# Waltzing the Bear, or: A Trusted Virtual Security Module

Ronald Toegl, Florian Reimair, and Martin Pirker

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Inffeldgasse 16a, A–8010 Graz, Austria
{rtoegl, freimair, mpirker}@iaik.tugraz.at

**Abstract.** Cryptographic key material needs to be protected. Currently, this is achieved by either pure software based solutions or by more expensive dedicated hardware security modules. We present a practical architecture to project the security provided by the Trusted Platform Module and Intel Trusted eXecution Technology on a virtual security module.
Our approach uses commodity personal computer hardware to offer integrity protection and strong isolation to a security module which implements a compact security API that has been fully verified. Performance results suggest that our approach offers an attractive balance between speed, security and cost.

**Keywords**: Trusted Computing, Hardware Security Module, Key Store, API Verification

## 1 Introduction

In public key infrastructures the secure handling of *private* key material is of critical importance. Yet, in many cases, simple *software key stores* are used due to their flexibility and low costs. They achieve only a very basic security against simple attacks, especially as their runtime memory is exposed to attacks [18]. On the other end of the market spectrum, Hardware Security Modules (HSMs) [11,3] offer isolated and protected secure processing environments hardened against a wide range of software and hardware attacks. Naturally, these devices tend to be expensive.

The technological convergence of Trusted Computing platforms with HSMs, as noted by Smith [34], allows for improved security levels on commodity general-purpose systems and thus offers a compromise between the two contradictory goals of minimizing costs and maximizing security.

In this paper we present the design and implementation of a *Trusted Virtual Security Module* (TvSM) that offers a restricted set of security critical operations and manages cryptographic keys. A hardware-supported virtualization platform on commodity PC hardware with Intel Trusted eXecution Technology (TXT) [17] extensions offers protection against insecure applications running on the same system and further ensures the integrity of the module through the Trusted

Platform Module (TPM). Within the TvSM, a flexible key hierarchy allows to support different use cases. As even for commercial HSMs logical API flaws have opened the way for attacks [7], we perform a formal security verification of the module's API. Prototype implementation results suggest that the cryptographic performance is in-league with medium-sized HSMs and a significant speedup when compared to the TPM.

*Outline* The remainder of this paper is organized as follows. In Section 2 we summarize the state-of-the-art of available security modules and of Trusted Computing platforms. Section 3 presents our architecture of a virtual security module, a software security module integrated with a modern trusted virtualization platform. A formal description of the module's API is presented along with verification results. In Section 4 we describe our implementation of the architecture, provide performance results and review the security level attainable. The paper concludes in Section 5.

## 2 Background

### 2.1 Security Modules

Current mass-market implementations[1] of secure key stores are either pure software based solutions or dedicated hardware implementations. *Software security modules* are, by concept, rather simple software libraries.

A typical client application will access them via the industry standard PKCS #11 [29] interface. Software security modules perform cryptographic operations in software and store sensitive data such as private keys protected with a secret password. Some modules take care to zero sensitive data after usage, some may not. Due to their operation within platform system RAM a malware-prone platform such as the common industry standard PC platform cannot effectively prevent the access to private key material in a software security module [18].

*Hardware Security Modules* provide a protected environment for processing of critical data. This comprises areas for execution of security functions as well as storage and management for sensitive data such as cryptographic keys. Hardware security modules are separate computing entities, often shielded or encased to prevent or detect physical manipulation. Some HSMs require proof of physical presence at system startup, be it to press a button, or to provide some sort of authentication token (a smart card, a PIN-code, or a multi-part USB-token). Dedicated hardware defense mechanisms protect data stored inside the HSM and will destroy the key memory if physical manipulation is detected.

Anderson presents a concise survey of hardware security co-processors [3] with the market offering a wide variety of functions and performances. Well-documented commercial designs are those of the IBM 4758 [35,11] and IBM

[1] `http://www.fanaticmedia.com/infosecurity/archive/Feb10/Authentication%20Tokens%20story.htm`

4764 series [5]. A less commercial, rather open hardware and software design was proposed by Gutmann [18].

The performance achieved often determines the price of a module. A typical smart card based HSM can sign data in a few seconds, some high speed HSMs with cryptographic hardware acceleration reach more than 10 000 signature creations per second.

A client application can access the module's services through PKCS #11 or custom Application Programming Interfaces (API). The design of such a security API is difficult and attacks have been found that allow the extraction of secret key material not only in theory, but also in practice for commercial HSMs [7].

### 2.2 Trusted Platforms

Trusted Computing as it is available today is based on specifications of the *Trusted Computing Group* (TCG). The core hardware component is the *Trusted Platform Module* (TPM) [37]. Similar in performance and implementation to a smart card, the TPM features tamper-resilient cryptographic primitives, such as an RSA engine. In addition, it is physically bound to its host device. The TPM can *bind* data to the platform by encrypting it with a *non-migratable* key, which never leaves the TPM's protection. The TPM also helps to guarantee the integrity of measurements of software components by offering a set of *Platform Configuration Registers* (PCRs). PCRs exactly document the software executed in the transitive trust model, where each software component is responsible to measure the following component before invoking it. A trustworthy *root of trust for measurement*, either a component of the *static* BIOS software or the result of a *dynamically* invoked special CPU instruction, serves as initial starting point for this process.

These mechanisms allow projecting the trust from a dedicated, trustworthy hardware security module onto the operating system and services of a general purpose platform. An early example is given in the Dyad System [38]. After the introduction of the TPM, the Enforcer platform [22] showed integrity protection mechanisms that can be applied as basis for possible security applications such as the Bear software security processor [21]. Similarly, IBM's Integrity Measurement Architecture [30] integrates PCR measurements of file accesses in a Linux. These first generation of trusted platforms offer hardware-guaranteed integrity measurement, but only operating system-level process separation and memory protection.

*System Virtualization* is a stronger way of dividing the resources of a computer into multiple execution environments. Commonly, virtualization is controlled by a singleton *hypervisor*, a superior control entity which directly runs on the hardware and manages it exclusively. It enables the creation, execution and hibernation of isolated *partitions*, each hosting a guest operating system and the virtual applications building on it. Among systems [27,15] that use system virtualization for separation and isolation, Nizza [33] extracts security critical code out of legacy applications, transferring them into a separate partition. A combination of virtualization and the static root of trust mechanisms

of the TPM leads to second generation trusted platforms such as [12,26,9,31]. A general drawback is the influence of firmware code on the static chain-of-trust. Individual hardware's BIOS is part of the state and this fact complicates update procedures and precludes the a priori definition of trusted configurations.

Such platforms have nonetheless been used to provide cryptographic key storage and protection. Targeted on applications like web-browsing and SSH authentication, Kwan and Durfee introduce Vault [20] to handle sensitive information via a GUI in a separate virtual machine. Likewise, TruWallet [14] acts as secure TLS proxy that securely stores and handles user credentials in an integrity-protected application in a TPM-protected compartment. Virtual TPMs [6] provide TPM functionality to several partitions by multiple isolated software simulations of TPMs.

Modern PC hardware from AMD [2] and Intel [17] provides, in addition to the TPM, the option of a *dynamic* switch to a known-secure system state and strong, chip-set guaranteed separation of memory for different partitions. Recently, progress has been made towards a third generation of trusted platforms, where a dynamic root of trust for measurement can be invoked at any time, making the chain-of-trust independent of legacy software components such as the BIOS and thus deterministic. BIND [32] uses AMD's Secure Virtual Machine (SVM) [2] protection features to collect fine grained measurements on both input and the code modules that operate on it so that the computation results can be attested to. Flicker [24] isolates sensitive code by halting the main OS, switching into AMD SVM, and executing short-lived pieces of application logic (PALs). PALs may use the TPM to document their execution and handle results. As a trusted hypervisor, TrustVisor [23] is initiated via the DRTM process, assumes full control and allows to manage, run and attest multiple PALs in its protection mode, without the repeating switch costs incurred by the Flicker approach.

## 3 A (Trusted Virtual) Security Module

Our architecture can be characterized by the integration of two components: A robust base software platform build on Trusted Computing enhanced commodity hardware, and a software security module which implements a compact security API. Together, they are designed to offer a flexible service that still exposes only a minimal attack surface.

### 3.1 Trusted Virtualization Platform

Our acTvSM platform[2] is a third generation Trusted Computing architecture, specifically designed to host software security modules. It enforces integrity guarantees to itself as a software platform and the applications and services it hosts while applying strong hardware-based memory separation of partitions. The platform takes advantage of the Linux Kernel-based Virtual Machine (KVM)

[2] Available for download at `http://trustedjava.sf.net`

hypervisor, which is operated in a non-trivial configuration, which we call *Base System*. In previous work we have presented the secure boot mechanism, the file system layout [36] and application management and update [16] processes. We now show the integration with the security module.

The *secure boot* process using Intel's TXT ensures execution of a known-good hypervisor configuration. The `GETSEC[SENTER]` CPU instruction provides a well-defined, trusted system state. During boot a *chain-of-trust* is established by measuring software components into the TPM's PCRs. A *measurement* $m$ in this context is a 160 bit SHA-1 hash which is stored in PCR $i$ using the one-way *extend* operation: $extend_i(m) = \text{PCR}_i^{t+1} = \text{SHA-1}(\text{PCR}_i^t \,||m)$. For a PCRs initial power-on state ($t = 0$) we write $initial_i$ and for several $m_j, j = 1..n$ we write $extend_i\{m_j, .., m_n\}$. Note that this operation is the only way to alter the content of a PCR and depends on the previous value. To achieve a certain value in a PCR the same measurements have to be extended in the same order.

The TPM also serves as access-controlled, autonomous storage for two policies which are enforced. First, a *Launch Control Policy* (LCP), which is evaluated by an *Authenticated Code Module* (ACM) provided by Intel, defines which secure boot loader is allowed to run as a so called *Measured Launch Environment* (MLE) [19]. Second, a *Verified Launch Policy* (VLP), which is evaluated by the secure boot loader, defines which subsequent Kernel, `initrd` and Base System are allowed to be loaded and executed. Furthermore, mechanisms in the Intel TXT chip set offer isolated partitions of main memory and restrict *Direct Memory Accesses* (DMA) I/O to defined areas.

Based on these PCR measurements access to data can be restricted to a known-good platform state. This can be achieved using the TPM's ability to *seal* data. Sealing and unsealing means to encrypt – respectively decrypt – data inside the TPM with a key which is only available inside the TPM and where usage of the key is restricted to a defined PCR configuration. Under the assumption that the TPM is a tamper-resistant device this is a robust way to guarantee that sealed data can only be accessed under a predefined platform state. An important novel aspect of our architecture is that the measurements are predictable so the PCR values can be calculated *a priori* and data can be sealed to a *future trusted states* after a planned configuration update.

One of the platform's peculiarities is a structured set of file systems. For instance, measurements of the base system are taken over its whole file system. To achieve the same measurement on every boot we use a read-only file system. Comparable to a Linux Live-CD, an in-memory file system is merged to form the runtime file system. Services and applications are stored on encrypted logical volumes. Images can be read-only (therefore deterministically measurable) or mutable.

Beginning from power on, the system performs a conventional boot and then calls `GETSEC[SENTER]`. After returning from that call, the system continues to transit between system states $S_{boot}$, $S_{update}$ and $S_{application}$. The state $S_{boot}$ is reached where ACM and MLE have been measured and the secure boot loader has already launched the kernel and the code contained in the initial ram-disk

is in charge. In $S_{update}$, the platform has measured the full code base of the base system and accepts maintenance operations[3] by the platform administrator, respectively TPM owner. $S_{application}$ differs from $S_{update}$ by an arbitrary token that indicates that changes to the persistent state are prohibited and applications can now be offered. Finally, the chain-of-trust includes the security module service by a measurement of the module's software image before it is mounted. More formally,

$$\begin{aligned} S_{boot} &:= \{initial_{14}, initial_{15}, \\ &\quad extend_{18}\{\texttt{secure boot loader}, \texttt{linux kernel}\}, \\ &\quad extend_{19}\{\texttt{initrd}\}\} \\ S_{update} &:= \{S_{boot}, extend_{14}\{\texttt{base system}\}\} \\ S_{application} &:= \{S_{update}, extend_{15}\{\textit{token}\}\} \\ S_{service} &:= \{S_{application}, extend_{13}\{\texttt{security module image}\}\}. \end{aligned}$$

This precise definition of the expected and trustworthy system state represents a integrity-guaranteed boot of the platform up to the security module service. With it the TPM's sealing mechanism is used to restrict access to the file system access keys for the `security module image` to $S_{service}$ only. Complementarily, the hardware assisted memory isolation of the chip-set provides runtime isolation. Note that TXT also enforces DMA restrictions and memory zeroization upon boot cycles. Thus the virtualization platform ensures the protected operation of the virtual security module. The virtual security module experiences no extra performance penalty when compared to other general purpose applications in their respective partitions.

### 3.2 Virtual Security Module

We now describe the design of a cryptographic software module, which we outline in Figure 1.

Obviously, our TvSM features many of the typical functionalities found in other cryptographic modules. Connections are managed in sessions and authentication protocols ensure that only authorized roles can perform critical operations. This especially applies on initialization, backup and handling of key material. RSA keys can be used to perform cryptographic signatures within the module, but only if their creation-time defined restrictions support these operation. Finally, the module supports the creation of key hierarchies which are rooted in a single so-called master key. Other keys, or a hierarchy of storage keys, can be wrapped with it. This architecture allows for external storage, back-up of keys and scalability.

In addition to this generic design, we provide two enhancements to ensure that the presented Trusted virtual Security Module does warrant trust in it.

[3] In [16] we describe an update process that does not transit untrusted states.

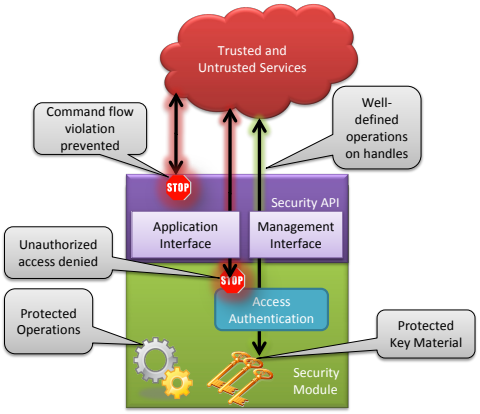


**Fig. 1.** The TvSM protects keys through a security API.

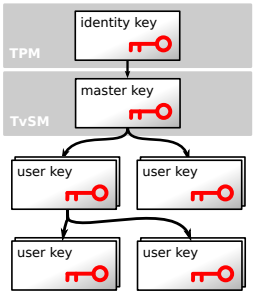


**Fig. 2.** The key hierarchy.

First, we apply recent advances in *security API* analysis already in the creation of the module's security API. To this end, we propose a set of operations that is specifically tailored to fit exactly the use case of performing cryptographic signatures and to protect the private key material needed for this task. Assuming that the design is correct, the module's behavior should protect the key material at all times and it is the security API's task to ensure that private keys remain under control of their authorized owner.

Second, we integrate the TvSM with the virtualization platform and build upon functionality that is protected by an actual hardware security module, namely the TPM. For instance, the module takes advantage of the hardware cryptographic random number generator of the TPM as seed in key generation.

We also root the key hierarchy of each instance of the TvSM in a unique TPM-key, which we refer to as identity key from now on. The overall key hierarchy structure is illustrated in Figure 2. We use the identity key to protect the master key, as well as for TvSM identification. Being non-migratable, the identity key identifies the TvSM uniquely. During setup we associate the TvSM instance to the identity key. The same key also protects the module's master key by binding it to the hardware platform. We choose the binding mechanism, as it does not enforce a trusted state for the key, as this would preclude updates of any part of the chain of trust after the module is deployed. Instead, the relation to a trusted state is achieved by storing the key blob of the identity key in an encrypted file system that is sealed to $S_{service}$ but can be re-sealed to future trusted states.

Previous work by Berger et al. [6] on the virtualization of the TPM recommended a similar coupling of hardware-to-virtual module key hierarchies. The Integration with our third-generation virtualization platform now allows us to leverage TPM sealing to this end, even in the face of updates of the chain-of-trust.

### 3.3 API Design

We decided against implementing a full PKCS#11 [29] interface. This would have increased implementation complexity, and would have precluded a concise, full-scope security analysis on the level of detail we desire.

Instead, we decided to create a new design that supports a single use case: the management and usage of asymmetric keys for cryptographic signatures. This focus allows us to reduce the attack surface and to design the actually needed functionality with full formal rigor. A number of contributions influenced the design. For instance, we avoid arithmetic operations like `XOR` on API data structures [40], clearly group related functionalities, use static types for keys according to their use [37,13], use the verified authentication protocol SKAP [8], store exported keys and their attributes together [7], and offer an easy to program, typed, and object-oriented interface definition.

We use the the following syntax to give an abstract description of the core operations. We denote as $k$ a cryptographic key pair where $k^{priv}$, $k^{pub}$ are the private resp. public asymmetric keys. Asymmetric encryption of $m$ under key pair $(k^{pub}, k^{priv})$ is $enc(m, k^{pub})$; decryption is $dec(enc(m, k^{pub}), k^{priv}) = m$. RSA signatures are written $sig(m, k^{priv})$. A handle (i.e. the assignment of a nonce to identify an object behind the module's security boundary) is the binding of nonce $n_1$ to key $k_1$: $\mathrm{h}(n_1, k_1)$. Keyed integrity is denoted by $\mathrm{HMAC}_k(m)$. Attribute $a$ is element in attribute set $\mathbf{A}$: $\mathbf{A} \supseteq a$.

We describe the signature of API functions from the external Dolev-Yao [10] view, and augment this with parameter conditions checked internally and elements freshly created, assuming successful authorization.

$$\texttt{function} : input \xrightarrow[internal\ state\ change]{conditions\ checked} output$$

Cryptographic keys are typed with attributes from the set $\{\mathrm{MST}, \mathrm{MIG}, \mathrm{SIG}, \mathrm{STO}, \mathrm{EXT}\}$ and size and algorithm information for the respective key type descriptors master, migratable, signature, storage and exportable. Note that only storage keys may wrap other keys, and only signature keys may sign messages. Attributes cannot be changed at any time after creation.

Overall, the API consistis of two interface, `MaintenanceSession` and `OperatingSession`, with ten methods each and 29 helper classes. The `MaintenanceSession` is concerned with initialization of a fresh module with a user-specific key hierarchy beneath the unique Identity Key pair $k_{ID}$. $k_{ID}$ is created when installing the module's software image and non-migratably bound to the TPM. A Client in the `MaintenanceSession` uses $k_{ID}^{pub}$ in the SKAP protocol to authenticate the session. Within the session, a singleton master key $k_M$ can be freshly created calling method (1). If a master key is created with attribute MIG set and if the necessary authorization is provided, it may be migrated (2) to another identity key $k_{ID2}$ by using the TPM binding mechanism. Inversely, it can be imported (3) to allow scalability by having several parallel instances of the TvSM with the same key hierarchy. This scheme can also serve to establish

a back-up process. $k_M^{pub}$ is world readable (4) and a TvSM instance can also be instructed to delete (5) the hierarchy root key.

$$\texttt{createMasterKey} : \mathbf{A} \xrightarrow[new\ k_M^{pub},k_M^{priv},k_M.\mathbf{A}=\mathbf{A}]{\mathbf{A}\supseteq\{\text{MST,STO}\}} \text{h}(n_M, k_M) \quad (1)$$

$$\texttt{migrateMasterKey} : k_{ID2}^{pub} \xrightarrow{k_m.\mathbf{A}\supseteq\{\text{MIG,MST}\}} \texttt{TPM\_BOUND\_DATA}_{k_{ID2}^{pub}}(k_M, \mathbf{A}) \quad (2)$$

$$\texttt{importMasterKey} : \texttt{TPM\_BOUND\_DATA}_{k_{ID2}^{pub}}(k_M, \mathbf{A}) \xrightarrow{\mathbf{A}\supseteq\{\text{MIG,MST}\}} \text{h}(n_M, k_M) \quad (3)$$

$$\texttt{getMasterKey} : \rightarrow k_M^{pub} \quad (4) \qquad \texttt{destroyMasterKey} : \rightarrow \perp \quad (5)$$

**Fig. 3.** Abstract TvSM Maintenance API.

Once a master key is established, it can be used to authenticate an `OperatingSession`. Here, the master key is accessible by its handle (13). The key hierarchy is built by creating (7) a fresh key pair under an existing storage key $k_1$ or $k_M$. The return value of this creation process is a data structure that holds the private key and the policy assigned to it. Confidentiality is protected by encryption with parent key $k_1$:

$$\begin{aligned}\text{KBLOB}(k_2, k_1) = \{&k_2^{pub}, enc(k_2^{priv}, k_1^{pub}), enc(k_S, k_1^{pub}), k_2.\mathbf{A},\\ &\text{HMAC}_{k_S}(k_2^{pub}||k_2^{priv}, k_S||k_2.\mathbf{A}||..)\}\end{aligned} \quad (6)$$

The TvSM is actually oblivious to the number of keys it can protect by leaving the actual storage location of key blobs at the discretion of its clients. Thus keys need to loaded (i.e. unwrapped) into a key slot before they can be used. Keys can then be loaded (8) only, if the parent key has been previously loaded for decryption. Slots may also be freed (14). Thus a hierachy of keys is created. Application keys can be migrated (9) to other storage keys, if the attribute MIG and a migration authorization secret was set at creation. Migration essentially means the re-wrapping of a key previously wrapped under $k_1$ to the new parent $k_2$. A different mechanisms is the import (10) of externally created keys. This allows the TvSM to hold legacy keys and enforce that they are only used if the correct credentials, i.e. authentication passwords are provided. Since the private part of the key has been created externally, this has to be noted in the attribute set. Only for this type of key, an export of the private key is allowed (11). Finally, SIG keys can perform RSA signatures (12).

### 3.4 API Behavior Verification

The correct handling of key material through a security API is a complex challenge and several theoretical attacks have been identified in the past. Even for commercial HSMs it has been shown that key material can be extracted [7] through the software interfaces concerning the creation, import and export of

$$\texttt{createKey} : \mathrm{h}(n_1, k_1), \mathbf{A} \xrightarrow[new\ k_2^{pub}, k_2^{priv}, k_2.\mathbf{A}=\mathbf{A}]{k_1.\mathbf{A}\supseteq \mathrm{STO} \oplus k_1.\mathbf{A}\supseteq \mathrm{SIG}} \mathrm{KBLOB}(k_2, k_1) \tag{7}$$

$$\texttt{loadKey} : \mathrm{KBLOB}(k_2, k_1), \mathrm{h}(n_1, k_1) \xrightarrow[new\ n_2]{k_1.\mathbf{A}\supseteq \mathrm{STO}\ \&\ \mathrm{HMACvalid}} \mathrm{h}(n_2, k_2) \tag{8}$$

$$\texttt{migrateKey} : \mathrm{h}(n_1, k_1), \mathrm{h}(n_2, k_2) \xrightarrow{k_1.\mathbf{A}\supseteq \mathrm{MIG}\ \& k_2.\mathbf{A}\supseteq \mathrm{STO}} \mathrm{KBLOB}(k_2, k_1) \tag{9}$$

$$\texttt{importKey} : k_2, \mathrm{h}(n_1, k_1), \mathbf{A} \xrightarrow[new\ n_2]{k_1.\mathbf{A}\supseteq \mathrm{STO}\ \& k_2.\mathbf{A}=\{\mathrm{SIG,EXT}\}} \mathrm{KBLOB}(k_2, k_1) \tag{10}$$

$$\texttt{exportKey} : \mathrm{h}(n, k) \xrightarrow{k.\mathbf{A}\supseteq \mathrm{EXT}\ k} k^{pub}, k^{priv} \tag{11}$$

$$\texttt{sign} : m, \mathrm{h}(n, k) \xrightarrow{k.\mathbf{A}\supseteq \mathrm{SIG}} sig(m, k^{priv}) \tag{12}$$

$$\texttt{getMasterKeyHandle} :\rightarrow \mathrm{h}(n_M, k_M) \tag{13}$$

$$\texttt{unloadKey} : \mathrm{h}(n, k) \rightarrow \perp \tag{14}$$

**Fig. 4.** Abstract TvSM Operating API.

cryptographic keys. Important issues are the incompleteness of security policies and implementations lacking checks and verifications of all pre-conditions and policies at all times.

We have verified the key security policies of the TvSM against a formal model of the API presented above. We created a machine-readable model based on the API analysis method presented by Froeschle and Steel [13] and refined for the results of [7].

The behavior of the API is modelled in the AVISPA intermediate format (IF), a language originally designed as machine-generated input format to the different back-ends of the AVISPA[4] protocol analysis push-button tool kit. Of the different AVISPA back-ends, we use SATMC [4] as back-end for our experiments. SATMC decides the security of a model against specifications by converting it through multi-set rewriting, linearization and various logical optimizations into a finite satisfiability problem which is then fed into a SAT solver. In our case, the MINISAT solver is used for this task. As with other model checkers, SATMC will either report a concrete attack trace or conclude that no attack could be found. SATMC is used because it can consider different system states in its analysis, a feature not available in most other protocol-oriented verification tools.

We consider the selection of key policies as the most important part of the design of an API, because most practical attacks have been found in this context. We therefore chose the following security rules for the key policies, which supplement the abstract API operations given in figures 3 and 4.

1. No MST key is EXT
2. STO keys are never EXT
3. A key must not be a SIG and STO key at the same time

From these statements we derive a truth table of valid key policies, which are the only ones allowed within the TvSM. This table is then used to refine the key

[4] `http://www.avispa-project.org/`

generation functions of the API definition to only create valid policies for the model. For each other operation in the API, the appropriate pre-conditions are set.

As with any other model checker tool, the such created model[5] is verified against a set of specifications. In the case of a security API, the specification should be security properties regarding the handling of private keys. In a formalization, these security properties are the goals of an attack on the API and describe what knowledge an Dolev-Yao intruder will attempt to gain. We assume that the intruder starts with his public/private key pair and can easily attain a handle to the Master-key through the API. Our security goal is then that the intruder must not learn any further private keys that were created under the protection of the security module. For SATMC this can be expressed by declaring new private keys as `secret` and defining that the attacker must not learn them.

We will now illustrate the verification undertaken with an example result from our experiments. Assume that due to some oversight in implementing the security policy of the module, the export key function (11) would not check that the key policy of the key it is applied to has the EXT flag set. The SATMC tool automatically detects this case and provides an easy-to-read trace containing the following API calls so that we can observe how an attack could be performed. First of all, a master key must be present; it is created (1) and a handle to it requested (13). In the next step, some other key $k$ with its security policy configuration is created (7) under $k_M$. Note that SATMC chooses one of several possible configurations; for this example it only matters that $k$ is not exportable, i.e. EXT set to `false`. Next in the trace, this key is loaded (8). Finally, the compromised operation of exporting a key (11) can be called on $k$. This should never have occurred, as the key is not EXT. This last step breaks the security goal, as the intruder learns the private key.

When insert the missing check of the EXT attribute in the such corrected model, no attack are be found. As the model checker has tested all possible command flows under the restrictions of its algorithm and the model, this *verifies* that the API is correct and secure. We conclude that our approach can be used to analyze the key policy of the TvSM and that it also allows to verify that the *right checks* are made *in the right places* of the code. These results can therefore be directly incorporated in the API specification and the TvSM implementation.

## 4 Implementation

### 4.1 Software Design

The general architecture of our implementation [28] is illustrated in Figure 5. The TvSM server is a boot-able binary image which includes the core executable, libraries, a Java Virtual Machine (JVM) and a Linux host operating system. The

[5] `http://www.iaik.tugraz.at/content/about_iaik/people/toegl_ronald/TvSMVerification.zip`

image can be loaded into an isolated compartment. Compartments are managed by a virtualization platform featuring attestation, secure boot, resource isolation, off-line image encryption and TPM forwarding. It is based on Debian Linux with KVM and several specialized Trusted Computing libraries and tools described in [36]. The DRTM late launch is performed with Intel's `tboot` loader. The API of our TvSM is accessible via the network interface using the Java remote method invocation protocol.

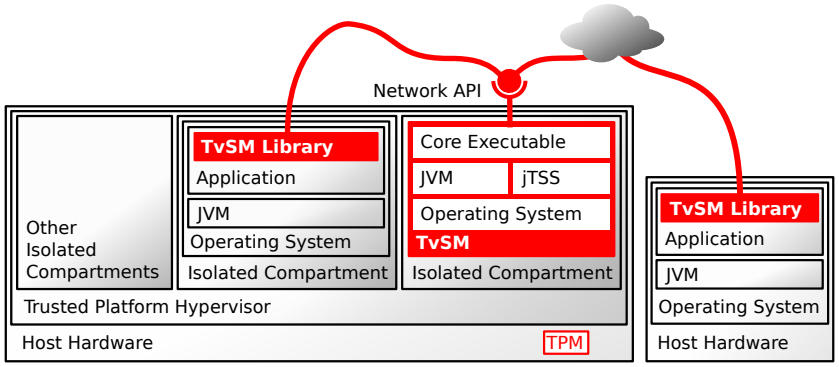


**Fig. 5.** Outline of the TvSM implementation.

An arbitrary client application can use the API through the TvSM client library which hides the communication overhead. A client may run on the same platform but confined in a different compartment or on a remote host platform.

### 4.2 Performance and Results

The TvSM server as well as the client application were operated on a Intel VPro platform with a Intel Core 2 Duo P9500 CPU running at 2.54 GHz per core running a Linux kernel 2.6.35-22 in 64 bit mode and Java JRE 1.6.0.22. Cryptographic operations are performed by the Crypto Provider of the SIC Crypto Toolkit [1].

The results for the TvSM are listed in Table 1 and measured from a Java environment, thus include all software initialization and communication overheads, with means calculated over 100 repetitions. Note that the same code base has been used for integration in the commercial XiTrust Business Server solution[6] that provides advanced signature services in the sense of EU Directive 1999/93/EC for electronic processes in large enterprises. The session opening in the SKAP protocol takes about 750$[ms]$, including a random number generation procedure for session secret creation. Creating a 1024-bit RSA key pair takes a mean time of about 60$[ms]$. It takes about 15$[ms]$ to load a key into the TvSM.

[6] `http://www.xitrust.com/`

**Table 1.** TvSM implementation performance.

| Operation | Duration [ms] | | |
|---|---|---|---|
| | min | max | mean |
| session init | 708,0 | 814,0 | 766,0 |
| create key | 21,0 | 378,0 | 62,0 |
| load key | 12,0 | 21,0 | 13,0 |
| sign | 6,0 | 9,0 | 7,0 |
| close session | 1,0 | 11,0 | 2,0 |

**Table 2.** TPM performance on RSA 1024 bit keys.

| TPM | Duration [ms] key gen. avg. | key gen. max | sig creation |
|---|---|---|---|
| Atmel v1.2.13.9 | 2756 | 6700 | 145 |
| Broadcom v1.2.6.77 | 1698 | 3538 | 300 |
| Infineon v1.2.3.16 | 3342 | 7773 | 200 |
| Intel v1.2.5.2 | 4564 | 15058 | 158 |
| STM v1.2.8.16 | 2228 | 8308 | 288 |

With a time of 7[$ms$] per operation, the TvSM is capable of creating about 150 signatures per second using a 1024-bit RSA key pair. Session shutdown is done in about 2[$ms$].

We now compare the performance results of our implementation to several available TPMs. Again we include all overhead introduced from a Java environment using the jTSS library for TPM access. A typical TPM chip is capable of creating about one key in three seconds and calculating about four to five signatures per second. However, the hardware random number generators limit the number of fresh keys that can be created as entropy needs to be gathered, literally, over time. Creating a 1024-bit key can take up to 15[$s$]. The mean durations for this test series are calculated over 15 repetitions.

In single threaded operation, our TvSM prototype implementation is capable of creating continuously about 17 1024-bit RSA keys per second and calculating about 150 signatures per second; in the signature creation task it equals about 30 times the throughput of a TPM, while in the key creation task this equals an speedup of 50 on average and up to 700 in extremis. A higher speedup can be expected with parallel sessions on multiple CPU cores.

### 4.3 Security Discussion

Attackers will attempt either to modify system states and their measurements, extract cryptographic materials, manipulate code execution, or attempt controlling the base platform or the applications executing on top of it.

Some selected TPM implementations have been based on resilient SmartCard architectures and certified according to Common Criteria EAL 4+ and provide a robust security against all but the most sophisticated hardware attacks. Yet, our platform does not store all critical data in the TPM at all times. Instead, we leverage TPM and TXT to provide a certain level of *modification resistance*, required by many applications and also of the FIPS 140 Security Level 2 standard [25], for the software executed. However, we are restricted to the security level that can be achieved by these chip-set features. Commodity devices can only be assumed to protect against very simple hardware attacks [17]. Hardware security therefore depends on physical access control and diligent operational procedures in data-centers and other deployment scenarios. We need to cau-

tion that not much effort, i.e. less than €100, is needed to break the security guarantees of TXT if an attacker has physical access [39].

However, the TPM protects the sensitive cryptographic data (keys) that the platform uses to guarantee the integrity of itself and of the TvSM application. The platform fully utilizes the hardware based PCR-mechanisms that protect measurements in the chain-of-trust of the components booted. Thus, a malicious boot with a following attempt to start a virtual application will fail if any part in the chain of trust was modified. Therefore, our platform can ensure that a trusted system state is reached after boot. Once booted, the TvSM API will be isolated via TXT and only provide the API specified. The formal analysis guarantees that private key material cannot be exposed using the instructions of the API offered. This will ensure a trustworthy behavior of the overall TvSM, assuming that there is no exploitable implementation bug and no physical attack.

## 5 Conclusions

We present a practical approach to project the security provided by the Trusted Platform Module through hardware virtualization and isolation on a virtual security module. Our TvSM offers a restricted set of security critical operations, e.g. cryptographic key management and signatures. It provides for high operational flexibility and fast cryptographic operations without extra hardware. The correctness of a formal model of the security API has been verified.

This Trusted virtual Security Module offers improved software attack resilience when compared to software key stores and better performance when compared to the TPM. Thus, while our approach does not achieve the same high level security as dedicated, tamper-resilient hardware modules, it offers an attractive cost-security-performance balance.

**Acknowledgments.** The authors thank the anonymous reviewers for their comments and Graham Steel for helpful remarks on programming SATMC. This work was supported by the Österreichische Forschungsförderungsgesellschaft (FFG) FIT-IT project acTvSM, no. 820848 and by the EC, through projects FP7-ICT-SEPIA no. 257433 and FP7-SEC-SECRICOM, no. 218123.

## References

1. CryptoProvider of SIC Crypto Toolkit. `http://jce.iaik.tugraz.at/sic/Products/Core-Crypto-Toolkits/JCA-JCE` (23 February 2011)
2. Advanced Micro Devices: AMD64 Virtualization: Secure Virtual Machine Architecture Reference Manual (May 2005)
3. Anderson, R., Bond, M., Clulow, J., Skorobogatov, S.: Cryptographic processors-a survey. Proceedings of the IEEE DOI - 10.1109/JPROC.2005.862423 94(2), 357–369 (2006)
4. Armando, A., Compagna, L.: SAT-based model-checking for security protocols analysis. Int. J. Inf. Secur. 7(1), 3–32 (2008)

5. Arnold, T.W., Doorn, L.P.V.: The IBM PCIXCC: a new cryptographic coprocessor for the IBM eServer. IBM J. Res. Dev. 48(3-4), 475–487 (2004)
6. Berger, S., Cáceres, R., Goldman, K.A., Perez, R., Sailer, R., van Doorn, L.: vTPM: virtualizing the trusted platform module. In: USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium. pp. 305–320 (2006)
7. Bortolozzo, M., Centenaro, M., Focardi, R., Steel, G.: Attacking and fixing PKCS#11 security tokens. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 260–269. ACM, Chicago, Illinois, USA (2010)
8. Chen, L., Ryan, M.D.: Attack, solution and verification for shared authorisation data in TCG TPM. In: Proceedings of Sixth International Workshop on Formal Aspects in Security and Trust (FAST'09) (2010)
9. Coker, G., Guttman, J., Loscocco, P., Sheehy, J., Sniffen, B.: Attestation: Evidence and trust. Information and Communications Security pp. 1–18 (2008)
10. Dolev, D., Yao, A.C.: On the security of public key protocols. In: Information Theory, IEEE Transactions on. Stanford University, Stanford, CA, USA (1981)
11. Dyer, J., Lindemann, M., Perez, R., Sailer, R., van Doorn, L., Smith, S.: Building the IBM 4758 secure coprocessor. Computer 34(10), 57–66 (2001)
12. EMSCB Project Consortium: The European Multilaterally Secure Computing Base (EMSCB) project (2004), http://www.emscb.org/
13. Fröschle, S., Steel, G.: Analysing PKCS#11 key management APIs with unbounded fresh data. In: Foundations and Applications of Security Analysis (2009)
14. Gajek, S., Löhr, H., Sadeghi, A.R., Winandy, M.: TruWallet: trustworthy and migratable wallet-based web authentication. In: Proceedings of the 2009 ACM workshop on Scalable trusted computing. pp. 19–28. STC '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1655108.1655112
15. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. In: Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003). pp. 193–206. ACM New York, NY, USA (October 2003)
16. Gissing, M., Toegl, R., Pirker, M.: Secure and trust computing, data management, and applications (2011), http://dx.doi.org/10.1007/978-3-642-22365-5_17
17. Grawrock, D.: Dynamics of a Trusted Platform: A Building Block Approach. Intel Press (February 2009), ISBN 978-1934053171
18. Gutmann, P.: An open-source cryptographic coprocessor. In: Proceedings of the 9th conference on USENIX Security Symposium - Volume 9. pp. 8–8. USENIX Association, Berkeley, CA, USA (2000)
19. Intel Corporation: Intel Trusted Execution Technology Software Development Guide (March 2011), http://download.intel.com/technology/security/downloads/315168.pdf
20. Kwan, P.C.S., Durfee, G.: Practical uses of virtual machines for protection of sensitive user data. In: Proceedings of the 3rd international conference on Information security practice and experience. pp. 145–161. ISPEC'07, Springer-Verlag, Berlin, Heidelberg (2007), http://portal.acm.org/citation.cfm?id=1759508.1759525
21. MacDonald, R., Smith, S., Marchesini, J., Wild, O.: Bear: An Open-Source Virtual Secure Coprocessor based on TCPA. Tech. Rep. TR2003-471, Dartmouth College (2003)
22. Marchesini, J., Smith, S., Wild, O., MacDonald, R.: Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Tech. rep., Department of Computer Science/Dartmouth PKI Lab, Dartmouth College (2003)

23. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient TCB reduction and attestation. In: Proceedings of the IEEE Symposium on Security and Privacy (May 2010)
24. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for TCB minimization. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008. pp. 315–328. ACM, Glasgow, Scotland UK (2008)
25. National Institute of Standards and Technology: Security requirements for cryptographic modules. FIPS PUB 140-3 (9 2009), draft
26. OpenTC Project Consortium: The Open Trusted Computing (OpenTC) project (2005-2009), http://www.opentc.net/
27. Pfitzmann, B., Riordan, J., Stueble, C., Waidner, M., Weber, A., Saarlandes, U.D.: The perseus system architecture (2001)
28. Reimair, F.: Trusted virtual Security Module. Master's thesis, Graz University of Technology (January 2011)
29. RSA Laboratories: PKCS #11 v2.20: Cryptographic Token Interface Standard. RSA Security Inc. Public-Key Cryptography Standards (PKCS) (June 2004)
30. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th USENIX Security Symposium. USENIX Association, San Diego, CA (2004)
31. Schiffman, J., Moyer, T., Shal, C., Jaeger, T., McDaniel, P.: Justifying integrity using a virtual machine verifier. In: ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference. pp. 83–92. IEEE Computer Society, Washington, DC, USA (2009)
32. Shi, E., Perrig, A., Van Doorn, L.: Bind: a fine-grained attestation service for secure distributed systems. In: 2005 IEEE Symposium on Security and Privacy. pp. 154–168 (2005)
33. Singaravelu, L., Pu, C., Härtig, H., Helmuth, C.: Reducing TCB complexity for security-sensitive applications: three case studies. In: EuroSys '06: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. pp. 161–174. ACM, New York, NY, USA (2006)
34. Smith, S.W.: Trusted Computing Platforms: Design and Applications. Springer Verlag (2005)
35. Smith, S.W., Weingart, S.: Building a high-performance, programmable secure coprocessor. Comput. Netw. 31, 831–860 (April 1999)
36. Toegl, R., Pirker, M., Gissing, M.: acTvSM: a dynamic virtualization platform for enforcement of application integrity. In: Chen, L., Yung, M. (eds.) Trusted Systems. Lecture Notes in Computer Science, vol. 6802, pp. 326–345. Springer Berlin / Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-25283-9_22
37. Trusted Computing Group: TCG TPM specification version 1.2 revision 103 (2007)
38. Tygar, J., Yee, B.: Dyad: A system for using physically secure coprocessors. In: Technological Strategies for the Protection of Intellectual Property in the Networked Multimedia Environment. pp. 121–152. Interactive Multimedia Association (1994)
39. Winter, J., Dietrich, K.: A hijacker's guide to the LPC bus. In: Petkova-Nikova, S., Pashalidis, A., Pernul, G. (eds.) EuroPKI. Lecture Notes in Computer Science, vol. 7163, pp. 176–193. Springer (2011)
40. Youn, P., Adida, B., Bond, M., Clulow, J., Herzog, J., Lin, A., Rivest, R.L., Anderson, R.: Robbing the bank with a theorem prover. Tech. rep., University of Cambridge (2005)