Flexible μ TPMs through Disembedding

Klaus Kursawe
Philips Research Europe
High Tech Campus 34
5656 AE Eindhoven
The Netherlands
klaus.kursawe@philips.com

Dries Schellekens
ESAT/SCD-COSIC and IBBT
Katholieke Universiteit Leuven
Kasteelpark Arenberg 10
3001 Heverlee, Belgium
dries.schellekens@esat.kuleuven.be

ABSTRACT

With the utilization of TPM-based trusted platforms in real applications, and the subsequent adaption of the specification to the experience gained from such utilization, it increasingly appears that the TPM architecture has some fundamental flaws that result in more and more complex and expensive hardware requirements. In this paper, we propose a new architecture that resets the trust boundary to a much smaller scale, thus allowing for much simpler and more flexible TPM implementations, without sacrificing the security gains from a classical TPM.

Categories and Subject Descriptors

C.2 [Computer Systems Organization]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; K.6.5 [Management of Computing and Information Systems]: Security and Protection

General Terms

Design, Measurement, Security

Keywords

Trusted Computing, secure coprocessor, Trusted Platform Module, Mobile Trusted Module

1. INTRODUCTION

Trusted Platform Modules (TPMs) were introduced by the Trusted Computing Platform Alliance (TCPA) to provide a low cost, universal building block on which platforms could build systems to provide a certain level of trust to the platform. This is done primarily by providing trustworthy measurement of the platform's boot sequence, as well as secure storage. Initially, TPMs were designed as separate chip; later versions have become integrated more closely into a platform's chipset.

Over time, the TPM specification has gained substantially in scope and complexity to accommodate different functional

requirements. A study by Sadeghi *et al.* [12] has revealed that some vendors already struggle with the corresponding implementation complexity; also, for the more cost sensitive mobile world, a different specification (the Mobile Trusted Module or MTM [9]) was required to keep the complexity to a manageable level.

In this, current TPM implementations have moved away from the original philosophy of the TCPA, namely, a simple, easy to implement and verify module that serves as a foundation for trust in the platform. In addition, thanks to increasing interest and utilization of TPMs, more requirements for additional functionality are generated, *e.g.*, support of multiple executions for virtualized machines [2], support for distributed protocols, etc.

In this paper, we investigate how TPM functionality can be achieved with simpler hardware, and with greater flexibility towards supporting specialized application demands. To this end, we re-investigate which of the functionality needs to be implemented inside the trust boundary (*i.e.*, in the trusted hardware), and which parts can safely be externalized onto the host platform. By externalizing large parts of the TPM implementation, we arrive at a hardware base that is smaller in size, more flexible to use, and simpler to implement and verify.

1.1 Related work

Chevallier-Mames *et al.* [4] propose a theoretical blueprint of a ROM-less smart card called Externalized Microprocessor (X μ P). All the executable code of the smart card is stored in the terminal and program instructions are fetched when needed. They propose a number of public key oriented alternatives to authenticate the externalized code: verification per instruction, batch verification with RSA screening, and verification of code sections. In the extended version of the paper [5] MAC based variants are given.

In order to embed a TPM in a reconfigurable System-on-Chip (SoC) design, Eisenbarth *et al.* [8] define a new FPGA architecture that includes a minimal root of trust called Bitstream Trust Engine (BTE). This trust component records the partial configuration bitstreams that are loaded on the reconfigurable logic, in Hardware Configuration Registers (HCRs), and limits access to the internal non-volatile memory (NVM) that stores the TPM's persistent state. Their solution allows field update of the TPM bitstream.

Recently, Costan *et al.* [6] proposed the Trusted Execution Module (TEM) as a more flexible alternative for TPMs. The TEM can execute arbitrary general purpose applications, split in executable fragments called closures. The closures are encrypted with the TEM's public encryption key,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASIACCS'09, March 10-12, 2009, Sydney, NSW, Australia
Copyright 2009 ACM 978-1-60558-394-5/09/03 ...\$5.00.

guaranteeing that only designated modules can run the application. The idea to split an application in atomic tasks is similar to our work. However, in this paper we focus on the orthogonal issue of verified execution, which proves to an external entity that a result is produced in a secure environment.

1.2 Structure of the paper

The paper is structured as follows. We start with a motivation for our alternative TPM architecture in Section 2. Section 3 describes the μ TPM scheme to disembed the TPM firmware. In Section 4 we compare the security of our μ TPM proposal to that of traditional TPMs. Section 5 discusses how to further minimize the μ TPM's memory footprint. We end with a conclusion in Section 6.

2. TOWARDS AN ALTERNATIVE TPM ARCHITECTURE

With the work on trusted computing progressing over the last ten years, the design of current TPM implementations is starting to show the first limitations. The primary problem is that the hardware is supporting both too much and too little functionality. In attempting to accommodate all requirements from different users of the technology, the “one-size-fits-it-all” approach towards TPMs has lead to huge implementations, making the hardware complex, hard to verify, and expensive. Each of those issues has already started to cause problems – a different specification was required for more cost-sensitive mobile devices [9], the complexity has been difficult to manage for some manufacturers [12], and governments keep worrying about verifiability of the security critical component [3]. With future specifications, this problem is likely to get only bigger rather than smaller – while few commands are likely to disappear, experience with TPMs in real platforms has lead to new requirements that future versions will have to accommodate (*e.g.*, cryptographic algorithm agility and advanced virtualization support).

We see that one way to approach these issues is to redefine the trust boundaries of the TPM architecture. By putting more functionality of a TPM outside of the trusted hardware, we increase flexibility (as outside mechanism can easily be adapted, and usually have more resources at their disposal than functionality implemented inside the TPM), and reduce the size of the critical hardware components we need to protect. To this end, we propose to remove the entire TPM code base – *i.e.*, the implementation of the TPM functionality – outside of the secure hardware. As with other TPM related storage, there is no security requirement to store this data inside the trusted hardware, as long as the TPM can authenticate it properly¹. The TPM trust boundary then only needs to incorporate the authentication key as well as enough RAM memory to store the command code during execution.

While this approach does require new mechanisms for code authentication and attestation, the functionality of the hardware can now be reduced to a very small number of commands, and future extensions to the functionality can be added without needing to modify the hardware specifica-

¹For protection of intellectual property, a producer may also want to encrypt the data; for the TPM security functionality this is not required though.

tion. More importantly though, externalizing the code from the hardware adds a new degree of flexibility that can handle most of the limitations current TPM implementations face:

Verifiability. Given that the TPM executes security relevant code, users (especially government related users) do want assurance that the TPM is implemented properly. By separating code and hardware, the hardware becomes substantially easier to verify, while it can be assured that the code has not been modified for a particular TPM; also, it is possible to separate code and hardware implementation, allowing for alternative implementations. If the standard proposes a generic programming language – *e.g.*, basing the TPM on a JavaCard [7] – it is even possible for pure software vendors or the open source community to provide different TPM implementations. In addition, users with special security need can use their own, individual code.

Customizability. TPM functionality becomes easier to extend or to limit. Current TPMs suffer from the fact that platform producers demand an increasing amount of functionality, while TPM manufacturers want to decrease the implementation complexity. With externalized commands, each platform can supply the commands needed for its particular operation, and omit functionality designed for different platforms; this even allows very special purpose commands in the TPM, or even freely programmable code. The only limit here is that guarantees given by the TPM according to the TCG specification – *e.g.*, the protection of certain keys – are never violated. This is assured by having one manufacturer authenticate their (external) firmware, and not allowing other processes to access the resources of that implementation.

Upgrades. TPM code can easily be updated in the field; the real hardware does not carry any functionality anymore, allowing to fix implementation bugs, retire unsafe protocols (such as SHA-1), or upgrading to newer versions of the TPM specification. To upgrade the TPM code, the old code simply can be replaced by (authenticated) new code. It must be taken care of though that the two versions cannot be mixed, as this may cause unpredictable behavior.

Specialized TPM implementations. Current TPMs are designed in a way that basically the same hardware is used in high-end servers and low-end embedded devices. In our proposed settings, low-end TPMs can implement the minimal hardware necessary to run, while high-end server TPMs can offer more memory, command cache and faster process management hardware to allow for rapid context switches. It is also possible for very low-end devices to build a TPM that does not have the resources for all commands in the TCG standard, but supports the commands required for the particular platform; while such a device can not claim TCG compliance, it can reuse hardware components and interface with TCG compliant software.

While the goal is to add flexibility to and simplify the hardware of the TPM functionality, one requirement for our

architecture is to allow for a functionality similar to current TPMs, *i.e.*, the trust model and functionality of the TPM specification must be compatible with the new hardware architecture. We extend the TPM requirements by allowing the architecture to support parallel, independent processes, *i.e.*, it is possible to run several different security co-processors in one hardware block without interference. This is relevant in the areas of virtualization, where one hardware TPM needs to support a number of virtual machines, each of which needs its own virtual TPM, or for the support of multiple TPMs in one platform acting on behalf of different stakeholders. The latter is for example proposed in the MTM specification, where separate logical TPMs are proposed to protect the mobile operator, the service provider and the end user, respectively. Allowing different logical TPMs with different code bases takes this approach further, even allowing individual applications (*e.g.*, a banking application) to have a dedicated, specialized trusted module.

As with classical TPMs, performance is not a priority – TPMs are not meant to work as cryptographic accelerators, but as slow and reliable building blocks. Nevertheless, the rather small communication bandwidth of current TPMs makes communicating large code segments a relatively inefficient operation. We would argue, though, that this is not a real problem; most frequently used commands are rather simple, and thus have relatively short implementations – in the open source TPM emulator [14], most commands can be implemented in few lines of C-code – with the notable exception of the DAA implementation, which, however, is so far not used in any real setting. As TPMs were never meant to perform fast, the additional delay is usually tolerable. An approach to speed up the scheme is to allow for a library of basic functions – *e.g.*, the hash function – to be either pre-installed, or loaded into the TPM at startup and cached for future use.

3. THE μ TPM ARCHITECTURE

We propose a new TPM architecture called μ TPM which builds upon the X μ P concept [4] and the BTE proposal [8] mentioned in the introduction. As motivated above, the firmware is stored outside the trust perimeter of the μ TPM and authenticated on a command granularity. We define a hardware root of trust that measures and reports the code running in the μ TPM environment and controls access to the internal resources (*e.g.*, NVM). Our solution can execute multiple processes – either TPM instances or even arbitrary general purpose code – in parallel.

3.1 Architecture overview

The μ TPM requires similar hardware components like a regular TPM (see Figure 1): a microcontroller (μ C) possibly supported by a cryptographic coprocessor, volatile memory (RAM) and multiple-time-programmable (MTP) non-volatile memory (NVM). However, the major difference is the lack of read-only memory (ROM). For current TPMs, the ROM contains the firmware that implements the fixed functionality defined by the TCG specifications².

We define a new trust anchor called Firmware Trust Engine (FTE). This component deals with the firmware exter-

²The TCG specifies an optional `TPM_FieldUpgrade` command to update the firmware. This implies that the firmware can be stored (partially) in reprogrammable NVM.

Figure 1: μ TPM architecture. The Firmware Trust Engine (FTE) controls the processes running on the microcontroller (μ C): it stores the firmware identifier $\mathcal{C}$ for each process (identified by a process identifier pID) in a Firmware Configuration Register (FCR), authenticates externalized code (using the key k_{Auth}) and controls access to the non-volatile memory. The unique Hardware Endorsement Key (HEK) is certified by the μ TPM manufacturer and signs the content of the FCR during remote attestation. The IO and XIO interface are used to transfer data and code respectively.

nalization and manages the multiple execution contexts. It plays the role of the TCG roots of trust inside the μ TPM, namely:

- Root of Trust for Storage (RTS) by providing shielded locations to store for instance key material (see Section 3.2);
- Root of Trust for Measurement (RTM) by measuring the identity of the externalized code when it is loaded in the μ TPM environment and recording this measurement in a Firmware Configuration Register (FCR) (see Section 3.3); and
- Root of Trust for Reporting (RTR) by signing the content of the FCR with its Hardware Endorsement Key (HEK) (see Section 3.4).

The communication interface of the μ TPM architecture is also different. A standard TPM has one I/O port to transfer data to the rest of the platform; *e.g.*, the Low Pin Count bus in case of the PC platform. The μ TPM has a second I/O port to manage the processes and load program code. We denote the port to transfer data with IO³ and the control port with XIO. In a real implementation both (logical) ports probably use the same physical bus.

Alternative implementation.

We envision the FTE to be implemented as dedicated hardware, but its functionality can also be realized with a basic operating system. In this case, ROM memory is needed to reliably store the operating system and (ideally) the microcontroller provides two protection rings, *i.e.*, user and supervisor mode. This approach is similar to JavaCard-based smartcards, which allow the deployment of multiple applets on a single card.

³Optionally this port supports locality.

Figure 2: Life cycle of a μ TPM process. After its creation, the process goes into a measuring state to calculate the firmware identity $\mathcal{C}$ and authenticate the individual firmware commands. Afterwards, the process only executes authenticated microcode. Depending on the code measurement scheme, the process can go back in measuring mode to authenticate additional commands; however this is optional. The selection and deselection commands allow over context switches.

3.2 Process management

The μ TPM architecture supports multiple execution contexts, which we call *processes*. A process typically corresponds with the implementation of one TPM, but as argued before, it could also be another – arbitrary – security critical task. Scheduling of the processes is done outside the μ TPM: the μ TPM Support Stack (μ TSS) has to indicate which process the μ TPM should run. Note though that TPM commands are atomic. To keep the scheduling simple, process switches are not allowed while the TPM is executing a command. This may block operation for a while if one process is executing a computationally heavy task, such as RSA key generation, or completely block the μ TPM if the code running in it is blocking (*e.g.*, infinite loop). However, those events are either rare or suggest that something has gone massively wrong already, while the ability to switch contexts at arbitrary times would lead to greatly increased complexity.

3.2.1 Process life cycle

The μ TPM processes can be managed with the following commands over the XIO interface:

- **CreateProcess(s)** creates a new execution context with a NVM size of s and returns a process identifier pID ⁴. If the desired amount of NVM cannot be allocated, an error code is returned, indicating the amount of memory the process can get. Note that a process, once created, is only destroyed by the corresponding μ TPM command, but survives platform reboots.
- **SelectProcess(pID)** switches context to the process pID . This especially implies that on the XIO interface, only microcode with the corresponding code identity are accepted (see Section 3.3).

⁴Alternatively, the command can have the pID as parameter. If the process number is no longer available, an error code is returned; the μ TSS can decide to delete the process in order to claim a desired pID .

Figure 3: Memory layout of a process. The volatile memory is divided into a dynamic part that is overwritten when the microcode of a different TPM command is loaded, and a static segment that is cleared when the process is deselected.

- **DeselectProcess()** temporarily stops the currently selected process. This involves clearing its volatile memory.
- **DeleteProcess()** permanently stops the currently selected process. This involves clearing its volatile and non-volatile memory and deleting the associated process descriptor.
- **ListProcesses()** outputs the list of all processes, *i.e.*, the process identifier, FCR and NVM size. This allows the user to identify unwanted μ TPM usage, or processes that use up too much memory.

The process life cycle is summarized in Figure 2. After the creation of a process, a measurement of the process' code base is made, and the code identity is stored in the FCR. Next, the process can be selected to actually run the previously measured code. The mechanisms to measure and execute code in a selected process will be described in Section 3.3.

Remark that that the execution of code is blocking; the process can only be stopped by a platform reset or with **DeleteProcess**, which deletes its volatile as well as persistent state.

3.2.2 Memory management

In order to minimize the hardware resources of the μ TPM, the size of the RAM and NVM memory should be kept small. Therefore, only the code and data of the running process are stored in the volatile RAM. We assume that the μ TPM has sufficient internal NVM and hence that each process has exclusive access to a dedicated part of this memory⁵. In Section 5 we alleviate this requirement by (partly) externalizing the NVM.

Non-volatile memory.

The FTE provides rudimentary *virtual memory management* for the NVM by mapping logical addresses to physical addresses. This ensures strict process isolation, as processes cannot access the physical NVM directly (see Figure 3). For each process, the FTE stores a *base address* b and the *size* s

⁵If desired, a mechanism to share NVM between different processes could be provided. However, this adds complexity to the FTE.

of its non-volatile memory. When a process wants to access the virtual NVM address l , the address gets mapped to the physical address $b+l$ on the condition that $l \leq s$. The NVM size and management allows an implementation to compromise on complexity and efficiency. A low-end μ TPM may for example restrict NVM usage to a constant size which is a power of two, which means that the address translation can be implemented efficiently without an adder circuit.

Volatile memory.

The volatile memory is directly accessible by the running process, without intervention of the FTE. Conceptually we divide the volatile RAM of the process into two *segments*. The *static* part contains code and data that is shared between different commands belonging to one particular process, *i.e.*, operational RAM (such as a loaded key from the original TPM) and shared library code. The *dynamic* part contains the current command to be executed, as well as all data only needed temporarily for this execution. The static part has to be loaded into RAM once the process is activated – *i.e.*, immediately after the process selection command. The dynamic part is loaded into RAM when a particular command is loaded with the **Execute** command, which will be described in Section 3.3.

3.2.3 Library support

To minimize the implementation size of each command, the μ TPM could include a library for standard cryptographic operations. In addition, it is possible to load specialized libraries into the static volatile memory (see Figure 3) that allow for optimizing the size of the individual code blocks. Those libraries are uploaded after a process is selected (with the **LoadLibrary** command, which will also be explained in Section 3.3), and stay in memory until the next context switch.

3.2.4 Context switch

During a context switch, the physical RAM memory is automatically cleared and overwritten. This guarantees that no secrets leak between processes. The shared library code and the dynamic memory segment are reloaded externally by the μ TSS. However, the process needs to temporarily store its volatile state to the non-volatile memory before it is suspended, and restore the state afterwards on resumption. This can be done in different ways.

- The memory management unit of the FTE implements *swapping*. This however adds extra complexity to the μ TPM and hence contradicts the principle of a minimal trusted computing base (TCB).
- Each process has an exit and entry routine, *i.e.*, code that is executed before the μ TPM switches to a new process, and code that is executed immediately after the process is selected. This code is registered with the FTE during the process creation. This concept is similar to JavaCard applets, which have a **select** and **deselect** method.
- The process itself is responsible for the storage of sensitive state information. The current TCG specification has commands to suspend and resume the TPM, normally used for power saving of the platform: the **TPM_SaveState** command stores the volatile state (*e.g.*,

content of PCR) in NVM and **TPM_Startup** restores this information. The μ TSS has to invoke the correct command before and after the selection of a process.

We opt for the latter approach as this requires minimal support by the μ TPM.

3.3 Code integrity measurement

The integrity of all external code executed on the μ TPM is measured by the Firmware Trust Engine. In this aspect, our architecture differs from X μ P proposal that only executes code authorized by the device manufacturer. We propose two schemes to measure the integrity of code.

3.3.1 Signature-based measurement

A first option is to sign the code of every TPM command individually. Consequently, the complete TPM firmware can be represented as a list of executable codes c_i , and associated digital signatures σ_i :

$$\{pk, \{\sigma_0, c_0\}, \{\sigma_1, c_1\}, \dots, \{\sigma_n, c_n\}\} \quad \text{with } \sigma_i = \text{sign}_{sk}(c_i)$$

We assume that the signature key sk is the same for all commands c_i . The hash of the corresponding public key is used as the code's identity:

$$\mathcal{C} = \mathcal{H}(pk)$$

It might occur that the main TPM functionality is produced by one entity (*e.g.*, a current TPM vendor), but extensions are added by a different entity (*e.g.*, government agency). In this situation the second entity has to sign the extensions as well as the original code with its own signature key.

Verification of digital signatures imposes a considerable computational and communication overhead. Therefore, the μ TPM verifies the signature on every TPM command once and calculate a MAC for subsequent verifications. The MAC key k is derived from the firmware identity and a device specific secret key k_{Auth} :

$$k = \mathcal{H}_{k_{Auth}}(\mathcal{C}) = \mathcal{H}_{k_{Auth}}(\mathcal{H}(pk))$$

This enables the μ TSS to transform the signed TPM commands into MACed executables:

$$\{\{\mu_0, c_0\}, \{\mu_1, c_1\}, \dots, \{\mu_n, c_n\}\} \quad \text{with } \mu_i = \mathcal{H}_k(c_i)$$

The μ TPM provides the following commands over the XIO port:

- **AssociateKey(pk)** loads the public key pk in volatile memory and stores $\mathcal{C} = \mathcal{H}(pk)$ in the Firmware Configuration Register. This command has to be invoked after the process creation.
- **Authenticate(σ, c)** loads the process code c in the dynamic RAM segment and verifies its signature σ . If the verification is successful, the μ TPM returns a MAC on the code c : $\mu = \mathcal{H}_k(c)$ with $k = \mathcal{H}_{k_{Auth}}(\mathcal{C})$.
- **Execute(μ, c)** loads the process code c in the dynamic memory segment, and verifies the MAC μ . If the verification is successful, the code c is executed.
- **LoadLibrary(μ, l)** loads the library code l in the dynamic RAM segment, and verifies the MAC μ . If the verification is successful, the library l is copied to the static memory segment of the process.

The first time that the `Authenticate` command is executed, a flag is set to prevent the invocation of `AssociateKey`. If the μ TSS wants to associate a different key with the process, the existing process must be stopped and a new one must be created, with `DeleteProcess` and `CreateProcess` respectively.

Using this approach, it is possible to add or replace commands without changing the firmware identity, provided that the code manufacturer signs the new commands (and the μ TSS authenticates them with `Authenticate`). This needs to be done carefully though, as mixing different versions of commands can lead to security problems. It is thus recommended to change the code identity (*i.e.*, different sk and pk) if major updates are performed, and replace the entire code base. This is equivalent with the creation of a new TPM and consequently existing keys have to be migrated with the migration or maintenance functionality⁶. It would be possible to build in some versioning support as well, but this appears to add unnecessary complexity to the system.

3.3.2 Binary measurement

The hash of the complete TPM firmware can also be used to identify the code:

$$\mathcal{C} = \mathcal{H}(c_0, c_1, \dots, c_n)$$

Note that the order in which the TPM commands are measured, determines the resulting hash. Alternatively, the measurement could be computed like the TCG PCR extension operation:

$$\mathcal{C} = \mathcal{H}(\mathcal{H}(\mathcal{H}(0, c_0), \dots), c_n)$$

During a measurement step, the μ TSS provides the μ TPM with the whole firmware image by sequentially loading the code fragments c_i . Each time a new command is loaded, the FTE updates the FCR and returns a MAC on the code. This time, the MAC key k is derived from the process identifier pID and the μ TPM secret key k_{Auth} :

$$\mathcal{H}_{k_{Auth}}(pID)$$

After this initialization, the μ TPM will only execute the MACed code.

The μ TPM provides the following commands over the `XIO` port:

- `Authenticate(c)` loads the process code c in RAM, updates the FCR and returns a MAC on the code c : $\mu = \mathcal{H}_k(c)$ with $k = \mathcal{H}_{k_{Auth}}(pID)$.
- `Execute( $\mu, c$ )` loads the process code c in the dynamic RAM segment, and verifies its MAC μ . If the verification is successful, the code c is executed.
- `LoadLibrary( $\mu, l$ )` loads the library code l in the dynamic memory segment, and verifies its MAC μ . If the verification is successful, the code l is copied to the static memory segment.

The first time that `LoadLibrary` or `Execute` is executed, a flag is set to disallow new invocations of `Authenticate`. If the μ TSS wants to authenticate additional code at a later stage,

⁶This complies with the TPM specification, which states that “When a field upgrade occurs, it is always sufficient to put the TPM into the same state as a successfully executed `TPM_RevokeTrust`.”

a new process must be created and the complete code base must be measured again.

If this method of authentication is chosen, users are free to select any subset of commands to be integrated into a process. Though it is not possible to add new commands while maintaining access to the process memory – otherwise, a TPM could be attested as a good code base, and later commands are added to read out critical keys.

3.4 Code integrity reporting

The FTE guarantees that a process only executes authenticated code and that processes run isolated from each other. Therefore, the secrets stored by a TPM (*e.g.*, SRK, monotonic counters, owner credential, etc.) stay confidential. The TPM process can generate its own Endorsement Key (EK), but a mechanism is desired to link this key to the μ TPM hardware and the firmware identity $\mathcal{C}$. Otherwise remote parties cannot distinguish between a hardware TPM and a software emulator [14].

For this reason, every μ TPM ships with a unique asymmetric key pair called Hardware Endorsement Key. During manufacturing, the μ TPM producer creates a HEK certificate that certifies that the μ TPM architecture is implemented correctly.

The FTE provides the following attestation routines:

- `FTE_FCRRead()` returns the firmware identity $\mathcal{C}$, which is stored in the Firmware Configuration Register.
- `FTE_Quote(b)` uses the HEK to create a signature on the blob b and the FCR content:

$$q = \text{sign}_{HEK}(b, \mathcal{C})$$

It is important to note that the attestation feature is not exposed externally over the `XIO` interface, but only internally to μ TPM processes.

With this functionality, a TPM process can register its own EK with a privacy CA by doing a quote operation on the public EK and a fresh nonce. The privacy CA should include the firmware identity $\mathcal{C}$ as an attribute in the endorsement credential. Once the TPM process has a valid endorsement certificate, the standard TCG protocols can be used to get Attestation Identity Keys.

3.5 μ TPM Software Support

The TCG Software Stack (TSS) has to be augmented to deal with the externalized firmware. It has to store the disembedded firmware and load the appropriate TPM microcode (over the `XIO` port) before passing the command parameters over the `IO` port. The integrity of this firmware is protected by the μ TPM’s authentication key k_{Auth} , however availability cannot be guaranteed. If the external firmware gets deleted (purposely or involuntary), the corresponding μ TPM process becomes unusable. However this is not a big concern as other denial of service attacks can be applied on current TPMs (*e.g.*, deleting part of the TSS or the TPM driver), and TPMs were never meant to withstand a denial of service attack.

Likewise, the CRTM (Core Root of Trust for Measurement) has to store the code of a small number of TPM commands used during the platform startup (*e.g.*, `TPM_Extend`). The CRTM might use another non-volatile storage medium (*e.g.*, BIOS Flash memory). If the authenticated firmware becomes inaccessible, the CRTM can choose to destroy the

process or re-authenticate the missing microcode; both prevent the chain of trust from being bypassed.

4. SECURITY COMPARISON

Our approach attempts to stick closely to the original TPM attack model. Nevertheless, the exposure of the internal TPM data does create small differences, which in some applications may require extra care to prevent an attack.

4.1 Denial of Service

In our system, the attacker can destroy the internal state of the TPM, and even the command set; while it cannot be replaced by something meaningful, such an attack may disable the TPM or force a reset into its native state. This is a stronger attack than possible on a normal TPM, where deletion of external data can only destroy key material, but no state information or functionality. We would argue though that in practice, there is little difference between an attack on a μ TPM and an attack on a classical one; if a classical TPM loses all key blobs, most crucial state information is lost as well, and an application using this TPM needs to find a way to recover without opening other avenues of attack.

The only setting where a difference may occur is in a virtualized environment; it must not be possible that one virtual environment can destroy state information of the virtual TPM of another one. This means that each virtual TPM must have a local copy of its relevant state information, and shared information must be stored by a trusted program (*e.g.*, the hypervisor), where it cannot be deleted by individual virtual machines (some mechanism like this would be required anyway to prevent the virtual machines from disrupting each other).

4.2 Access analysis

While our μ TPM implementation protects TPM data from being read by an attacker, the attacker may be able to see the encrypted internal state, or – for example, by analyzing the cache of the μ TPM or the untrusted storage – obtain information about the last commands submitted to the μ TPM. While this does not endanger security in most settings in which a TPM is used, this increased visibility should be taken into account, and critical applications may need to apply additional measures to hide the activity of the μ TPM, for example by adding fake encrypted state blocks and clearing all caches after usage.

4.3 Vendor backdoor

For some implementations of code integrity measurement, it is possible for the vendor to authorize TPM commands that are invisible to the user (*i.e.*, not supplied with the original μ TPM on delivery), but used later as a backdoor to violate TPM security. Thus, in security settings where the vendor is untrusted, the MAC based binary measurement should be used rather than the one based on vendor signatures, or the user should use their own verification key when the `AssociateKey` command is executed, and thus lock the original vendor out of the TPM.

5. MEMORY EXTERNALIZATION

To allow for several independent processes to run on a μ TPM, and to minimize the amount of memory needed inside the μ TPM, we require a mechanism that allows for ex-

ternalization of the NVM, and potentially even parts of the RAM.

There are two major issues with externalized memory. For one, the memory management may get more complex, especially if the memory is not sufficient for a single process. Secondly, externalized memory needs an additional protection against replay attacks, *i.e.*, an attacker presenting passing an old memory block as a new one.

While we design our solutions to allow for an arbitrary number of processes, realistically the number of processes using a single μ TPM is relatively low. The only use case we see for a large number of processes is on a server with many virtualized operating systems and a large number of applications; in this case, it is safe to assume that the manufacturer would choose a μ TPM with a large amount of internal memory and a fast communication bus. In other words, while we do not want to set an upper limit on the number of processes and their memory requirements, we are not trying to build an architecture that scales efficiently to hundreds of processes if the μ TPM has only 1 kbit of memory.

5.1 Memory protection

Whenever memory is externalized to the insecure storage, it needs to be encrypted and authenticated to protect its confidentiality and integrity. This can simply be done by using a symmetric cipher in an authenticated encryption mode such as EAX [1], GCM [10], and OCB [11]. To guarantee *freshness*, whenever externalized memory is updated, the μ TPM needs to make sure that all old versions of that memory become unavailable. This is done by either adding a counter value to the externalized memory, or changing the memory authentication key every time that the memory is updated. The former solution requires less storage in the TPM, as one master key can be used for all processes, and each memory block only needs a counter that can be much shorter than a cryptographic key.

Ideally, if the amount of virtual memory is limited, the list of counters fits into the μ TPM. If one wants to allow an arbitrary number of processes, though, the protected NVM may even be too small for all counter values. In this case, the counter values themselves need to be externalized to insecure storage, and the freshness of the counters protected by a master counter. Advanced tree-based solutions for the management of a large number of counters exist [13]; however, we assume that the number of processes is relatively low, and a simple list based approach suffices.

5.2 Memory management

In a simple setting, we can assume that each process has its own NVM and RAM, which fits completely into the μ TPM hardware; however, the size of the μ TPM's memory is insufficient to simultaneously hold the memory of all processes. Thus, swapping is only required when the process is switched. In this case, management of the external memory is relatively simple – as an answer to a process creation/switch command, the μ TPM returns the (encrypted and authenticated) content of the current process, and expects the content of the new one in return. This does not necessarily need to be implemented in the μ TPM hardware itself; the process can handle the retrieval and cleaning of the memory itself (*e.g.*, with the `TPM_SaveState` and `TPM_Startup` command).

If the NVM or RAM needed for the process exceeds the

available size of the hardware memory in the μ TPM, externalization gets more complicated. Ideally, one can design complete virtual memory management and simulate page faults as TPM responses to the μ TSS. We do, however, feel that this contradicts the principle of a minimized hardware unit. It is possible for the μ TPM to communicate to the μ TSS during command execution by using error codes; the μ TPM is a passive device, hence error codes are the only mechanism to signal the μ TSS about problems. This would allow us to implement a function that can swap in and out blocks of internal memory manually while commands are being processed. However, we do believe that this design is an abuse of error codes, and that there is little need to support single commands that exceed the capacity of the μ TPM.

5.3 Memory availability

It should be noted that externalization of the NVM allows for some new forms of denial of service attacks. While the externalized firmware can easily be replaced if it got lost, an attacker may be able to remove the externalized NVM for good, and the only recovery from such an attack is the reset of the μ TPM to its default setting. While this attack is not critical in most TPM usage scenarios – to execute it, the attacker already requires a level of control over the host platform that allows for many other ways of denial of service attacks – this is an attack that is not possible for a normal TPM, and some TPM based protocols may assume it impossible.

One solution to this issue is for the μ TPM to collaborate with secure storage, as specified by the TCG Storage Work Group [15]. In this case, the μ TPM only releases its internal memory once it receives an acknowledgement from its counterpart in the hard disk that the memory content has successfully been stored in a hard disk section that is unavailable for the operating system.

6. CONCLUSION

We have shown that many of the current issues with TPM implementations that stem from complexity and inflexibility can be overcome by redefining the trust boundaries. By putting the firmware outside of the secure hardware and securing it cryptographically, our architecture allows for simplified hardware, while gaining flexibility in the supported command set and even allowing multiple secure coprocessors to share the same hardware. Our architecture is largely compatible with the current specification, provided an additional software layer between the classical TSS and the μ TPM; thus allowing for the improved architecture without having to adapt the TCG specification.

7. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their valuable comments.

This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by the FWO project G.0300.07 (Security components for trusted computer systems), by the IBBT QoE project, and in part by the European Commission through the IST Programme under Contract IST-027635 OPEN_TC.

8. REFERENCES

- [1] M. Bellare, P. Rogaway, and D. Wagner. The EAX Mode of Operation. In B. K. Roy and W. Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 389–407. Springer-Verlag, 2004.
- [2] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association.
- [3] BSI. Federal Government's Comments on the TCG and NGSCB in the Field of Trusted Computing. http://www.bsi.bund.de/sichere_plattformen/trustcomp/stellung/StellungnahmeTCG1_2a_e.pdf.
- [4] B. Chevallier-Mames, D. Naccache, P. Paillier, and D. Pointcheval. How to Disembed a Program? In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 441–454. Springer-Verlag, 2004.
- [5] B. Chevallier-Mames, D. Naccache, P. Paillier, and D. Pointcheval. How to Disembed a Program? Cryptology ePrint Archive, Report 2004/138, 2004. <http://eprint.iacr.org/>.
- [6] V. Costan, L. F. G. Sarmenta, M. van Dijk, and S. Devadas. The Trusted Execution Module: Commodity General-Purpose Trusted Computing. In G. Grimaud and F.-X. Standaert, editors, *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*, volume 5189 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2008.
- [7] K. Dietrich. An Integrated Architecture for Trusted Computing for Java enabled Embedded Devices. In *2nd ACM workshop on Scalable Trusted Computing – STC '07*, pages 2–6, New York, NY, USA, 2007. ACM.
- [8] T. Eisenbarth, T. Güneysu, C. Paar, A.-R. Sadeghi, D. Schellekens, and M. Wolf. Reconfigurable Trusted Computing in Hardware. In *2nd ACM workshop on Scalable Trusted Computing – STC '07*, pages 15–20, New York, NY, USA, 2007. ACM.
- [9] J.-E. Ekberg and M. Kylänpää. Mobile Trusted Module (MTM) – an introduction, Nov. 2007. <http://research.nokia.com/files/NRCTR2007015.pdf>.
- [10] D. A. McGrew and J. Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In A. Canteaut and K. Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer-Verlag, 2004.
- [11] P. Rogaway, M. Bellare, and J. Black. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. *ACM Transactions on Information and System Security*, 6(3):365–403, 2003.

- [12] A.-R. Sadeghi, M. Selhorst, C. Stübke, C. Wachsmann, and M. Winandy. TCG inside?: A Note on TPM Specification Compliance. In *1st ACM workshop on Scalable Trusted Computing – STC '06*, pages 47–56, New York, NY, USA, 2006. ACM.
- [13] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *1st ACM workshop on Scalable Trusted Computing – STC '06*, pages 27–42, New York, NY, USA, 2006. ACM.
- [14] H. Stamer and M. Strasser. A Software-Based Trusted Platform Module Emulator. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Trusted Computing - Challenges and Applications, First International Conference on Trusted Computing and Trust in Information Technologies, Trust 2008, Villach, Austria, March 11-12, 2008, Proceedings*, volume 4968 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2008.
- [15] Trusted Computing Group. TCG Storage Architecture Core Specification. available at <https://www.trustedcomputinggroup.org/specs/Storage/>.