publisher or other authority identified by a public key. In this case the publisher may later authorize any PCR configuration using a certificate signed using the associated private key. Another policy expression allows more complex logical expressions of authorized configurations (*e.g.* PCR configuration 1 *or* PCR configuration 2). Both of these enhancements are designed to make software updates and grouping of equivalent programs easier to manage.

Attestation Translation: A smart card service that provides attestation using cryptography and signature formats unavailable within a TPM. This is a proof of concept of attestation translation: a more sophisticated implementation could provide platform attestation in more widely used signature and certificate formats like X.509. This facility should simplify the deployment of attestation because existing servers and protocols can be used.

The paper is organized as follows. In section 2 we describe the coupling architecture. In section 3 we describe the security primitives that we have prototyped, and in sections 4 and 5 we describe the limitations of the coupling architecture and possible further work.

2 TPM to Smart Card Coupling

We seek to approximate a field-programmable TPM in which the secure execution environment for user extensible application programs is *part of* the host platform TPM. However when emulating a programmable TPM using a conventional fixed-function TPM and an external smart card, the secure execution environment is external to the TPM, is independent of the host state, and can be freely roamed between machines. This creates several challenges that we need to address: First, the execution environments provided by the host system and a smart card are relatively independent. For example, smart card applications can still run if the smart card is moved between different host machines. Second, TPM-to-host-TCB communications are relatively well protected (for example TPM communications are on a motherboard internal bus) whereas in coupling with an external smart card, the smart card bus is exposed, and communications are sometimes managed by drivers running outside the TCB (Fig. 1(b)).

2.1 Coupling Security Requirements

If the smart card is to provide TPM-enhanced platform services we need to couple the smart card and the host platform more tightly.

[1] At the time of this writing the current TPM specifications is version 1.2.

In particular we identify the following security requirements:

1) Smart card applications should be able to determine the host hardware and the host TCB (*e.g*. to support a smart card enhanced version of *Unseal* or *Quote*.)
2) The host TCB should be able determine the identity of the smart card and its applications (*e.g*. to ensure that TCB confidential data is not improperly released to an un-trusted smart.)
3) The smart card applications should have a bi-directional confidential channel to the host. (*e.g*. to support confidential communication of data passed to a smart card enhanced version of *Seal* or confidential communication of data returned from a smart card enhanced version of *Unseal*).

Our solution to these requirements is described in the next section.

2.2 Cryptographic Marrying (Smart Card to TPM Binding)

We assume that a trusted authority determines the TPM-to-smart card binding policy. In the case of an enterprise this might involve an IT department coupling an employee's smart card with the TPM on her PC (either under conditions of physical security, or remotely given knowledge of keys in the devices to be coupled). In the case of an OEM this might involve shipping a pre-coupled TPM and smart card together with an associated platform certificate.

We have implemented a system in which a unique TPM is coupled with a single smart card, but generalizations are straightforward. During this platform binding step we generate and store the following cryptographic keys to identify the smart card and associated TPM:

- The TPM generates an Attestation Identity Key (AIK) which is used to identify the TPM and the host. The public portion of this key is communicated to the smart card under conditions of physical security in the marrying step, and is stored in smart card non-volatile storage as shown in Fig. 2.
- The smart card generates an RSA key pair which is used to identify the card. The public portion of the key is communicated to the platform TCB under conditions of physical security and is secured in host platform secure storage.

The binding and initialization step need only be performed once. At run time code in the TCB and in the smart card builds a secure authenticated channel based on these authentication keys.

The explicit software-constructed secure channel is sufficient to support secure communication between smart card applications and the trusted computing base. Some of our smart card applications additionally employ cryptographic properties of the TPM itself without need for further channel security (beyond boot-strapping with the married AIK). First, the attestation translation and enhanced seal operations need to determine the current platform configuration. We provide this proof using the *TPM_Quote* operation (using the married AIK). This cryptographic primitive is already designed to work securely in the face of untrustworthy host software. Second, the count-limited key function uses the TPMs HMAC-based

Fig 2. Cryptographic Binding of the TPM and Smart Card

proof-of-password-possession protocols. Finally, the count-limited key function uses the TPM capability for remote creation and encryption of keys using the TPM key-storage hierarchy.

In our implementation two more smart card keys are created as part of the marrying step. A symmetric AES key is created for off-card storage of smart card created sealed data blobs, and an RSA signing key is created for the attestation translation function. If a new binding is performed all previous bound data becomes inaccessible and the quote translation key is destroyed (just like installing a new owner in a TPM).

We also need integrity protected host storage for the host TCB to store the married smart card public key. Since the security model for trusted computing does not generally assume storage is trustworthy unless protected by cryptography, we use the host *TPM_Seal* primitive to integrity protect the married smart card public key. The smart card has genuine access-protected storage so no cryptographic measures are needed.

Our current implementation assumes that the smart card applications are loaded prior to the marrying step. A more sophisticated version would provide a user-accessible smart card execution environment and services that let the smart card applications authenticate themselves (see section 5).

Our architecture is generic and independent of the nature of host software: it can be applied to systems that employ a hypervisor, an operating system without a hypervisor, or applications running directly on the computer hardware. From our perspective all that differs is the nature of the TCB and the PCRs that are used to identify the platform state. Of course the choice of trusted computing base has practical implications for the comprehensibility and relevance of host PCR values [6].

3 Smart Card Enhanced TPM Security Primitives

In this section we describe the implementation of three security primitives that demonstrate the possibilities of the TPM to smart card coupling architecture.[2] Note that the smart card functions appear somewhat hard to use. This is because we generally favor performing only essential security functions in the smart card (which is slow

[2] These experiments used a Dell Optiplex 745 running Vista SP1 and containing an Atmel TPM version 1.2 with firmware version 13.9. The smart card was a Gemalto.NET v2 card with 80Kbyte of memory for code and data.

and hard to debug) with other logic and complex data structure creation being performed by host software.

3.1 Count Limited Objects

Count-limited key objects are keys that can only be used a preset number of times [9]. The TPM provides monotonic counters that external software can use to decide whether a key should be used, but - as we have already observed - the attack resistance of host software is low. Our implementation of count-limited keys uses a key on the TPM and a use-counter on the smart card and no external software is involved in authorizing the use of the key.

The design is as follows: The smart card creates a TPM key and sets the key use authorization (the *useAuth*) to a random value (the TPM will not use the key for cryptographic operations unless the requestor proves knowledge of the *useAuth* value). The smart card exports the key as a blob encrypted so that it can only be decrypted by the married TPM. The smart card associates the (secret) *useAuth*-value with an internal counter, and will only authorize use of the key a preset number of times. When the count is exceeded the key can no longer be used.

TPM key use is authorized by means of an HMAC-based protocol that does not reveal the *useAuth* authorization data in plain-text and is replay resistant (with some assumptions – see [8]). This means that host software cannot use keys without the cooperation of the smart card.

In more detail, the smart card exposes a pair of functions to support this functionality. *CreateCountLimitedKey* creates a key with a random *useAuth* value and exports it encrypted so that it can only be loaded into the married TPM. This function also creates a counter set to the maximum number of uses and associates it with the freshly created secret *useAuth*. Later, host software can load the key into the TPM and ask the smart card to provide authorization for its use through *GetCountLimitedUseAuth*. In this function the smart card decrements and then checks that the counter limit has not been exceeded. If not, the requested command is authorized.

The command pseudo-code is as follows:

CreateCountLimitedKey

Input:

A TPM parent storage public key p,
Algorithm parameters for the key to be created a,
The number of times the key can be used n

Output:

An encrypted key that can be loaded into a TPM, an identifier for the counter c

Actions:

1) Create a new RSA key with parameters supplied
2) Create a new counter set to value n
3) Create a new random useAuth value a for the key and associate it with the counter
4) Encode the RSA key and useAuth into a TPM key structure then encrypt with the provided TPM parent public key
5) Return the key blob and an identifier for the counter

GetCountLimitedUseAuth

Input:

The TPM command string to be authorized s,
The counter identifier c

Output:

20 byte authorization value for the supplied command string or an error

Actions:

1) Decrement the counter c. If the counter does not exist or the count value is less than zero return an error
2) Return the HMAC of the command string s using the authorization secret associated with the counter

Some scenarios demand that it be proven that count-limited keys are created under conditions of physical security. For instance in our simple implementation it is not possible to prove to an outside party that the key is indeed count-limited. There are many variations of the simple design that overcome this shortcoming: E.g. rather than creating the key inside the smart card it could be created on a secure server (or a Host Security Module) and the count limit and *useAuth* data could be separately communicated to the smart card. Alternatively the smart card could certify the key that it created.

Our counters share some of the features of the implementation of TPM-supported monotonic counters proposed by *Sarmenta et al*. [9]. In particular our counters can be used as part of the authorization policy for key or other object use.

3.2 Flexible *Sealing* and *Binding*

Sealing encrypts data together with a tag indicating some expected future platform state encoded in PCR values. The related *Unseal* function will only decrypt and reveal the data if the platform is in the pre-authorized state. *Sealing* is a powerful feature of the TPM, but unfortunately it is often hard to predict future configurations because of unexpected changes in the platform configuration and state. The *sealing* capability (and related capabilities for associating keys with PCR states, *Unbinding*, etc.) would be easier to use if the TPM had more flexibility in the expression of authorized configurations.

We have extended the simple TCG binding model to provide more powerful sealing policy specifications using code implemented on the smart card. The cases we have implemented are:

- Sealing and binding to any one of a list of PCR configurations.
- Sealing and binding to a public key so that the key owner can later authorize any PCR configuration with a signed certificate.

In the latter case, when an *Unseal* or *Unbind* operation is attempted, the caller must also provide a valid digital certificate authorizing the current configuration from the policy-associated signature authority.

In both cases the smart card must check that the current married TPM PCR values represent a state authorized by the sealer. In our implementation the smart card

performs this check in the same way that any other remote entity would determine the platform configuration: *i.e*. the smart card demands that host software provide evidence for the current state by means of the output of a *Quote* operation using the married AIK and a smart card provided nonce (to prevent replay). If host software can respond with evidence of an authorized configuration, the smart card will release the sealed data to the TCB.

We describe the smart card operations that support the *Seal* and *Unseal* implementations; *Unbind* is similar to *Unseal*.

Sealing to a List of PCR Configurations

The following smart card functions support *sealing* to a list of configurations. *SealToConfigurationList* is the smart card function that protects the data. The sealer need only specify the hash of the list of authorized configurations at this stage. *UnsealConfigurationList* is the corresponding unseal function. Here the caller must specify the whole configuration list (which the smart card will hash to ensure it matches the specified policy) and the policy element number that the smart card should attempt to satisfy. The smart card must also be given proof of the current platform configuration by means of the output of a *TPM_Quote* operation over the relevant PCRs. Replay resistance for the quoted configuration is provided by a smart card provided nonce, which must be obtained using the smart card *GetNonce* function.

In more detail, the pseudo-code for the commands follows:

SealToConfigurationList

Input:

A secret s,
the hash of a list of authorized configurations l

Output:

A sealed encrypted blob

Actions:

Integrity-protect and encrypt the concatenation of s, and l

GetNonce

Input:

None

Output:

A 20 byte random nonce

Actions:

Create and return a random nonce

UnsealConfigurationList

Inputs:

A sealed blob, b
The expected configuration list, l
The list element number that we expect to satisfy, i
The output of a TPM *Quote* on the current configuration, q

Outputs:

The previously sealed data or an error

Actions:

1) Decrypt the sealed blob *b* returning the secret *s* and the policy list hash *h*
2) Check that the hash of *l* matches the policy hash *h*
3) Check that the TPM signature q is formed signature using the married AIK over the *l[i]* (the configuration element that we expect to match) and the previously supplied nonce
4) Return the secret data *s* if all of the above checks succeed, else return an error

Sealing to a Configuration Authorized by a Public Key

The following smart card functions support *sealing* to PCR configurations authorized by a public key. *SealToPublicKey* encrypts a secret and the public key of an entity trusted to authorize future platform configurations. *UnsealPublicKey* is the corresponding unseal function. *UnsealPublicKey* must be provided with the original sealed blob and a signed statement from policy key holder authorizing a PCR configuration. The caller must also provide the result of a *TPM_Quote* operation that proves compliance with the specified configuration. If policy compliance is proven the sealed data is released. As before, the caller must obtain a fresh nonce from the smart card and have it incorporated into the Quoted data structure.

In pseudo code:

SealToPublicKey

Input:

A secret *s*,
A public key *k*

Output:

An encrypted blob

Actions:

Integrity-protect and encrypt the concatenation of *s,* and *k*

GetNonce

See above

UnsealPublicKey

Inputs:

A bound blob *b*,
An authorized PCR configuration from the server *c*,
A signature over the authorized PCR configuration from the server *S,*
The output of a TPM Quote operation *q*

Outputs:

The sealed data or an error

Actions:

1) Decrypt the sealed blob *b* returning the secret *s* and the public key *k*
2) Validate that the signature *S* is valid for the configuration *c* using the public key *k*

3) Validate that q is a TPM signature over the configuration c using the married TPM AIK and the expected nonce
4) Return the secret s if the above checks succeed, else return an error

One detail is omitted from the description above. The TPM implementation of *Seal* records PCR values at the time of *sealing* for the purposes of source platform and configuration authentication. Our implementation of *Seal* also takes the output of a *Quote* operation over a smart card provided nonce to provide similar capabilities.

All data communicated between the TCB and the smart card is passed over the secure channel described in section 2. The channel endpoints are authenticated using the married TPM and smart card keys.

3.3 Enhanced *Quotes*

The *TPM_Quote* operation creates a signature using an AIK over a data structure that includes TPM internal state as reflected in PCR values, and externally provided data (for freshness, or to associate the configuration with some other cryptographic object). This building block is designed to be used in cryptographic protocols that prove knowledge of the AIK *and* prove the current platform state. Unfortunately the TPM signature format is non-standard, and this is one of the things that has made it difficult to adopt TPM attestation technology.

We have prototyped a smart card function that translates the platform configuration provided by the TPM into another format. Our proof of concept also uses non-standard data structures, but a more sophisticated implementation would use a certificate format like X.509. Such certificates could be used for network access control, or in an email or document signing scenario to prove the machine and machine configuration when the document was signed [11],[12].

Our configuration rewriting function is called *TranslateQuote*. It must be called with fresh evidence of the current configuration by means of the output of the *TPM_Quote* operation over a smart card nonce. *TranslateQuote* checks the quote signature is properly formed and is issued by the married AIK. If both conditions hold, the smart card generates a signature over the TPM-specified state, and an external nonce.

In pseudo code:

GetNonce

See above

TranslateQuote

Inputs:

The data structure supplied to the *TPM_Quote* operation q,
The TPM-quoted signature s over this data structure and the previously obtained nonce,
External data to sign d

Outputs:

A smart card created signature or an error

Actions:

1) Check that *s* is a valid signature over the data *q* and the nonce using the married AIK
2) If the check succeeds return a smart card signature over a translation of q and the external data d, else return an error

4 Programmable TPMs

Our coupling architecture strikes a useful balance between flexibility and deployability using today's generally available commodity hardware since it requires no modification to the current specifications of the TPM[3] and uses general purpose programmable smart cards, but it is interesting to speculate on the design and improved functionality of a future programmable TPM.

There are many possible models for a programmable TPM. Useful starting places include multi-application programmable Java or .Net smart cards, or the Trusted Execution Model (TEM) described by *Costan et al*. [13]. Perhaps the simplest conceptual design for a programmable TPM is to replicate the security model for code executing *outside* the TPM but applied to user-code running *inside* the TPM. The TPM already has a model for authenticating security modules executing outside, which is the notion of locality coupled with the DRTM launch procedure [14]. This secure late-launch procedure has been used by the Oslo project [15] and Flicker [16]. Applying this idea to an execution environment *inside* the TPM would involve the definition of a new locality for access by TPM applications, and new PCR-registers dedicated to their measurements. See Fig. 3. However, beyond privileges associated with access locality, internal TPM applications would have the same access to other TPM functions and keys as applications running outside the TPM.

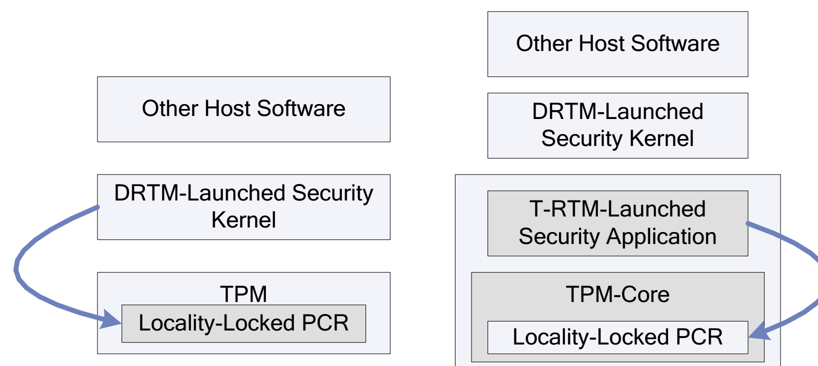

Fig. 3. Left: A TPM supporting a DRTM-launched security kernel. The DRTM procedure and platform hardware and firmware ensures that a special PCR contains a reliable measurement of the external security kernel, and that the TPM can authenticate commands originating from the security kernel. Right: Applying this model to a programmable TPM would define a new locality and associated "TPM-Root-of-Trust-of-Measurement" (T-RTM) to hold measurement of the TPM internal programmable security functions.

[3] As of this writing the current version of TPM Specifications is 1.2.

Unfortunately there are limits to the types of functions that can be supported using this sort of programmable TPM because TPM protected data is not accessible to the user applications. So while it would be possible to create a new class of storage key with sophisticated migration features with this design, it would not be possible to provide this migration capability to the storage root key (SRK) because the SRK private data is inaccessible. Allowing third party code access to TPM private keys and other protected data changes the TPM security model profoundly, so we generally prefer designs that supplement existing functionality rather than replacing or modifying it.

5 Conclusions and Future Work

We currently only support applications that are pre-loaded onto the card prior to TPM card marrying. This is probably adequate for most enterprise use (the enterprise will load line-of-business applications onto the smart card prior to issuance) but does not exercise the full potential of the coupling architecture.

To go beyond pre-loaded applications we must provide an isolated execution environment for applications in the smart card, *and* provide a means for these applications to authenticate themselves to the host computer. The isolation and authentication primitives are necessary because we can no longer necessarily trust the applications running on the card. This seems most straightforwardly solved by re-applying the principles of authenticated operation *but within the smart card*. In particular we would need to modify the smart card application loader to measure and record the application digest (or other authentication data) and provide the smart card application with sealing and attestation services. Smart card applications could use these primitives to prove to the platform TCB that it is communicating with a trustworthy card *and* card application.

Delivering the promise of Trusted Computing has been delayed by a number of problems. These include the relative unavailability of mainstream operating systems and hypervisors with useful security properties, problems balancing the high levels of security provided by the TPM and ease of management, and problems using the TPM to enhance existing security applications and scenarios. Our work demonstrates that logic and cryptographic operations running on a smart card coupled with the host platform and TPM can mitigate all of these issues, and is also an interesting prototyping environment for experimenting with new functionality that could be incorporated into future TPM designs.

The three applications we implemented were chosen to exercise local- and remote-trust verification, and to mitigate some of the problems that the authors have experienced in trying to apply trusted computing to real problems. Other candidate applications included keys with more sophisticated key management and migration functions, a software-TPM on the smart card, a “roaming-TPM” for use in an enterprise, and general experimentation on the correct definition of security primitives for future TPM designs.

References

1. Trusted Computing Group TPM Specification Version 1.2 Revision 103 (2007), `https://www.trustedcomputinggroup.org/specs/TPM/`
2. England, P., Peinado, M.: Authenticated operation of open computing devices. In: Batten, L.M., Seberry, J. (eds.) ACISP 2002. LNCS, vol. 2384, pp. 346–361. Springer, Heidelberg (2002)
3. Sparks, E.R.: A Security Assesment of Trusted Platform Modules. Dartmouth College, Technical Report. TR2007-597
4. Halderman, J.A., et al.: Lest We Remember: Cold Boot Attacks on Encryption Keys. In: Proc. 2008 USENIX Security Symposium (2008)
5. Bruschi, D., et al.: Attacking a Trusted Computing Platform. Improving the Security of the TCG Specification. Technical Report. Università degli Studi di Milano. Milan (2005)
6. England, P.: Practical Techniques for Operating System Attestation. Proceedings of Trust (2008)
7. Costan, V., et al.: The Trusted Execution Module: Commodity General-Purpose Trusted Computing. In: Eighth Smart Card Research and Advanced Application Conference
8. Offline dictionary attack on TCG TPM weak authorisation data, and solution. In: Chen, L., Ryan, M.D., Grawrock, D., Reimer, H., Sadeghi, A., Vishik, C. (eds.): Future of Trust in Computing, Vieweg & Teubner, 2008 (2008)
9. Sarmenta, L.F., et al.: Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS (Extended Version), Mit Technical Report MIT-CSAIL-TR-2006-064 (2006)
10. George, P.: User Authentication with Smart Cards in Trusted Computing. In: Arabnia, H.R., Aissi, S., Mun, Y. (eds.) Security and Management, SAM 2004, pp. 25–31. CSREA Press, Las Vegas (2004)
11. Balacheff, B., et al.: A trusted process to digitally sign a document. In: Proceedings of the 2001 workshop on New security paradigms. pp. 79–86 (2001) 1-58113-457-6
12. Giraud, J.-L., Rousseau, L.: Trust Relations in a Digital Signature System Based on a Smart Card. In: Proceedings of 23rd National Information Systems Security Conference, Baltimore
13. Costan, V.: The Trusted Execution Module Commodity General-Purpose Trusted Computing. In: The Eighth Smart Card Research and Advanced Application Conference
14. Grawrock, D.: The Intel Safer Computing Initiative: Building Blocks for Trusted Computing, 1st edn. Intel Press (2006) 0976483262
15. Kauer, B.: OSLO: Improving the Security of Trusted Computing. In: Proceedings of the 16th Usenix Security Symposium (2001)
16. McCune, J.M., et al.: Flicker: An Execution Infrastructure for TCB Minimization. In: Proceedings of the ACM European Conference on Computer Systems (EuroSys 2008) held in Glasgow (2008)