Implementing an Application-Specific Credential Platform Using Late-Launched Mobile Trusted Module

Sven Bugiel[*]
Royal Institute of Technology
Stockholm, Sweden
bugiel@kth.se

Jan-Erik Ekberg
Nokia Research Center
Helsinki, Finland
jan-erik.ekberg@nokia.com

ABSTRACT

Contemporary trusted execution environments provide a good foundation for implementing secure user credentials, but these are not properly bound to the application instances that implement their use. This paper introduces a framework for application-specific credentials and provides a prototype implementation using TCG MTM and DRTM technologies. Measurements and a security analysis is presented for the realised architecture.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*; K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*Security and Protection*

General Terms

Security

Keywords

DRTM, Flicker, MTM, credentials, secure hardware, trusted computing, software security

1. INTRODUCTION

Today, most end-user devices are strongly personal. Laptops, mobile phones, and PDAs are rarely used by more than one individual. At the same time, a typical user is required to hold and use a plethora of virtual credentials to identify himself to networks and services. These passwords and keys are mostly stored on the user's device. This causes a paradigm shift from an user-centric security model to a vastly simplified process-centric one. In absence of multiple users on the device, processes and applications rather than users take the central role in defining the achievable end-to-end security level of a network attach, service request, or peer-to-peer communication.

It is well known that the Trusted Computing Bases (TCB), i.e., the hardware and software basis that is critical for the security of the system, has in contemporary devices grown too large to be considered really secure. At the same time many hardware platforms have added trusted execution environments to support secure boot and credential processing. Technologies in this context include TPMs, PC processors with DRTM, and mobile processors with, e.g., ARM TrustZone.

This paper explores application-level binding of credentials in the specific context of TPMs/MTMs and DRTM. We present an architecture where each application running on the platform potentially can get its own credential engine (an MTM) for implementing application-specific secure storage, remote attestation, or even more complex services. The MTM code is rooted in a standards-compliant way on the platform, by mapping the MTM trust roots to a minimal TCB, realised with the Flicker architecture. Due to DRTM size constraints, the credential engine is split up into several pieces and Trust Domains (TD) are introduced, secured by TPM functionality.

The paper starts by presenting the used building blocks and a problem statement in Section 2. Section 3 presents the overall architecture in a piecemeal manner. An analysis of the achieved security, in the context of an attacker model, is put forward in Section 4. Section 5 superficially describes our reference implementation. Section 6 describes related work. The paper concludes with further work and conclusions.

2. TECHNICAL BACKGROUND

In this section, we provide the necessary technical background information, required to understand the security benefits offered by Flicker, the specific problem that we address in this paper, and our solution for the stated problem.

2.1 Trusted Platform Module

The cornerstone of TCG trusted computing is the *Trusted Platform Module* (TPM), an opt-in hardware device, that is deployed on the majority of today's server and business class platforms. The TPM is specified by the TCG (currently in version 1.2 [28]) in order to implement security design goals like the provision of secure storage or the attestation of the platform status to remote parties.

Platform State Attestation: TPM maintains the plat-

[*]the first author's work was done while being employed by Aalto University, Helsinki, Finland, as part of a project funded by Nokia

form's state in the abstract form of aggregated integrity measurements stored in *platform configuration registers* (PCRs) in the TPM. A PCR can not be simply overwritten with a new value, but only be *extended* with a new measurement. As a consequence, each PCR can hold an unlimited number of measurements and the order of the extensions will affect the aggregate value. Based on the PCRs, the TPM can *quote* the platform status to an external party. This attestation is a digital signature of the composite value of selected PCRs and parameters guaranteeing protocol freshness. The *Attestation Identity Key* (AIK) used for signing is dedicated for only this use and has been certified by a certification authority.

Wrapped Keys and Sealed Storage: The purpose of the sealing operation is to locally secure (encrypt, protect integrity, bind) externally supplied data or locally stored, private portions of other TPM keys (*"wrapping"*). The wrapped keys logically form a hierarchy with the *Storage Root Key* (SRK) as root. The private portion of the SRK never leaves the TPM.

Of special interest for this paper is, that the sealed data also contains information about the platform configuration at seal creation time *DigestAtCreation* (DaC) as well as a TPM-chip specific value to guarantee the device binding. Seals and wrapped keys require key-specific authentication and can be optionally bound to certain platform configurations (*DigestAtRealease* (DaR)).

DRTM: The *Dynamic Root of Trust for Measurements* (DRTM) in TPM v1.2 allows that the root of trust for the PCR measurements can be, even repeatedly, initialised at any point in time. The feature is enabled by a new CPU instruction, which creates a hardware-secured execution environment that can be attested by means of a TPM.

Both Intel and AMD have introduced this new instruction in CPUs available for commodity systems [11, 2]. The vendor implementations differ slightly, and for the remainder of this paper we focus on AMD platforms, but in principle the results apply also to Intel platforms.

The new AMD CPU instruction, called `SKINIT`, takes as its sole argument the physical memory location of a *Secure Loader Block* (SLB). That SLB must contain a standalone program image (max. 64kB), called *Secure Loader* (SL), to be atomically executed in the secure environment. `SKINIT` re-initialises the CPU such that execution of the SL is hardware-isolated in a way that it is secured against software-based attacks and also many hardware attacks [2].

The key step in the late-launch process is the measurement of the SLB by the TPM. `SKINIT` is the only instruction or computer event that creates the necessary bus cycles to trigger a dynamic reset of PCR17 to zero. Additionally, `SKINIT` hands over a copy of the SLB to the TPM, which now extends the reset PCR17 with a hash of the SLB. No software is able to reset the PCR17 to zero and its default value at boot is -1. Thus, PCR17 is at this point unique for each SLB identity. Implementations of the late-launch concept are Kauer's *OSLO* [13] and Intel's *tboot* [10].

2.2 Flicker

The Flicker architecture by Jonathan McCune *et. al.* [18] executes a small, security-sensitive part of programs under the protection of DRTM. The selected code is denoted as *Piece of Application Logic* (PAL). A special emphasis of the Flicker architecture is the minimisation of the mandatory trusted computing base of the PAL.

The Flicker project declared the following goals for their architecture: 1) Isolation of security-sensitive code from all other software and devices; 2) protection that is provable to a remote party; 3) meaningful attestation of exactly the code executed, its inputs and outputs; and 4) to have a minimal mandatory TCB for the security-sensitive code. Goals 1) and 2) are inherently fulfilled through DRTM. In Flicker, the initial mandatory software TCB consists of only 250 lines of code, to which the programmer of a PAL adds only the necessary security-sensitive code of his program. This *"bottom-up approach to the challenge of managing TCB size"* [18] forms the basis for satisfying goals 3) and 4). The hardware TCB includes the CPU, the RAM, the North Bridge, and the TPM. Thus, the attestation of the executed SL contains only meaningful, tractable information for a remote verifier to assess the trustworthiness of the PAL.

Architecturally, Flicker suspends the current execution environment (most likely the OS), executes the `SKINIT` instruction to set up the secure environment in which the PAL then executes, and afterwards resumes the previous execution environment. A kernel module provides virtual file system entries for the I/O communication between the application and PAL. A core library, against which PALs are linked, is responsible for setting up the secure environment and tearing it down. It is called *SLB Core* and forms the mandatory part of the software TCB for PALs.

Before the PAL resumes the legacy execution environment, it extends PCR17 with the hashes of its I/O parameters and of a well-known value. It thus provides the means to verify the parameters' integrity via remote attestation and to distinguish between measurements taken during the Flicker session and afterwards. It further prevents software in the legacy environment from unsealing data sealed specifically for PALs.

2.3 Problem Statement

In Flicker, applications are able to operate on their credentials under the hardware-based isolation of late-launch, since the execution environment is measured and credentials can be sealed by TPM with explicit binding to this measurement. E.g., the Flicker authors describe in [18] how a PAL is used in the context of SSH password authentication.

However, the sealing operation binds the state only to the PAL (and to the device), but not to the application the PAL belongs to. Although the credentials' secrecy is preserved at all times, deployed credentials can easily be misused by other applications. Any program with access to the right SLB and the sealed data can mount this attack.

In the same context, the Flicker designers identified the necessity for replay prevention mechanisms of the sealed state, and also sketched a solution based on PAL-specific counters deployed in the TPM non-volatile memory (NVM). Unfortunately, due to the scarcity of TPM NVM, the proposed replay protection scheme does not extend beyond 2-3 PALs per device, so a better scaling solution is needed.

3. ARCHITECTURE

In this section we present our architecture, which provides the link between applications, their respective sealed credentials, and the secure execution environment. Additionally, it implements a protection of the sealed states against replay

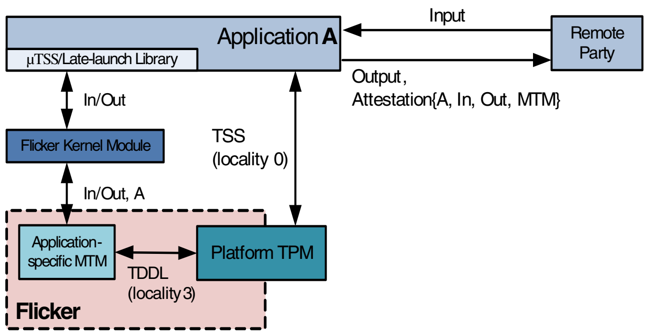


Figure 1: Big picture of the credential storage and usage by applications

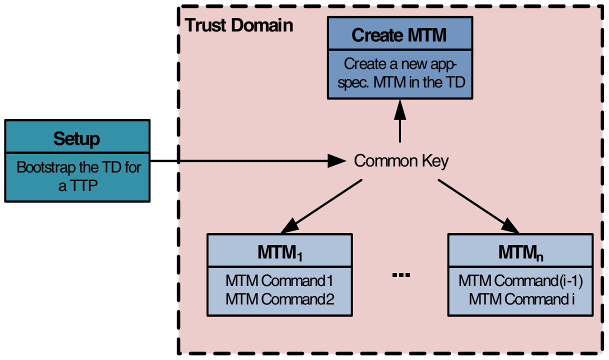


Figure 2: Set of MTM PALs, bootstrapped for a particular Trust Domain

attacks, suitable for the constrained resources in terms of secure monotonic counters.

3.1 Overview

Our main target is to provide all applications with the possibility to instantiate and execute their own *Mobile Trusted Module* (MTM) under the protection of DRTM, using a minimal TCB. The MTM can then be used by applications to store and to use their credentials in a secure and isolated manner.

An MTM is essentially an TPM version 1.2 [28]. Key feature of MTM [25, 26] is the standards-compliant support for several parallel MTMs and the opportunity to deploy MTM as software rather than as discrete hardware. The security properties for such a software MTM are specified in form of new *trust roots*, that must be met by the underlying platform in order to guarantee the required trustworthiness and protection of the software MTM.

Many implementations of software-based MTM exist [29, 9, 5]. We re-use an MTM [6], which introduced a software-based MTM approach tailored specifically for secure environments with constrained resources. Even though the MTM implementation is minimised, it does not fit into a single Flicker PAL/SLB. To overcome this problem, our architecture supports the notion of Trust Domains (TDs). Each TD is centred around a Trusted Third Party (TTP), such as an application vendor or independent party, that establishes its TD on the platform by means of a bootstrapping procedure. The TD will allow multiple PALs to share context. Applications/users can select a TTP during the instantiation of their own MTM, i.e., the device can host several TDs in parallel.

Figure 1 shows the credential usage by applications. These leverage their specific MTM PAL sets to use their credentials securely in protocols involving a remote party and to store their credentials securely when they are passive. Moreover, applications can make use of the attestation functionality of the TPM to quote the MTM execution in the same manner as described for PALs in Section 2.2, but additionally including the application ID. In contrast to the platform TPM, which is shared among all software and users of the platform, the MTM is application-specific.

The MTM is implemented in software, in compliance with the MTM specifications. This requires strong isolation from other software and the operating system running on the platform, and is achieved by means of DRTM in a Flicker session. Applications, that want to make use of their specific MTM, use an access library that is responsible for the communication with the Flicker kernel module and that implements specific parts of the TCG Software Stack (TSS) [27], denoted here as μTSS, since the MTM expects its commands in a specific format, defined in the MTM and TPM specifications. The kernel module computes, in addition to the original Flicker kernel module functionality, the application ID. This ID is implicitly passed as an argument to the MTM PAL, where it is used to bind credential execution in an application-specific manner.

The MTM makes use of the platform TPM, e.g., as a randomness source and to read/extend PCR17. In contrast to the legacy OS and applications, which access the TPM at locality 0 (or the legacy locality), the MTM will access the TPM at locality 3.

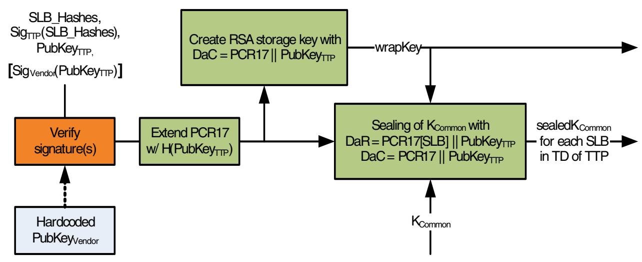
An MTM consists of an immutable part, the MTM command logic, and a mutable part, the MTM state. In this architecture, this fact is exploited by splitting the MTM up into these two parts [6]. Only the mutable MTM state is bound in an application-specific manner, while the MTM functionality is implemented as PAL, bound only to a TD. The mutable state constitutes one input to the PAL.

As mentioned, the MTM consists of a set of PALs and each piece implements a certain subset of the MTM logic or the instantiation of a new, application-specific MTM. For each TD, the PALs will share a domain-specific secret key, only known to PALs operating in that specific TD (illustrated in Figure 2). A special, domain-less PAL is responsible for bootstrapping the TDs.

3.2 Application Identification

A secure and unambiguous identification of the application that initiates a Flicker session is needed. In our architecture, this problem is solved based on IBM Research's *Integrity Measurement Architecture* (IMA) [19].

In a system that implements IMA, all loaded executable content, e.g., executables, scripts, or dynamically loaded libraries, is measured before execution. The measurements are stored in an ordered event log inside the kernel. Further, the measurements are extended into a certain PCR – usually PCR10 –, thus providing the means to report the platform state in an integrity-protected manner, based on the event log and the PCR10 attestation.



SLB_Hashes, Sig_{TTP}(SLB_Hashes), $PubKey_{TTP}$, [Sig_{Vendor}($PubKey_{TTP}$)]
Create RSA storage key with DaC = PCR17 || $PubKey_{TTP}$
wrapKey
Verify signature(s)
Extend PCR17 w/ H($PubKey_{TTP}$)
Sealing of K_{Common} with DaR = PCR17[SLB] || $PubKey_{TTP}$ DaC = PCR17 || $PubKey_{TTP}$
sealedK_{Common} for each SLB in TD of TTP
Hardcoded $PubKey_{Vendor}$
K_{Common}

Figure 3: Establishment of a common key for a set of PALs in the same Trust Domain

In our architecture, the IMA subsystem and the Flicker kernel module were patched, such that the Flicker module is able to compute the ID of processes that initiate a Flicker session. The kernel module inspects the task structure of the calling process inside the kernel run-time system for the inodes of the executable, all memory mapped files (e.g., dynamically loaded libraries), and configuration files. It then requests the measurements for theses files from IMA via a new, small API call and computes the aggregate hash of the returned values in a fashion similar to PCR extension. This aggregate hash forms the (load-time) application ID and is implicitly passed as input to the PAL, where it is used to bind the MTM state to the application. The patches add solely 14 LoC to IMA and 66 LoC to the Flicker kernel module, respectively.

3.3 Trust Domain Bootstrapping

The PALs, that belong to the TD of a particular TTP, share a secret common key, denoted K_{Common}, set up by the bootstrapping PAL for each TD. The sealing capabilities of the TPM and the special role of PCR17 during late-launch will guarantee the confidentiality, integrity, and authenticity of this shared key. Only the MTM PALs operating in the correct TD will have (authenticated) access to K_{Common}.

The work-flow of the bootstrapping PAL is as follows: It first generates the K_{Common} – an AES-128 key – for the new TD. Afterwards, it creates the data structure that holds the version information of the MTMs, which will operate in this TD, and it uses K_{Common} to protect these information. This procedure is explained in more detail in the subsequent Section 3.5. It further creates a TPM storage key, that is used to seal K_{Common} once for each MTM PAL, that will operate in the new TD. The encrypted version information, the storage key, and the seals are returned to the legacy environment for persistent storage.

The sealing of K_{Common} is illustrated in Figure 3. The bootstrap PAL requires as inputs a) an array of hashes, each identifying an MTM PAL that shall operate in this TD; b) a signature of this array by the TTP of the domain; c) the public key of this TTP; and d) optionally a signature of the TTP public key by the vendor of the PALs.

The bootstrap PAL first verifies the hash array signature. It further (optionally) verifies the authenticity of the TTP public key based on a signature of this key by the vendor and a vendor public key hardcoded in the PAL (which is thus also part of the PCR17 value at this point in time). Alternative approaches could be based on a secure UI showing, e.g., the fingerprint of the TTP public key or a secure remote attestation of the bootstrap PAL results.

Next, the bootstrap PAL extends PCR17 with the hash of the TTP public key. This hash identifies the TTP and thus the TD. The PAL then leverages the platform TPM to create a wrapped RSA storage key. The DigestAtCreation in the TPM key structure will be set to store the PCR17 value at the key creation time. Thus, a verifier with knowledge of a) the hash of the bootstrapping SLB and b) the TTP public key is able to verify that this key was actually created by the bootstrapping PAL as part of a Flicker session and that it is dedicated for the TD of the particular TTP.

In order to protect the confidentiality of the previously generated K_{Common}, it is sealed with the created TPM key, once for each individual PAL that should have access to it (based on the hash-array input). The DigestAtRelease for the seals are bound to the PCR17 values of the respective PAL digests, extended by the TTP public key hash. For the same reason as for the wrapped key, the DigestAtCreation information of the sealed blobs is set to the PCR17 value at sealing time.

In order to protect the version information of the MTM states, the bootstrapping PAL derives from K_{Common} of the TD a special key $K_{Version}$, which is used for symmetrically encrypting the information.

3.4 Common Key Unsealing

To operate, the MTM PALs require access to the TD-specific K_{Common}. For this purpose, the MTM PALs require certain input parameters. These are a) the wrapped TPM storage key created by the bootstrapping PAL for the TD of the application, b) the sealed K_{Common} for this domain, and c) a hash of the public key of the TTP of the TD the application executes in. Furthermore, the MTM PALs contain a hardcoded hash of the bootstrapping PAL, which hence is also part of their PCR17 value.

In order to unseal K_{Common}, the MTM PALs proceed as follows:

1. PCR17 is extended with the supplied hash of the TTP public key
2. The PCR17 value of the trusted Setup PAL, operating in this TD, is computed based on the hardcoded hash of the same and the hash of the TTP public key
3. If the DaC of the received wrapped TPM storage key or the received sealed K_{Common} does not correspond to the computed PCR17 value of the Setup SLB, then abort, since the trusted origin of the wrapped key or seal could not be verified
4. Unseal K_{Common} with the supplied TPM storage key. If the MTM SLB digest and the TTP public key hash are correct, then the DaR of the seal corresponds to the current PCR17 value and the unsealing succeeds. If it fails, one of the input parameters is incorrect or the MTM SLB was modified.

After unsealing K_{Common}, the MTM PAL derives an application-specific key K_{App} from K_{Common} plus the aggregate application hash provided by the kernel. This derived key is thus unique for a specific application in a specific TD.

3.5 State Integrity and Replay Prevention

For replay-protection, we deploy virtual monotonic counters using a TPM as described in [20]. One counter is main-

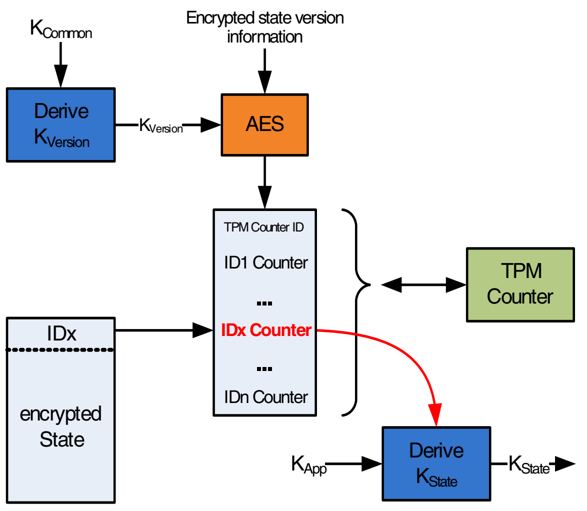


Figure 4: Derivation of the encryption key for an MTM state within a particular TD

tained for every instantiated MTM state (application) and the freshness of the states is verified based on these counters.

The state version information created by the Setup PAL consists of virtual counters. The Setup PAL allocates space in the TPM NVM for a reference counter and saves the space index in the version information. The TPM_DefineSpace command, for creating a new NVM space, offers fine grained access control settings. We restrict read and write operations to the NVM space based on the locality, here locality 3, and a 20 bytes authorisation value derived from $K_{version}$ and the string *"counter"*. As $K_{version}$ is derived from K_{Common}, the secret authorisation value is TD-specific.

Figure 4 depicts the details of how the key for decrypting the mutable MTM state (K_{State}) is derived and the role of the virtual counters in this context. First, $K_{Version}$ is derived from K_{Common} and used to decrypt the virtual counters in the state version information for this TD. The maximum counter value is set to 2^{31}, since the most significant bit is used to indicate if the counter is assigned or free. Every instantiated MTM state is tagged with a domain-unique ID during its instantiation and this ID forms the index to the virtual counter assigned to this MTM. Both the counter ID and the corresponding counter value are used with the application-specific key K_{App} to derive the latest key K_{State} for this application's MTM state. Hence, K_{State} changes with each update of the MTM state, thus changes to the state file are hidden. This further provides indirect means to verify the state freshness, because an incorrect counter ID or value results in an erroneous state decryption.

As proposed in [20], the freshness of the virtual counters is verified based on a physical monotonic reference counter, here based on the TPM NVM and monotonic counter functionality implemented in the PALs. The sum of the virtual counters and the reference counter are kept synchronised, and thus, if the sum and the reference counter value correspond to each other, the freshness of the virtual counters is assured, provided that the virtual counters are integrity protected.

In this implementation, the maximum number of possible TDs is limited by the maximum number of reference counters that can be created in the TPM NVM, which is on most contemporary TPMs very scarce. However, the number of MTMs in each domain is theoretically unlimited.

3.6 MTM State Instantiation

An application that wishes to use an MTM as its credential platform, is required to first instantiate an MTM state that is specifically bound to the application. The PAL for the state instantiation assigns a fresh virtual counter to the new state and increments it by one. Then the instantiation PAL creates a new MTM state for the application and appends information for its integrity validation. The state plus its integrity information is then symmetrically encrypted with the key K_{State}, that is derived from the application-specific key K_{App} together with the new counter value and counter ID of the state. Thus, the state information is application- and domain-specifically bound. The encrypted state and the updated state version information are returned to the application, which then is able to issue MTM commands on its specific MTM state.

In our architecture, the MTM state is created on the user platform and this prohibits a feasible pre-installation of the AIK or EK. The required credentials like the AIK can and are created in the context of the state instantiation. However, in order to be useful, the AIK has to be certified by a privacy CA. Normally, the privacy CA verifies the authenticity and the origin of a new AIK by means of the EK. In absence of such an EK, a new provisioning protocol is required. Such a protocol could leverage the platform's TPM EK or, in case of our architecture, also the TPM's binding capabilities with respect to the late-launched SLB's PCR17 value. The provisioning has to take into account that the AIK is bound to a specific application on the platform and this application's identity has to be part of the certification. For now, our architecture creates a new AIK for each new MTM state and we refer to future work for the design of a provisioning protocol of the AIK or the AIK certificate.

3.7 MTM Execution

Any application, that has successfully created its own MTM state, is able to leverage its MTM to store and use its credentials. While the MTM state forms the application-specific data, the MTM functionality is implemented in several SLBs. Each such SLB implements a different subset of the MTM commands from the TPM and MTM specifications. It is the task of an application-side library to load the correct SLB for the command that shall be executed.

The command execution consists of three major steps: 1) the decryption and validation of the received MTM state; 2) the MTM operation on this state; 3) the update of the state version information and the re-encryption of the updated MTM state. In addition, the MTM command will input any expected input required by the command, and produce the corresponding result according to specification. The keying needed to access and use the state as well as operations needed to validate freshness and authenticity of e.g. K_{Common} are equivalent to the procedures described in previous subsections.

As in Flicker, the MTM related input and output parameters as well as the application ID are extended into PCR17 for a remote attestation of this application-specific MTM execution.

4. SECURITY ANALYSIS

In this section we provide a security analysis of our architecture. We base the analysis on a defined attacker model and different attack scenarios. We conclude the section with an argumentation of how our architecture fulfils the Roots of Trust for a software-based MTM.

4.1 Attacker Model

For the attacker we assume the same model as in the Flicker infrastructure and TPM specifications.

The attacker is able to execute code at the highest system privilege level, including the `SKINIT` instruction. Moreover, the attacker has physical access to the target platform and is able to perform simple hardware-based attacks.

The primary goal of the attacker is to disclose applications' credentials. The disclosure of credentials constitutes a degradation of the security level in comparison to the original Flicker architecture and hence the analysis has to show that our architecture does not exhibit such a degradation.

The secondary goal is to break the link between the applications and their MTM in order to be able to misuse the protected credentials. The protection of this binding involves in our architecture parts of the OS and, moreover, our reference implementation does not comprise the security of the OS, for instance an authenticated boot to ensure boot time system integrity [1]. However, the analysis will show that the misuse of credentials is, even in absence of OS security mechanisms, mitigated.

4.2 Attack scenarios

In the following, we analyse attacks against our architecture, that might disclose or misuse applications' credentials. MTM state replay attacks are considered separately in Section 4.3.

The MTM PAL execution is inherently from the DRTM design resistant against software-based and against simple hardware-based attacks, such as malicious DMA devices or hardware debuggers. As explained in Section 2.1, late-launch disables all hardware debugging features, preventing an attacker from disclosing secrets with a hardware debugger. Moreover, it disables DMA access to the SLB region and ensures that no other software executes in parallel or concurrently.

The bus, that connects the chipset with the TPM, is slow enough to allow an attacker to eavesdrop on the TPM I/O with commodity, low-cost equipment. To counteract this attack, it is possible to establish a secure channel between the PAL and the TPM, called the *transport session* in the TCG specifications.

Hardware-based attacks, that target the applications or OS on the platform, can reveal secret information kept in memory. In our architecture we prevent this type of attacks by not storing sensitive information in the application, but rather keeping it inside the application-specific MTM. Thus, the sensitive information is protected at run-time by the security capabilities of DRTM and at passive time by the sealed storage capabilities of the TPM.

In our architecture, the trust among the MTM PALs is established by means of the TPM and DRTM capabilities. Figure 5 illustrates how these capabilities are applied in our architecture in order to resist attacks, which are based on modified PALs or input.

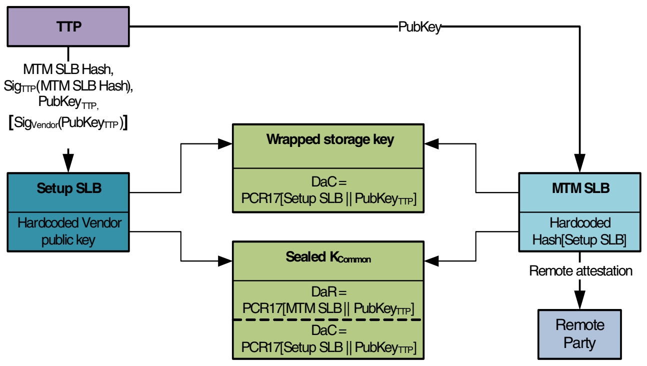

TTP
PubKey
MTM SLB Hash,
Sig$_{TTP}$(MTM SLB Hash),
PubKey$_{TTP}$,
[Sig$_{Vendor}$(PubKey$_{TTP}$)]
Setup SLB
Hardcoded Vendor public key
Wrapped storage key
DaC = PCR17[Setup SLB || PubKey$_{TTP}$]
Sealed K$_{Common}$
DaR = PCR17[MTM SLB || PubKey$_{TTP}$]
DaC = PCR17[Setup SLB || PubKey$_{TTP}$]
MTM SLB
Hardcoded Hash[Setup SLB]
Remote attestation
Remote Party

Figure 5: PAL trust establishment within the architecture

Any modification of the input to the Setup PAL is detected by the remote party during remote attestation, thus results in a denial of service, but no misuse or disclosure of credentials. For instance, an exchanged TTP public key and/or different selected MTM SLBs in the hash array cause a different PCR17 value and are hence reflected in the remote attestation. The remote party is thus able to detect untrusted values and abort any protocol run, invalidating all credentials in this untrusted domain.

The Setup and MTM SLBs are stored on persistent storage and thus accessible by the attacker. Modifications of the SLBs are, however, detected and result in a termination of the MTM operation. A modified Setup SLB has a different PCR17 value and hence generates a DigestAtCreation (DaC) value in the wrapped storage key and the seal, that is different from the trusted one, which is hardcoded into the MTM SLBs. Thus, the MTM SLBs are able to detect the modification to the Setup PAL and abort their operation. Similarly, modifications to the MTM SLBs result in a termination, because the PCR17 value of a modified MTM SLB is different from the value used during the setup. Thus, the DigestAtRelease (DaR) of the sealed common key prevents the TPM from unsealing the common key.

Software-based attacks could target the operating systems parts, involved in our architecture, or the applications, which make use of a late-launched MTM. Attack software might compromise the Flicker kernel module, the IMA subsystem, or the process management of the kernel in such a way, that the computed application ID is incorrect and the link between applications and their MTM is broken. In this scenario, our architecture provides the same security level as the original Flicker architecture. However, in contrast to Flicker, our architecture forces the attacker to mount sophisticated attacks. In conjunction with integrity protecting mechanisms for the OS, this kind of attacks could be further mitigated (e.g., only possible at run-time) or completely prevented.

Furthermore, the attacker could target the applications on the system. If the application is compromised before it creates its own MTM, the liability lies at the remote verifier to detect modified applications and abort. However, attacks at run-time remain undetected. The fundamental issue is the time-of-measurement-time-of-use problem. IMA measures the executables and libraries at load-time. Thus, if malware compromises a running process, this modification is not reflected in the measurement value. Such a compromised application is nevertheless still able to use its specific MTM. Although the compromised application can misuse its credentials, it is, nevertheless, not able to disclose them.

4.3 MTM state replay attacks

The architectural design of the replay prevention is explained in detail in Section 3.5. The key $K_{Version}$ is used to protect the virtual counters for the MTM states. The integrity of the counters is assured based on a monotonic TPM NVM based reference counter and the encryption of the version information. The virtual counter ID and value, which are assigned to an MTM state, are used to derive the encryption key K_{State} for this particular state.

As a consequence, any modifications to the MTM state file or the encrypted virtual counter data structure, both stored on persistent storage, result in failed integrity checks of the MTM state and the termination of the MTM operation.

Manipulations of the reference counter, implemented in the TPM non-volatile memory, are prevented by the TPM's design and the properties of the virtual counter protection, that is implemented in our architecture. Write access to the NVM space requires the correct authentication value $K_{Counter}$, which is derived from $K_{Version}$. An attacker could try to acquire $K_{Counter}$ either indirectly through cryptanalysis of the AES-128 encrypted virtual counters or directly through a brute-force attack against the authentication value of the TPM NVM space. The former is prevented by the cryptographic properties of the AES-128 implementation and the latter is prevented by the TPM's *"anti-hammering"* methods, which are specifically designed to block brute-force attacks against TPM authentication values.

4.4 MTM Roots of Trust

To conclude the security analysis, we analyse how our architecture meets the required Roots of Trust for a secure implementation of a software-based MTM.

Root of Trust for Enforcement (RTE): The RTE is the RT responsible for constructing all other roots, which are composed of allocated resources. It must consist of dedicated resources and be supplied with an EK and/or AIK. The dedicated resources in our architecture are DRTM and the platform's TPM, which provides the EK and AIK.

Root of Trust for Storage (RTS): The RTS provides the PCRs and protected storage for the MTM. It is established by means of a device secret. In our architecture, the RTS is constructed by the RTE. The MTM state implements the PCRs of the MTM. The protected storage is provided by K_{Common} and the keys derived from it, which are only accessible to the legitimate PALs.

Root of Trust for Reporting (RTR): The RTR reports the measurements stored by the RTS. Further, it holds the secrets required for attestation. The keys of the MTM and the measurements are stored inside the MTM state and thus they are protected by the RTS. The attestation functionality constitutes of some immutable or protected code. In our architecture, this code is implemented in the MTM PALs and thus authenticated plus integrity protected by the RTE.

Root of Trust for Verification (RTV): The RTV is responsible for checking measurements against Remote Integrity Metrics and extending the integrity metrics into PCRs. Some immutable or protected code forms the RTV and in our architecture this code is implemented in the MTM PALs and is hence protected by the RTE.

SLB	$\bar{x}$ (ms)	σ (ms)
Setup	31649.48	22641.21
State instantiation	5893.63	1163.15
MTM SLBs	4218.69	3.7

Table 2: Mean SLB run-times and their standard deviation (10 measurements, excluding MTM command execution time)

5. IMPLEMENTATION

Our implementation was tested on a Dell PowerEdge T105 with a Quad-Core AMD Opteron 1356 CPU (2,1 GHz) and an ST Microelectronics ST19NP18 TPMv1.2 chip. Both machines run Ubuntu Linux 9.04 with a vanilla kernel version 2.6.33, whose IMA subsystem has been modified for the architecture in this paper.

5.1 Size figures

The main programming language used for the implementation is ANSI C99. The only component, which is not written in ANSI C99, but Assembly x86 code, is the SLB core library provided by Flicker.

We used the open-source SLOCCount[1] tool for the measurement of the lines of code.

Table 1 presents the sizes in lines of code of the particular components of the PALs in our architecture. The size for the MTM applies to a monolithic implementation and thus constitutes the lines of code that are distributed among the MTM PALs. In our implementation exist three MTM PALs, each containing approximately one third of the monolithic implementation. The mandatory software TCB in Flicker is the SLB Core with 223 lines of code, to which we add in our architecture the other components.

5.2 Performance

The mean run-times and the corresponding standard deviations of the SLBs in our architecture are listed in Table 2. The measurements were computed based on the time stamp counter of the CPU. MTM command execution time is excluded from the measurements, because it greatly depends on the kind of command, that is executed. The upper bound, however, is approximately the RSA-1024 private key generation time, which is required for the creation of a wrapped key by the software-based MTM.

Further, it should be noted that these measured performances are too fast, since the TPM on our test machine measures SLBs incorrectly over a length of 0 bytes. `SKINIT` yields therefore a mean execution time of 0.003 ms. The estimated performance benefit of the faulty execution, compared to the correct one, is approximately 133.35 ms, if we assume an SLB size of 48kB and a linear duration increase in Table 2 of [18].

A critical factor for the maximum run-time of a Flicker session are the disabled interrupts during the session. The SLB execution suspends the legacy OS and unhandled interrupts may cause a system crash during resumption. In our Linux setup, the network card driver in Linux froze the

[1]http://www.dwheeler.com/sloccount/

Component	Description	LoC
SLB Core[a]	Environment initialisation, memory protection, ring 3 switch, clean up, resume OS	223
TPM Driver[b]	Small TDDL(I) for communication with the platform's TPM	226
TPM Commands[b]	Required TPM commands	703
Key chain	Verifying the wrapped storage key and sealed K_{Common}, deriving the keys	464
Version management	Virtual counter management and verification, derive counter authorisation	148
State management	MTM state instantiation, state integrity and confidentiality	114
MTM[c]	Monolithic MTM implementation	3790
Crypto	Cryptographic primitives	4775
Utilities	Standard library functions, parameter I/O	228

Table 1: The size in lines of code that each component of the MTM and Setup PALs adds([a]provided by Flicker; [b]provided by Flicker, but modified/extended; [c]available from [6], but adapted/extended)

Command	$\bar{x}$ (ms)	σ (ms)
TPM_CreateWrapKey	27864.63	21305.600
TPM_LoadKey12	2616.50	0.609
TPM_GetPubKey	89.84	1.417
TPM_Seal	282.38	0.684
TPM_Unseal	954.39	0.576
TPM_FlushSpecific	42.01	0.370
TPM_ReadPcr	18.24	0.435
TPM_Extend	24.62	0.494
TPM_NV_DefineSpace	192.03	0.699
TPM_NV_ReadValueAuth	18.14	0.436
TPM_NV_WriteValueAuth	83.25	1.770
TPM_OIAP	18.06	0.126
TPM_OSAP	36.06	0.076
TPM_GetRandom (20 bytes)	36.16	1.453

Table 3: Mean performance and standard deviation in milliseconds of the required TPM Commands to the platform TPM (50 measurements)

Function	$\bar{x}$ (ms)	σ (ms)
SHA1 of 2048 Bytes	0.06	0.003
HMAC-SHA1 of 2048 Bytes	0.06	0.003
RSA-1024 private key generation	1650.59	1354.522
RSA-1024 public key generation	0.02	0.001
RSAES-PKCS1-v1_5 (1024 bits) Encryption of 64 Bytes	2.04	0.01
RSAES-PKCS1-v1_5 (1024 bits) Decryption of 64 Bytes	47.60	0.494
RSASS-PKCS1-v1_5 (1024 bits) Signing of 64 Bytes	47.47	0.544
RSASS-PKCS1-v1_5 (1024 bits) Verification of 64 Bytes	2.04	0.008
AES-128 CBC Mode Encryption of 2320 Bytes	0.23	0.004
AES-128 CBC Mode Decryption of 2320 Bytes	0.23	0.003

Table 4: Mean performance and standard deviation in milliseconds of the included cryptographic library functions (50 measurements)

system on resumption with Flicker sessions longer than 8 seconds, and e.g. the Setup PAL (that generates an RSA key) had to be run with the network cable unplugged.

Table 3 lists the mean execution times and their standard deviation for the required TPM commands in our architecture. The measurements have been taken with the `gettimeofday` command inside a small benchmark program in the legacy OS on our test machine.

Most of the TPM commands are very fast and also constant in their performance. Exceptions are the commands involving private key operations, like unsealing, key-loading, or especially the wrapped key creation. The latter one is not only extremely inconstant, but also exhibits a very high mean execution time.

Table 4 lists the mean execution times and the corresponding standard deviation of the cryptographic primitives in our crypto library. The measurements were taken with the `gettimeofday` command inside a benchmark program running in legacy OS, an upper bound for the speed when run under DRTM. The cryptographic primitives execute on one CPU with 2.1 GHz clock rate and perform therefore very fast and with negligible deviations. The only exceptions are, similar to the TPM commands, the functions involving private key operations.

In comparison, the TPM commands execute slower than our crypto library and form the bottleneck for the execution time of our MTM PALs. Tests have shown, that the unsealing of K_{Common}, which involves the loading of the TPM storage key and the TPM unsealing operation, account for approximately 90% of the run-time of each MTM PAL.

6. RELATED WORK

In this section different approaches for securely storing and executing credentials are presented.

One solution is a hardware-secured credential repository, which is based on the MyProxy network proxy [16], and designed for usage inside a grid. In this setup, the users' long-term credentials are stored remotely and are secured by a security co-processor on the remote machine. After user authentication, short-term credentials are derived from the long-term credentials for authentication in the grid. These repository systems are of course user-centric, and do not store or execute the credentials in an application-specific manner.

Another approach for credential storage and use is the On-board Credentials (ObC) by Nokia Research Center, Finland [15], an open design where anyone is allowed to design and deploy their own credential mechanism without the compulsory involvement or approval of the device manufacturer or a TTP. ObC can be deployed on top of DRTM and is thus, from a functional point of view, a suitable option as credential platform in our architecture. However, ObC does not provide standardised interfaces for applications at the same level as MTM.

In general, a microkernel or hypervisor based approach has been adopted by several projects in order to provide a higher level of security through isolated, protected execution environments and a minimal, simpler TCB [8, 3, 24]. One solution, that specifically targets the security of users' credentials is the TruWallet architecture presented in [7]. It is based on the secure execution environment, provided by an underlying security microkernel, to implement a compartmentalised wallet application.

Although hypervisors and microkernels are substantially smaller than commodity OSes and thus in the size range for formal verification, the assumption of a formally verified hypervisor/microkernel is at this time only conditionally valid. The currently most famous example for a formally verified microkernel is seL4, which has approximately 7500 LoC [14]. The verification implies the fulfilment of critical security properties, but the actual goal of the verification is to prove functional correctness and not the security. Thus, certain security-relevant properties, such as covert channels, have not be formally validated in this contexts. Moreover, the proof does not cover the x86 architecture.

The major chip vendors have also improved their hardware support for virtualisation and platform security. This includes Intel's VT/TXT [11] or AMD's AMD-V/SVM [2], which are adopted by projects like Xen or VMWare. Recent research projects that make specific use of the improved HW features include the SecVisor and TrustVisor project by CMU [22, 17]. The SecVisor is an architecture that implements a tiny hypervisor (about 1100 LoC) that uses the CPU supported memory virtualisation capabilities for isolating the memory region of a commodity OS kernel from the rest of the platform software. The TrustVisor architecture [17] settles in the middle-ground between Flicker and a full-fledged hypervisor regarding the TCB perimeter. TrustVisor (about 6300 LoC) adopts the Flicker concept of PALs and provides applications with the possibility to register their PALs at the TrustVisor. The PALs then execute in total isolation from the rest of the system's software and malicious DMA-capable peripherals. The isolation is enforced by CPU virtualisation features. In addition, each PAL can be attested remotely, based on a DRTM measurement of the TrustVisor, extended with the PAL measurement. Moreover, the TrustVisor provides each PAL with it's own software TPM. The main motivation for the TrustVisor, is to avoid the performance penalty of Flicker, that has to be accepted in order to achieve the extremely minimal TCB.

Further, it has been shown, that compartmentalised applications under a hypervisor are also suitable to implement platforms like MTM [29, 9, 21]. Such compartments are also easy to measure, enabling advanced security features, as shown by, e.g., TruWallet. However, the essential prerequisite is the integrity of a trusted hypervisor/microkernel at all time. Contemporary research investigating this issue are, for instance, [12, 4, 23]. In contrast, compromised software at the highest system privilege level in a Flicker based architecture can not jeopardise the secrecy of applications' credentials. Also, as long as full compartmentalisation of applications is not used, none of these above mentioned architectures, with the exception of TrustVisor, explicitly address application-binding as a requirement.

7. FUTURE WORK

In this section, we briefly discuss possible future optimisations and open research issues that we foresee.

A first enhancement of the security would be to provide user interaction in order to enable multi-factor authorisation and improve security. For example, the well-known values used for TPM command authorisation could be substituted by user supplied passwords. Moreover, the credentials are then bound in an application- and user-specific manner and thus facilitate usage on multi-user systems. The prerequisites for secure user I/O are a trusted channel and a trusted path between the user and the secure environment.

A further open issue for the application identification relates to applications consisting of interpreted scripts. The kernel's process management does not provide information about script files that are interpreted, but solely the interpreter. Since the script file obviously forms the application, it has to be part of the application ID. Thus, the current application ID for scripts is incomplete.

In terms of applicability, the application identification ideally is based on some more flexible mechanism than a hash measurement, which prohibits easy updates of applications and libraries without losing the bound credentials.

8. CONCLUSIONS

In this paper, we presented our solution for an architecture that provides application-specific credentials, deployed by means of TCG DRTM, and employed the Flicker architecture as the main building block for our reference implementation.

We combined a minimal TCB with a standardised credential platform and deployed this in an application-specific manner with no degradation of the security level provided by the underlying security architecture.

The concept of Trust Domains emerged as a reusable design pattern for late-launched credential platforms.

9. ACKNOWLEDGEMENTS

We would like to thank Jonathan McCune of CMU for providing the Flicker source code and sharing his experience when problems with the Flicker architecture were encountered. We are also grateful for the feedback and comments from the anonymous reviewers.

10. REFERENCES

[1] TrustedGRUB. `http://sourceforge.net/projects/trustedgrub/`.

[2] Advanced Micro Devices. AMD64 Virtualization: Secure Virtual Machine Architecture Reference Manual, May 2005. Publication mo. 33047, rev. 3.0.

[3] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Seattle, WA, USA, March 2008.

[4] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks. In *STC '09: Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54, New York, NY, USA, 2009. ACM.

[5] Kurt Dietrich. An integrated architecture for trusted computing for java enabled embedded devices. In *STC*

'07: Proceedings of the 2007 ACM workshop on Scalable trusted computing, pages 2–6, New York, NY, USA, 2007. ACM.

[6] Jan-Erik Ekberg and Sven Bugiel. Trust in a Small Package: Minimized MRTM Software Implementation for Mobile Secure Environments. In *STC '09: Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 9–18, New York, NY, USA, 2009. ACM.

[7] Sebastian Gajek, Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. TruWallet: Trustworthy and Migratable Wallet-Based Web Authentication. In *STC '09: Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 19–28, New York, NY, USA, 2009. ACM.

[8] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, 2003.

[9] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261, January 2008.

[10] Intel Corporation. tboot, 2009. `http://tboot.sourceforge.net/`.

[11] Intel Corporation. Trusted eXecution Technology (TXT) – Measured Launched Environment Developer's Guide, December 2009.

[12] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: Policy-Reduced Integrity Measurement Architecture. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28, New York, NY, USA, 2006. ACM.

[13] Bernhard Kauer. OSLO: Improving the security of Trusted Computing. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–9, Berkeley, CA, USA, 2007. USENIX Association.

[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, New York, NY, USA, 2009. ACM.

[15] Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. On-board credentials with open provisioning. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 104–115, New York, NY, USA, 2009. ACM.

[16] M. Lorch, J. Basney, and D. Kafura. A hardware-secured credential repository for Grid PKIs. In *CCGRID '04: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 640–647, Washington, DC, USA, 2004. IEEE Computer Society.

[17] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010.

[18] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, New York, NY, USA, 2008. ACM.

[19] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[20] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 27–42, New York, NY, USA, 2006. ACM.

[21] Andreas U. Schmidt, Nicolai Kuntze, and Michael Kasper. On the deployment of Mobile Trusted Modules, 2007.

[22] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, New York, NY, USA, 2007. ACM.

[23] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code execution on Legacy Systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–16, New York, NY, USA, 2005. ACM.

[24] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, 2006.

[25] Trusted Computing Group. Mobile Trusted Module (MTM) Specification. Version 1.0 Revision 6, 26 June 2008.

[26] Trusted Computing Group. TCG Mobile Reference Architecture Specification. Version 1.0 Revision 1, 12 June 2007.

[27] Trusted Computing Group. TCG Software Stack (TSS). Specification Version 1.2 Level 1 Errata A, 7 March 2007.

[28] Trusted Computing Group. Trusted Platform Module (TPM) Main Specification. Version 1.2 Revision 103, 9 July 2007.

[29] Johannes Winter. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30, New York, NY, USA, 2008. ACM.