Technical University of Denmark
Department of Informatics and Mathematical Modelling

The Royal Institute of Technology
Telecommunication Systems Laboratory

Sven Bugiel

Using TCG/DRTM for application-specific credential storage and usage

Master's Thesis
Helsinki, June 25, 2010

Supervisors: Professor Christian Damsgaard Jensen, Technical University of Denmark
Professor Peter Sjödin, The Royal Institute of Technology
Instructor: Jan-Erik Ekberg, M.Sc. (Tech.), Nokia Research Center

(Sven Bugiel)
Helsinki, June 25, 2010

Abstract

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 has lead to the paradigm shift from an user-centric security model to a vastly simplified process-centric one. In absence of multiple users on the device, processes rather than users will take a central role in defining the achievable security level. The work in this thesis concentrates on the application-binding aspects between device security hardware and applications running on top of the device's operating system.

The hardware and software basis that is critical for the security of the system is denoted as the Trusted Computing Base (TCB). However, the size and complexity of the TCB in contemporary mainstream operating systems provides malware with a big attack surface to compromise system security and to gain illegitimate access to credentials.

The Flicker architecture by Jonathan McCune et.al. of CMU provides the means to securely execute a small piece of security sensitive software (PAL) in an isolated execution environment with a minimal, secure TCB. Thereby, Flicker makes use of the TCG DRTM technology, available in contemporary PC chipsets. However, the link between an application, that leverages Flicker, and its PAL is missing. This raises the security issue that deployed credentials can be easily misused by other software on the system, such as malware.

This thesis presents a Flicker based architecture, that establishes this missing link by providing a framework in which each application is bound to a software-based Mobile Trusted Module (MTM), effectively building a system with application-specific secure credentials. The viability of the architecture is confirmed by presenting a prototype implementation together with a related attacker model and security analysis.

Acknowledgements

I am grateful to my supervisors, Professor Christian Damsgaard Jensen of DTU and Professor Peter Sjödin of KTH, for their kind help and guidance during this thesis.

I would like to express my sincere gratitude to the members of the Trustworthy Mobile Platforms group at Nokia Research Center, Finland. My special thanks goes to Jan-Erik Ekberg, my instructor, for his outstanding expertise and support during this time, his excellent reviews of my drafts, and for making the work real fun. Big thanks also to Dr. N. Asokan for giving me the opportunity to conduct my thesis in his group and for his much appreciated feedback on my architecture designs. To the remaining members of the group, thank you for your valued advice and for creating such a nice atmosphere to work in.

I would also like to profoundly thank Jonathan McCune of CMU for providing the Flicker source code and sharing his experience when I encountered problems with the Flicker architecture.

Finally, I am deeply grateful to my parents for their unconditional love and support and their faith in me. Without them, I would not have been able to go my own way during these last years.

Sven Bugiel
Helsinki, Finland
June 25th 2010

Abbreviations and Acronyms

AES	Advanced Encryption Standard
AIK	Attestation Identity Key
AP	Application Processor
BSP	Bootstrap Processor
BIOS	Basic Input Output System
CA	Certificate Authority
CBC	Cipher-Block Chaining
CPU	Central Processing Unit
CRTM	Core Root of Trust for Measurement
DaC	DigestAtCreation
DaR	DigestAtRelease
DEV	Device Exclusion Vector
DMA	Direct Memory Access
DRTM	Dynamic Root of Trust for Measurement
EK	Endorsement Key
GDT	Global Descriptor Table
GIF	Global Interrupt Flag
GUI	Graphical User Interface
HMAC	Keyed-Hash Message Authentication Code
I/O	Input/Output
ICH	Input/Output Controller Hub
IMA	Integrity Measurement Architecture
IPL	Initial Program Loader
KDF	Key Derivation Function
LoC	Lines of Code
LPC	Low Pin Count
MAC	Mandatory Access Control
MCH	Memory Controller Hub
MBR	Master Boot Record
MLTM	Mobile Local Owner Trusted Module

MRTM	Mobile Remote Owner Trusted Module
MTM	Mobile Trusted Module
NVM	Non-Volatile Memory
OAEP	Optimal Asymmetric Encryption Padding
ObC	On-board Credentials
OS	Operating System
PAL	Piece of Application Logic
PCR	Platform Configuration Register
PRF	Pseudorandom Function
PSS	Probabilistic Signature Scheme
RAM	Random access memory
RIM	Reference Integrity Metric
ROM	Read-only memory
RSA	Algorithm for public-key cryptography
RT	Root of Trust
RTE	Root of Trust for Enforcement
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
RTV	Root of Trust for Verification
RVAI	Root Verification Authority Information
SHA	Secure Hash Algorithm
SL	Secure Loader
SLB	Secure Loader Block
SMM	System Management Mode
SMP	Symmetric Multiprocessing
SRK	Storage Root Key
SRTM	Static Root of Trust for Measurement
SVM	Secure Virtual Machine
TC	Trusted Computing
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TD	Trust Domain
ToCToU	Time of Check to Time of Use
TPM	Trusted Platform Module
TrEE	Trusted Execution Environment
TRNG	True Random Number Generator
TSS	TCG Software Stack
TTP	Trusted Third Party
TXT	Trusted eXecution Technology
VMM	Virtual Machine Monitor

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Users are today required to identify themselves to a plethora of Internet and net-based services, such as online banking, email accounts, or e-governmental institutions. For that purpose, they are required to manage and apply a number of different "virtual" credentials – in the most cases passwords, but also cryptographic keys and certificates are possible. In general, these credentials are defined as cryptographic objects and related data that are used to establish the identity of a party and secure communications over the Internet [45].

Usually, these virtual credentials are used and stored on the commodity PC platforms of the users. For example, cryptographic keys have to be stored on some kind of persistent storage, normally the PC's hard disk, passwords be entered via a user interface, or cryptographic operations, like decryption, must be performed. The hardware and software basis that is critical for the security of the system and that has inevitably to be secure in order to enable the secure application and storage of credentials is denoted as the Trusted Computing Base (TCB). However, the size and complexity of modern system- and application-level software entails an ill-defined TCB that comprises the bulk of the operating system. It further provides an attacker or malware with a big attack surface to compromise the system integrity and confidentiality, rendering the secure credential storage and usage basically infeasible.

Industry and research have introduced a number of solutions to counter this problem and to facilitate a more secure computing. One example are hyervisor or microkernel based architectures that implement a small TCB by compartmentalisation of the software. Another example are the specifications of the Trusted Computing Group (TCG), which have been widely adopted today, e.g., the cornerstone of TCG trusted computing, the Trusted Platform Module (TPM), is deployed on almost every new business or server class platform. One of the newer concepts of TCG, that has been realised in cooperation with the major chip vendors, is Dynamic Root of Trust for Measurement (DRTM), also known as “late-launch”. DRTM provides the possibilities to establish a hardware protected and extremely isolated execution environment in order to bootstrap a secure OS with a minimal TCB. In conjunction with a TPM it also provides the means for secure storage and the remote attestation of the execution environment.

The Flicker research project of the Carnegie Mellon University (CMU) leverages DRTM to establish such an isolated environment, in which applications can execute sensible computations with protection from software-based and even simpler hardware-based attacks. The Flicker concept thus offers a suitable approach to provide applications with the possibility to securely use and store credentials in isolation from all other application- and system-level software on their platform. However, the binding between the execution environment and the applications is missing, raising security issues like the misuse of deployed credentials by other applications, including malware.

In this thesis we propose an architecture, based on the Flicker infrastructure, that provides this link and enables the application-specific storage and usage of credentials by means of a software-based Mobile Trusted Module (MTM) as credential platform.

The motivation to focus on an application-specific binding of the execution environment instead of an user-specific is the increasing personalisation of computing platforms. While it is common for a long time in the mobile devices sector, e.g., mobile phones, that people own their personal device instead of sharing one with other people, this also becomes more and more the standard in the non-mobile sector. Under this assumption, credentials on a particular platform are used by only one user and are usually intended for usage by only one specific application. A possible scenario would be an online banking application that establishes an authenticated and secure channel to the banking server. It therefore requires an isolated, secure execution environment for its cryptographic operations and further the means to store its credentials securely during passive periods. A two-factor authentication, that enables multi-user scenarios through user-specific authentication remains a possible add-on to an application-specific architecture.

1.2 Thesis Structure

This thesis is structured as follows. In Chapter *2 Technical Background and Problem Statement*, we elaborate on the Flicker infrastructure and the necessary technical details in order to understand the security benefits that Flicker provides for our architecture. Based on this elaboration, we state in more detail the problem addressed in this thesis. Afterwards, we compare in Chapter *3 Related Work* different solutions to the stated problem, based on selected criteria. Further, we show alternatives to certain parts of the proposed architecture and justify our design decisions. The architecture is explicitly presented in Chapter *4 Architecture*. We then analyse it for its security properties in Chapter *5 Security Analysis* and base the analysis on an attacker model and different attack scenarios. In Chapter *6 Implementation*, we present numbers and figures about the actual implementation and discuss in Chapter *7 Future Work* further improvements and open research questions. We conclude the thesis in Chapter *8 Conclusion*.

2

Technical Background and Problem Statement

In this chapter, we present important technical background information for the understanding of the later presented architecture and its analysis. The main focus lies on the Flicker infrastructure (Section 2.4), which is an essential component of the architecture in this thesis. Flicker, on the other hand, is built on top of Trusted Computing Group DRTM and Trusted Platform Module (TPM), which both play an essential role and whose relevant parts are explained separately (Sections 2.1 and 2.3). Further, the Mobile Trusted Module (MTM) requires some explanation, since it is used as the credential platform in our architecture (Section 2.2).

2.1 Trusted Platform Module

The *Trusted Platform Module* (TPM) is an opt-in hardware device, usually mounted on the motherboard and connected to the system via the Low Pin Count (LPC) bus, thus available before any other initialised device during the bootstrap.

The TPM is specified by the Trusted Computing Group (TCG) (currently in version 1.2 [117]) in order to implement security design goals [26, chap. 1] such as the secure reporting of the environment, the provision of secure storage, or the attestation of the platform status to remote parties. For that reason, the TPM is equipped with a number of essential components. To name the major components: a) an execution engine for the specified commands; b) a True Random Number Generator (TRNG); c) Non-Volatile Memory (NVM);

d) a cryptographic co-processor; e) Platform Configuration Registers (PCRs); and f) a small number of monotonic counters.

Applications can access the TPM in an easy and direct fashion by means of the TCG Software Stack (TSS) [116], which also provides more advanced features like the access to an TPM on a remote machine or the persistent storage of key objects on disk.

An TPM is a complex device and in this thesis the focus is limited to the thesis relevant details. Of special importance for the understanding of our presented architecture are the details of the sealed storage and the platform state attestation.

2.1.1 Platform State Recording

The TPM implements the means to securely store and attest the state of the platform.

The state is maintained in the abstract form of aggregated measurements inside the TPM's PCRs. This abstraction is necessary, since keeping a full record of the platform's operating system and applications' versions – or similar logging mechanisms – would surely exceed the TPM's storage capabilities.

A Platform Configuration Register is an 160-bit shielded storage location inside the TPM, of which an TPM in version 1.2 contains exactly 24. The particular integrity metrics that form the platform state are measured as cryptographic hashes. A very important fact here is that a PCR can not be simply overwritten with a new value, but only be *extended* with a new measurement. The new value of the extended PCR is computed as follows:

$$PCR_i^{NEW} = H(PCR_i^{OLD} \,||\, value\ to\ add)$$

where H is a cryptographically secure hash function with a block size of 160-bits, e.g., SHA-1 for TPMs in version 1.2, and $||$ denotes concatenation. One important consequence of this extension mechanism is that each PCR can hold an unlimited number of measurements and that the order of the extensions will affect the aggregate value.

Before the v1.2 specification, PCRs could only be reset to zero at system reboot time. But with the v1.2 specification the TCG introduced the concept of *dynamic* PCRs. In particular, the lower 16 PCRs are considered *static*, i.e., reset still only at system reboot, while the higher 8 PCRs are dynamic,

Locality	**Usage**	**PCRs that can be Reset by the Locality in addition to "Any"**	**PCRs that can be Extended by this Locality in addition to "Any"**
Any	—	16,23	0-16,23
0	The legacy environment for the Static RTM and its chain of trust	—	—
1	An environment for use by the Trusted Operating System	—	20
2	The "runtime" environment for the Trusted Operating System	20,21,22	17-22*
3	Auxiliary components. Use of this is optional and, if used, it is implementation dependent	—	17-20*
4	Trusted Hardware. This is the Dynamic RTM	17-20	17,18*

Table 2.1: *TPM localities (Table adapted from [115, p. 8] and [114, p. 30]); *the TPM ensures after a reset of PCR17, that PCR17 is extended at least once by locality 4 before the localities 2 and 3 are allowed to extend this register*

meaning that they can be reset under certain circumstances without a system reboot. To distinguish between a reset at reboot time and a dynamic reset, the former one sets the dynamic PCRs to the value -1, while the latter one sets them to zero. The right to reset dynamic PCRs depends on the locality (i.e., privilege level, with 4 being the highest and 0 the lowest) at which the reset command is issued. Table 2.1 lists the defined localities, their usage, and which PCR reset and extension capabilities each locality possesses. Table 2.2 summarises the specified type and usage for all PCRs. The usage, localities, and reset/extension capabilities are tuned such that they fulfil the requirement that each locality has only the least required privileges for reset/extension.

The PCR17 has a special role. It can only can be dynamically reset by a special LPC bus cycle, which is only given when the CPU issues a late-launch

PCR Type	**PCR Index**	**PCR Usage**
Static	0	CRTM, BIOS, and Host Platform Extensions
	1	Host Platform Configuration
	2	Option ROM Code
	3	Option ROM Configuration and Data
	4	IPL Code (usually the MBR)
	5	IPL Code Configuration and Data (for use by the IPL Code)
	6	State Transition and Wake Events
	7	Host Platform Manufacturer Control
	8-15	Defined for use by the Static Operating System. Host Platform
Dynamic	16	Debugging
	17	DRTM
	18	Auxiliary components
	19	Trusted Operating System
	20-22	Used by the Trusted Operating System
	23	Not used

Table 2.2: *PCR usage (Table adapted from [114, p. 33, p. 41])*

command. This bus cycle can not be emulated by software. Consequently, although software can still extend and read the PCR17, it can not reset it, only the CPU is capable of doing so.

The PCR extension functionality can be used in different ways to record the state of the platform. For example, one of the goals of the TCG is to establish trust into the booted operating system. The TrustedGRUB [9] implements this authenticated boot based on a *Static Root of Trust for Measurement* (SRTM), anchored at an immutable part of the BIOS, called the Core Root of Trust for Measurement (CRTM), and recursively applied measurements of the next piece of software to be executed during bootstrap. Thus, after the boot process completes, the PCR values represent the recorded boot sequence and can be used to verify that the loaded OS is trusted at this point in time. For instance, PCRs 8 and 9 contain the measurement of the bootloader stage 2 and PCR14 contains the measurement of the files that were actually loaded (e.g., kernel, initrd, modules, etc.). It has to be noted that, in contrast to a *secure* boot, the goal is not to abort the boot process when a part of the boot could not be verified, but to measure the boot process and to be able to attest this measurement to a remote party, which at a later time evaluates based on the measurements the trustworthiness of the booted system.

In order to meet the trust requirements, the authenticated boot measurement chain must meet the following three conditions:

1. The CRTM must be the first code running on the system and it can not be modified
2. The PCR values can no be reset, except at boot time before the CRTM executes
3. The measurement chain is contiguous, all software executing was measured before its execution, and the measured code is not modified between measurement and execution

Another example for recording the platform state is the *Integrity Measurement Architecture* (IMA) by IBM Research [95], which extends the trust measurements of an authenticated boot by also including the dynamic executable content in the application layer. In a system that implements IMA, all executable content loaded, e.g., executables, scripts, or dynamically loaded libraries, is measured and the measurement is stored in an ordered event log inside the kernel. The event log's integrity can be verified based on the TPM, because the IMA also extends a certain PCR – usually PCR10 – with the measurement when storing it in the log. This way, the exact platform state can be reported by means of the event log and the attested IMA PCR. A verifying party can calculate the alleged value of the IMA PCR from the event log, and if the computed value and the attested PCR value correspond to each other, the verifying party validated the integrity of the log and knows the exact platform state based on it.

2.1.2 Platform State Attestation

The TPM is able to attest – or quote – the platform status, based on the PCRs, to an external entity. Such an attestation is in its essence just a digital signature of the composite of one or several selected PCR values, with some additional information about the selected registers and information to guarantee the freshness of the attestation.

The key used for the attestation is called *Attestation Identity Key* (AIK). It is a TPM-generated 2048-bit RSA key, whose private portion never leaves the TPM in unencrypted form. A Privacy Certificate Authority (CA) is used to certify that the AIK was generated by and belongs to a legitimate TPM. Thus, another external verifier can validate the AIK's certificate, based on its trust in the issuing CA, before verifying any attestation of a TPM. Further, for verified attestations, the verifier can also decide based on the reported platform state if and how much trust he puts in the platform corresponding to the reporting TPM. One use case for remote attestation, that was explained earlier, is IMA. In this case, the TPM quotes the IMA PCR and thus enables an external party to retrieve and verify a detailed description of the platform state, i.e., the loaded software, and thus to evaluate its trust into the reported system.

It is important to notice that an AIK can only be used to quote, i.e., sign, TPM internal data like PCR values, but never externally supplied data. In order to sign external data with a TPM, a different key type than an attestation key is required. This distinction is essential to guarantee that data signed by an AIK originated indeed from inside the TPM of the AIK, e.g., the attested platform state is indeed the state recorded by the TPM and not a counterfeited one, simply signed with the AIK.

2.1.3 Sealed Storage

Another important property of TPM is the capability to provide sealed storage. A TPM contains a special key, *Storage Root Key* (SRK), whose private portion, similarly to the AIK, never leaves the TPM in plaintext. By specification the SRK is also a 2048-bit RSA key. The purpose of storage keys is to seal other data or sensible information like an AES key.

The sealing operation is essentially an encryption function, but it attaches further integrity-protected information to the encrypted data.

The foremost feature is that the unsealing of the blob can be bound to a specific platform configuration, meaning specific values of selected PCRs. With this feature, the sealed data will only be revealed in a given platform state.

Moreover, the sealed data contains also a) information about the platform configuration at seal creation time; b) a TPM specific value to guarantee that only the TPM who sealed the data is able to unseal it; and c) authentication data, such as a password, required to unseal the data.

The TPM does not inherently support chaining mechanisms for asymmetric encryption functions, thus prohibiting the sealing of larger amounts of data. So it is common practise to first encrypt the data that shall be sealed with a symmetric algorithm and then to seal the symmetric key with the TPM. Consequently, only if the secret symmetric key can be unsealed, the encrypted data can be decrypted, and thus the binding of the encrypted data to a certain platform state still holds.

Sealing can be used to create a key chain (or rather a tree) with the SRK at the root and with leaves that are either storage or signing keys. All intermediate keys, which act as parents for other keys, must be storage keys.

New keys are only created inside the TPM and their private portions are encrypted with their respective parent public key. Thus, the secret portion of any key generated by a TPM is never available in cleartext outside the TPM. Due to this fact, newly created keys are called wrapped keys.

Like the unsealing of sealed data, the usage of wrapped keys requires authentication and can be optionally bound to a certain platform configuration.

2.2 Mobile Trusted Module

A Mobile Trusted Module (MTM) is basically an TPM version 1.2 [117], that differs in certain aspects from the TPM design in order to be more suitable for mobile device specific uses cases [36, 112].

For example, the differences include the concept of *secure boot*. In contrast to an authenticated boot, which only measure the boot process, secure boot aborts the boot sequence in case a step in the process could not be verified as trusted. The verification in each step is accomplished by means of *Reference Integrity Metric* (RIM) certificates, which include the reference measurement for a particular step, and *verified PCRs*, into which the measurements are extended. A verified PCR can only be extended in the context of RIM certificate verification – not with the standard PCR extension command.

A further important difference is the introduction of two interleaving profiles for an MTM. Depending on the owner entity of the MTM, one distinguishes between Mobile Local Owner Trusted Module (MLTM) and Mobile Remote Owner Trusted Module (MRTM). In the former profile the owner, such as a mobile phone user, has physical access to the mobile device and in the latter profile the owner has no physical access to the device, but has an interest in ensuring the integrity and confidentiality of its data on the device. Examples for remote owners are the device manufacturer or a network service provider. In order to support these parallel stakeholders the reference architecture presented by the TCG [113] provides the possibility of deploying several MTMs, each with a different owner, on one single device.

One key feature of the MTM specification, which enables the support for parallel MTMs and which also provides the opportunity to deploy MTM on contemporary hardware, is the possibility to implement the MTM as software rather than as discrete hardware. The security properties regarding the necessary isolation and integrity 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-based MTM.

The Roots of Trust (RTs) have to be initialised during the initial boot steps, in a state where no software is executing prior to any MTM. The specification defines four RTs – namely the *Root of Trust for Enforcement* (RTE), the *Root of Trust for Storage* (RTS), the *Root of Trust for Verification* (RTV), and the *Root of Trust for Reporting* (RTR). The RTE states that the integrity and authenticity of the MTM execution environment has to be guaranteed with platform-specific mechanisms. The RTS is established by means of some form of device secret. Some immutable or similarly protected code constitutes the RTV, which is an engine that makes the initial measurements to be added to the MTM before the module is fully functional. The RTR holds the secrets required for attestation purposes. It has to be noted, that the RTR and RTV can be contained in the RTS, if the RTS has suitable statefulness guarantees.

Exemplary implementations of a software-based MTM are based on SELinux and a kernel-level isolation domain [6, 126]; virtualisation architectures [121, 50, 99]; or on processor security extensions like ARM TrustZone [18], Texas Instruments M-Shield [104, 107], or Sun JavaCard [106], as demonstrated in [35] and [30].

2.3 Dynamic Root of Trust for Measurements

A further important change introduced in the TPM v1.2 specifications is the concept of a *Dynamic Root of Trust for Measurement* (DRTM), also denoted as *"late-launch"*. This concept is an alternative to the Static Root of Trust for Measurement (SRTM). The TCG identified the following considerable drawbacks of the SRTM approach [26]:

1. The SRTM solution does not scale, because it relies on a possibly large chain of trust, of which a verifier has to know all components in order to evaluate its trust into the system.
2. Moreover, the time of measurement is at boot time when the system starts securely into a trusted state. But between this point in time and the point of the trust verification other applications may compromise the system's trustworthiness. Thus, one requires a measurement at runtime and not load-time.

3. Furthermore, the SRTM approach requires all executable content within the Trusted Computing Base (TCB) to be measured, but many systems have an ill-defined TCB and therefore force the SRTM approach to measure the executable content on the whole system.

Additionally, Kauer has shown in [59] that SRTM is prone to several attacks, which compromise the trustworthiness and security of the system, and that there currently exists no secure implementation of a static authenticated boot.

In contrast to SRTM, DRTM does not operate at boot time, but allows that the Root of Trust for Measurement (RTM) can be initialised at any point in time. This process can also be executed repeatedly. A key feature to this capability is the introduction of 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 instruction in CPUs available for commodity systems. Intel included it as part of their *Trusted eXecution Technology* (TXT) [54], formerly known as LaGrande Technology, and named the instruction `SENTER`. AMD integrated DRTM as underlying trust mechanism in their *Secure Virtual Machine* (SVM) technology [12, 13] and called the instruction `SKINIT`. Although the DRTM mechanisms of both vendors are similar, their implementations differ when it comes to the implementation details. The practical work in this thesis is based solely on the AMD platform, but in principle the results apply also to Intel platforms.

The `SKINIT` instruction takes as its sole argument the physical memory address where a *Secure Loader Block* (SLB) is located. That SLB contains a program image called the *Secure Loader* (SL), which includes all necessary the code and initialised data structures for its execution in the secure environment. The instruction re-initialises the CPU such that it establishes a secure execution environment for the SL code. The execution of the SL is hardware-protected in a way that it is safe from software-based attacks and even ordinary hardware attacks [13, Section 15.26].

Before the CPU hands over control to the SL, it enables the Device Exclusion Vector (DEV) protection for the physical memory region of the SLB in order to prohibit any Direct Memory Access (DMA) to this region through attached devices. Further, it clears the Global Interrupt Flag (GIF) in order to prevent any code executing previous to the `SKINIT` instruction from regaining control. `SKINIT` also disables the hardware debug features, since this might subvert the secure execution of the SL. On a multi-processor system an additional inter-processor handshake is performed in order to halt all Application Processors (APs), because the SL has to execute on the Bootstrap Processor (BSP). All of these measures guarantee the atomic and hardware-isolated execution of the SL code. In essence, `SKINIT` puts the platform into a state that is very similar to the state directly after a reboot and thus provides most of the security benefits of a reboot.

The key step in the late-launch process is the measurement of the SLB by the TPM. The `SKINIT` instruction is the only instruction that creates the necessary LPC bus cycles to acquire locality 4 of the TPM and trigger a reset of the dynamic PCR17 to zero (See also Table 2.1 on page 7). Subsequently to this reset, `SKINIT` hands over a copy of the SLB to the TPM, which extends the reset PCR17 with a hash of the SLB. It is important to notice that only the CPU is able to fabricate this specific new value of PCR17, since software is not able to reset PCR17 to zero and a system reboot resets it to -1. Moreover, it is important to notice that the PCR17 is at this point unique[1] for each SLB identity and thus, if the SLB is trusted, the PCR17 value is trusted and it forms the anchor of a new, dynamically created chain of trust.

After the protection mechanisms are enabled and the TPM has measured the SLB, the CPU enters flat 32-bit protected mode and hands over the control to the SL by jumping to the entry point of the SL.

Figure 2.1 illustrates the layout of an SLB. The maximum size for a secure loader block is 64kB. The first word of the SLB has to be the offset from the SLB start address (saved in the EAX register of the CPU after the `SKINIT` instruction) to the entry point of the SL. The second word has to be the size of the SLB, which also defines the area that is measured by the TPM. The unoccupied memory region between the SL code/data and the end of the SLB (pointed to by the ESP register) is used for the SL program stack during the execution. The SLB header contains the necessary data structures like, for example, a Global Descriptor Table, to set up the execution environment

The rationale behind the late-launch concept is to bootstrap a secure OS or secure VMM in an isolated and trusted way, that can be attested at any point in time, without the overhead of an actual reboot and thus without the need for an SRTM.

The SL would thus prepare the environment and the system for the boot of the OS/VMM. In contrast to the SRTM, here the BIOS, the bootloader and all software that ran before the `SKINIT` instruction are excluded from

[1]of course under the assumption that the used hash function is cryptographically secure and thus has a sufficient pre-image and collision resistance

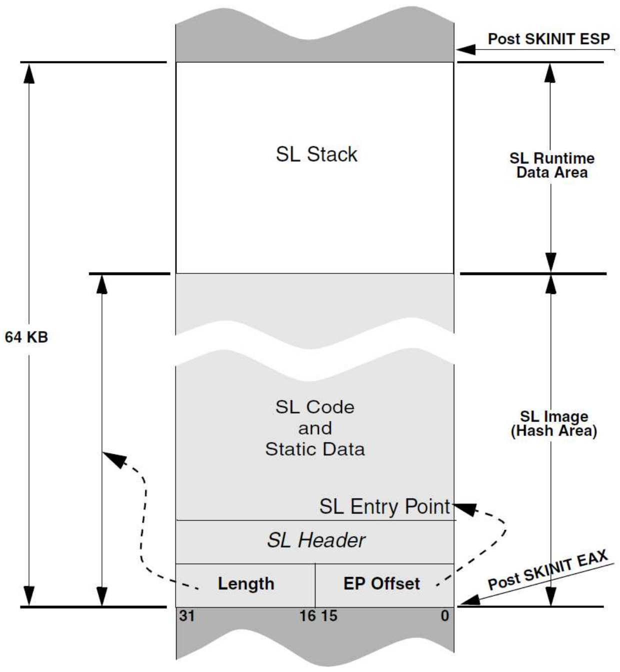

Figure 2.1: *SLB example layout (figure taken from [12, p. 56]*

the TCB, since `SKINIT` resets the system into a clean state, similar to the one of a reboot, from which the new secure OS/VMM is launched. Figure 2.2 illustrates the trust chains in both the SRTM and DRTM case. The highlighted path segment depicts the late-launch of the Secure Loader. The node *HW* denotes the involved hardware components like the CPU, TPM, RAM, and Northbridge[2], which are necessarily part of the TCB. Although the BIOS is not included anymore in the trust chain of DRTM, Kauer argues that the BIOS still has to be partially trusted, because one has to *"trust the BIOS for providing the System Management Mode (SMM) code as well as correct ACPI tables. As both can be security critical, a hash of both of them should be incorporated at boot time into a PCR by the operating system"* [59, p. 4]. For that reason, DRTM should be supplemented by SRTM up to the point where the BIOS is measured in order to provide some means to attest the absence of SMM based attacks like the one presented in [123].

Implementations of this late-launch concept are Kauer's *OSLO* [59] and Intel's *tboot* [53].

[2]the chipset responsible, among other things, for the communication between the CPU and the RAM

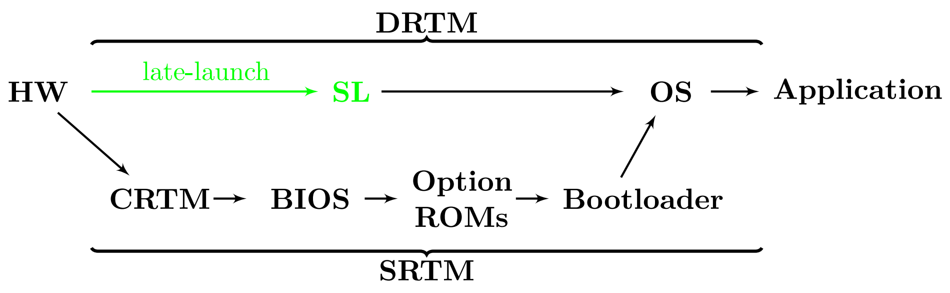

Figure 2.2: *Comparison of trust chains in SRTM and DRTM*

2.4 Flicker

The Flicker architecture by Jonathan McCune *et. al.* [73, 72] leverages DRTM to establish a secure execution environment that offers the above mentioned hardware-security features. But in contrast to the original TCG intentions to use DRTM to late-launch a secure OS/VMM, Flicker uses DRTM to execute a small, security-sensitive part of a program 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: Isolation of security-sensitive code from all other software and devices; protection that is provable to a remote party; meaningful attestation of exactly the code executed, its inputs, and outputs; and to have a minimal mandatory TCB for the security-sensitive code.

While the isolation and the provable protection goals are inherently fulfilled through DRTM, the TCG intentions for DRTM miss the other two goals. If DRTM is used to bootstrap a secure OS or VMM, the TCB for an application running on top of these expands to the secure OS/VMM. Bearing in mind that a mainstream hypervisor like Xen [32] comprises about 50,000 lines of code, it is obvious that the mandatory TCB is significantly increased and that attestations include unnecessary and uninformative facts about the platform state. Flicker takes here *"a bottom-up approach to the challenge of managing TCB size"* [73, p. 317].

In Flicker, the initial mandatory software TCB consists of about 250 lines of code, to which the programmer of a PAL adds only the necessary security-sensitive code of his program. Due to the fact that DRTM provides the hardware isolation from all other software and the protection from software-

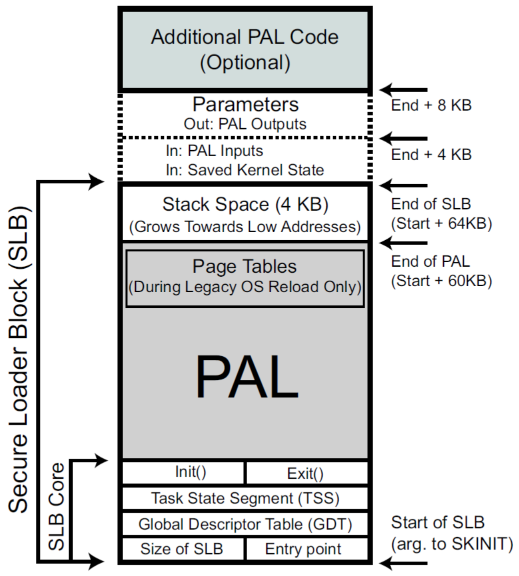
Additional PAL Code (Optional)
Parameters
Out: PAL Outputs
In: PAL Inputs
In: Saved Kernel State
Stack Space (4 KB)
(Grows Towards Low Addresses)
Page Tables
(During Legacy OS Reload Only)
PAL
Init()
Exit()
Task State Segment (TSS)
Global Descriptor Table (GDT)
Size of SLB
Entry point
End + 8 KB
End + 4 KB
End of SLB (Start + 64KB)
End of PAL (Start + 60KB)
Start of SLB (arg. to SKINIT)
Secure Loader Block (SLB)
SLB Core

Figure 2.3: *Memory layout of an SLB, containing a PAL, for Flicker (figure taken from [73, p. 318])*

based attacks, all software executing before an SL, like the BIOS or OS, are excluded from the TCB.

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 in order to assess the trustworthiness of the PAL.

Architecturally, Flicker suspends the current execution environment (most likely the OS), executes the `SKINIT` instruction to initiate the secure environment in which the PAL then executes, and afterwards resumes the previous execution environment. To provide a better abstraction of the rather low-level and complex setup of an SLB and the `SKINIT` execution, the Flicker architecture includes a kernel module that provides virtual file system entries for the I/O communication between the user/application and the PAL as well as a core library against which PALs are linked. The library 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.

Figure 2.3 illustrates the memory layout of an SLB for Flicker that contains a PAL. Corresponding to the general layout of a secure loader block in Figure 2.1 (p. 15), the first two words of the SLB contain the offset to the entry point (of the SLB Core initialisation function) and the size of the SLB – here the SLB Core size plus the size of the PAL code.

The first bytes of the SL code segment contain data required to properly set up the execution environment. These include an uninitialised Global Descriptor Table (GDT) and a Task State Segment. The GDT is a skeleton defined at the time the SLB Core is linked with the PAL code. The next bytes of the SL code are occupied by an initialisation function, executed after handover of the control from CPU to SL (i.e., it is the actual entry point of the SL code section), an exit function that cleans up before the SLB returns to the legacy OS, and the SL part of the resumption code.

The GDT and Task State Segment are required during the execution of the SL code, because the initialisation function enables the processor's segmentation support, whereby the segments start at the base of the PAL code. Since this value is unknown at build time of the SLB, the GDT and Task State Segment are simply skeletons, which will be fixed up by the Flicker kernel module before the `SKINIT` execution.

Because the `SKINIT` instruction disables the paging mechanism for virtual memory, the exit function has to re-enable this feature in order to cleanly resume the operation of the legacy OS. The higher 12kB of the PAL memory region are therefore used by the exit function as space to restore the page tables, required to re-activate the paging mechanism. The PAL code in this memory region is thereby overwritten with the page tables. However, the exit function executes last in the PAL and is located in the lower region of the SLB, thus no critical code at this point in time is overwritten during the restoration.

The free memory region between the end of the PAL code and the end of the 64kB SLB area are used as stack space during the PAL runtime. In the default setting SLB Core reserves a 4kB stack.

The Flicker kernel module allocates 128kB of kernel memory space, of which the lower 64kB are used for the SLB, while the higher 64kB are used to store the input parameters from the application to the PAL and the output parameters from the PAL to the application. In its current design Flicker supports 4kB input and output, but provides the possibility to increase these sizes. The input and output values are written/read via the file system entries of the kernel module. Moreover, the kernel module uses the first bytes of the input memory region to store the CPU state just before the `SKINIT` instruction. This state is then restored from this well-known address on return from the SLB to the legacy OS.

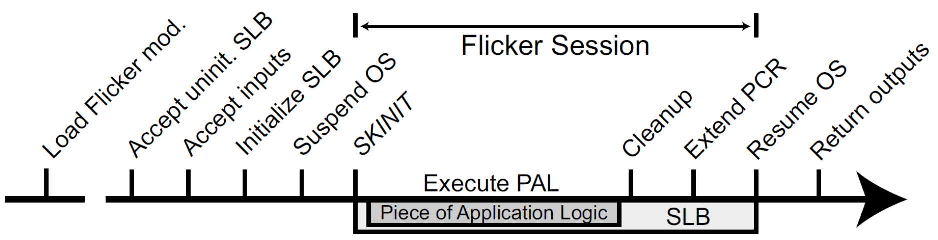

Figure 2.4: *Time-line showing the execution of a PAL with Flicker (figure taken from [73, p. 318])*

The remaining 56kB of the allocated kernel memory may be used for optional PAL code. But it should be noted, that the higher 64kB of the allocated space are per default not protected by `SKINIT`, and it would require additional steps in the initialisation function to set up this protection by means of the DEV. In that case, this new protected memory region should also be measured and the measurement extended into PCR17 in order to guarantee a sound attestation.

Figure 2.4 shows a detailed time-line of the particular steps of a PAL execution with Flicker.

Before the execution of any PAL, the Flicker kernel module has to be loaded. As already mentioned, the kernel module provides file system entries that simplify the I/O communication with the PAL, e.g., an entry to which applications write the SLB that contains their PAL, entries for the input and output parameters of the PAL, and a control entry to initiate the `SKINIT` execution. Moreover, the module is necessary, because `SKINIT` is a privileged instruction that requires kernel level permissions to execute and, furthermore, the memory region for the SLB has to be allocated in the kernel's address space. The physical address of the latter one is denoted as `slb_base`.

An application, that wishes to execute a PAL, first writes the SLB memory image and optional input parameters to the corresponding file system entries of the Flicker kernel module. It then triggers the execution through the control file system entry. The kernel module then fixes the skeleton GDT of the SLB image, takes the necessary steps to suspend the legacy OS (e.g., halting all application processors), saves the CPU state of the bootstrap processor, and executes the `SKINIT` instruction, which starts the Flicker session.

A Flicker session consists of the execution of the PAL (preceded by the very short initialisation function), a subsequent cleanup when the PAL has finished, a PCR17 extension, and then the resumption of the legacy OS.

The initialisation function simply enables the processor's segmentation feature based on the initialised GDT, loads the segment selectors, and then jumps to a well-known function in the PAL code. With this jump the PAL code starts executing and on return it jumps to the exit function. The initialisation function also makes sure that the PAL executes on CPU privilege level 3 (ring 3) and requests the locality 3 from the TPM for usage by the PAL.

The exit function is responsible for the return to the higher CPU privilege level 0 (ring 0) via a call gate[3] in the GDT, for cleaning up, for extending PCR17, and for the resumption of the legacy OS. Cleanup means the erasure of sensitive data that is stored in memory. This is necessary, because the protected memory becomes accessible again to the untrusted legacy execution environment after the return.

Further, PCR17 is extended with a well-known value. This enables a remote verifier to distinguish between measurements taken during the trusted and secure execution of the PAL and measurements taken afterwards by the untrusted legacy execution environment. Moreover, it prevents software in the legacy environment from unsealing data that has been sealed to a specific PAL via its PCR17 value.

The SL resumption code then restores the page tables, reloads a GDT that covers all of the memory, as required by the legacy OS, and enables paging plus virtual memory based on the restored page tables. Control is then handed over to the legacy OS – to be precise to the Flicker kernel module – which restores the CPU state, activates the halted application processors, and re-enables global interrupts.

After this reactivation of the legacy OS, the application can receive the PAL outputs via the corresponding file system entry of the kernel module.

The application is, furthermore, able to attest the execution of the PAL and the integrity of the result. If the PAL extends PCR17 with the hashes of the input and output parameters before the register is extended with the well-known value that marks the end of the Flicker session, then PCR17 reflects exactly the specific combination of the PAL, its parameters, and its result. Based on the TPM remote attestation of PCR17 and the mechanism how the PCR17 value can only be fabricated by a late-launch, the application can convince a remote verifier that a particular PAL has been executed and that it has indeed produced the particular output from the particular input.

[3]a mechanism that allows less privileged code to call code with a higher privilege level via a CALL FAR to a function referenced in the GDT

Although the execution of a PAL seems quite complex and the suspension of the operating system during the execution of the PAL – during which all external interrupts are ignored – looks rather interruptive, performance and data integrity tests by McCune *et al.* [73] show that the impact on the operation of the legacy OS, even under higher load, is negligible. File consistency is preserved, even during larger bulk data transfers and during file downloads that interleave with Flicker sessions and thus are interrupted. The test results have been empirically verified in the context of this thesis.

2.5 Problem Statement

The goal of this thesis is to design an architecture that uses DRTM to implement a secure credential execution and storage for applications. The foundation of this architecture is the Flicker infrastructure, which fulfils the necessary presupposition, namely a hardware-based isolation of the measured, secure execution environment for the credential platform from the legacy environment, and beneficially also a small Trusted Computing Base for the software-based MTM.

As described in Section 2.3, the launch of a Secure Loader (SL) puts the platform into a state similar to the one after a reboot and thus isolates the SL from the legacy environment such that it executes securely. The Flicker infrastructure, as explained in Section 2.4, adds to this concept the functionality of easily resuming the legacy execution environment. That way, it offers the possibility of executing security sensitive parts of applications under the hardware-based protection of DRTM.

The way how DRTM executes in the context of Flicker constitutes a security risk, if the secure execution environment is used for an application-specific credential platform, because due to DRTM the SLB and thus the PAL are stateless and not bound to the corresponding application in the legacy environment.

In the Flicker infrastructure, PALs may receive input from the legacy environment and thus this input defines the state of the PAL. But this approach also requires an authentication of the input in order to verify the data's origin. Bearing in mind, that an SLB is just a byte blob consisting of code and pre-initialised data, it is not feasible to deploy any secrets, like keys or passwords, used for authentication, securely within the SLB.

Another approach to achieve applications' authentication could be to store the state of the PAL inside the SLB data section and to rely on remote attestation of the PAL execution in order to verify the state. But for several reasons this approach fails. For instance, valid states could be replayed by just overwriting the SLB image and further, since the state is likely to be mutable, a remote verifier had to keep track of all possible valid states.

The statelessness of PALs also entails security issues like the misuse of deployed credentials by other software on the system. For example, an SLB containing a PAL that establishes for an online banking application an attested, secure channel to a remote server, might be launched by malware on the system. Since there is no link between the legacy environment and the PAL, the latter one will successfully establish this channel on behalf of the malware.

The designers of the Flicker infrastructure have identified the statelessness problem in the context of multiple Flicker session scenarios. They propose a sealed storage solution, based on the TPM sealing capabilities, to maintain the state of the PAL. But this solution is PAL specific and not application-specific. It ensures that only the legitimate PAL is able to unseal the state, but it does not solve the problem that the sealed state, and thus the PAL, are not bound to an application.

In the same context, the Flicker designers identified the necessity for replay prevention mechanisms of the sealed state and sketched a solution based on counters, which are deployed in the TPM non-volatile memory. Unfortunately, the available non-volatile memory space is very scarce and considering the high number of applications on a normal platform which would use a PAL, it is extremely likely that the NVM space is insufficient to provide replay-protection for all states. Alternatively, the replay prevention could be based on TPM monotonic counters. However, the TPM specifications define the minimum number of monotonic counters per TPM to only four and most modules on the market do not provide more than this minimum. Hence, the possible number of PAL states, that are protected against replay-attacks, is considerably constrained. Moreover, only one TPM monotonic counter is allowed to be incremented – possibly multiple times – between system reboots, hence prohibiting any interleaving PAL executions that require different monotonic counters to be updated. Thus, a more feasible approach to the replay prevention has to be provided by our architecture.

A software-based Mobile Trusted Module (MTM) was selected as the application-specific credential platform in our architecture. The above stated issues imply that the MTM is by default stateless and not bound in an application-specific manner, since the link between the legacy environment and the MTM PAL is missing. The architecture therefore has to solve three fundamental problems in order to achieve secure application-specific MTMs:

1. a mechanism to unambiguously identify the application that initiates a Flicker session and thus its specific MTM
2. a mechanism that, based on the prior application identification, binds the Flicker session or the MTM to the identified application in an one-to-one manner
3. a mechanism that isolates a bound MTM PAL state from PALs other than the legitimate one, executing on behalf of the legitimate application corresponding to this MTM

These points form the fundamental requirements to enable the implementation of application-specific MTMs based on the Flicker infrastructure. Additional security requirements include the replay prevention of old MTM states or means, based on the platform TPM, to attest the computational results of the MTM.

3 Related Work

As described in the preceding chapter, the Flicker infrastructure provides applications with the possibility to execute sensitive computations in an isolated environment that is resistant against software-based and simple hardware-based attacks. The kind of code executed in such a Piece of Application Logic (PAL) is generic and only limited by the constraints imposed on it by the DRTM mechanism, for example in terms of code size. Based on the TPM sealing and attestation capabilities, the infrastructure further offers secure storage of data and the authenticated, secure reporting of the executed PAL. Moreover, DRTM noticeably reduces the required Trusted Computing Base for the security of the PALs, because it excludes the bulk of the BIOS, the entire bootloader, and the legacy execution environment from the TCB. The declared goal of this thesis is the design and evaluation of an architecture, built on top of Flicker, that implements an application-specific credential storage and usage. The credential platform chosen for our architecture is the Mobile Trusted Module, which provides the application's developer with standardised interfaces (TCG Software Stack) that already have been implemented in popular libraries like TrouSerS [8].

In this chapter different approaches to the secure storage and execution of credentials are presented. Their respective precedences and drawbacks, with respect to the above mentioned architecture attributes, are briefly evaluated.

3.1 Credential Managers

Password Managers and Key Stores

Probably the most widespread credentials are passwords and user keys (e.g., for PKIs), and thus by consequence the probably most widespread form of credential platforms are password managers and key stores. Both are today by default included in almost all operating systems and are accessible through system software or cryptographic service providers. Also, almost all Internet applications like web-browsers, email clients, etc., as well as stand-alone software applications provide them independently from the OS. Normally, these types of software use cryptographic means to guarantee the confidentiality and integrity of the password/key database stored on persistent storage. Additionally, access to this database may be authenticated by a master password that the user has to remember. In general, all these approaches are subject to software-based attacks like malware (trojan horses, key-loggers, etc.)[103], because the security sensitive computations (like the decryption of the database) or user I/O are not performed in an isolated execution environment. Thus, the Trusted Computing Base of this software includes the operating system and other software running on the platform. Moreover, the credentials are not stored in an application-specific manner, because every application with knowledge of the master password or access to the database encryption key is capable of accessing the credentials. Furthermore, password managers and key stores generally do not provide standardised interfaces that enable other entities to use or retrieve the credentials.

Single Sign-On

Newer, more generic solutions for identity management of the user are gaining popularity. In this context, the user delegates the management of his identity and credentials to a Trusted Third Party (TTP) and is only once required to authenticate himself to the TTP (Single Sign-On). For example, the popular OpenID protocol [5] implements a decentralised authentication system for web-based services and websites, whereby the user delegates his authentication to an OpenID provider. The protocol is open and standardised, thus guaranteeing a variety of providers and decentralisation. The user authentication at websites or services, succeeding the user login at the provider, is URL based. The OpenId protocol centres around the user identity and thus credentials are not stored or used in an application-specific manner, since each application that implements the protocol has access to the same user credentials. In terms of isolation and secure storage, OpenID exhibits the same vulnerabilities as usual web logins, including phishing, key-logging, or session hijacking. Other identity management solutions are the SAML (Security Assertion Markup Language) based Liberty Alliance [3] and Shibboleth [7] projects. Microsoft offers with CardSpace [10] a comparable, but centralised, solution as part of their .NET framework. In CardSpace, the user manages different identities (i.e., InformationCards), which are kept inside his "wallet" (i.e., CardSpace or Identity Selector). Since the wallet is an application running on the user machine, it is prone to software-based attacks at runtime and to the same issues as encrypted databases for password or key storage.

Central Credential Repositories

A solution proposed by the Grid community to deal with user credentials in the context of PKI infrastructures, namely the user's key pair, are central credential repositories like the MyProxy [82] system. In this system the user's long-term credential, i.e., his key pair, is stored on the proxy which acts as the repository. After the user has been authenticated at the proxy and has authorised the use of his credentials, the proxy derives a short-term credential from the long-term credential and uses the former for the user authentication inside the grid.

In recent research this system has been strengthened in terms of the security of the long-term credential. In [67] the authors use hardware-security (IBM's 4758 secure co-processor [33]) to increase the security of the private key storage and generation. In [70] this approach is extended by further security features. Short-term credentials are accompanied by certificates that attest to the conditions under which the credentials were created, hence other entities can evaluate the level of trust that they put into the short-term credentials. Moreover, a policy framework was introduced, which allows the user to define the usage of his long-term credentials.

In both cases the authors' motivation is to improve the security of the private keys and avoid the TCB on the user-side. The application of secure hardware provides highly secure storage and execution of the credentials, thus avoiding the insecure storage on the user-side as well as the complex, ill-defined TCB on commodity desktop systems of users, which compromises the security of the credentials at runtime. However, the user has to authenticate himself to the proxy, most likely via his web browser, and thus malware may still be able to impersonate the user and abuse his credentials. The repository system is user-centric – providing the user with mobility – but it does neither store or execute the credentials in an application-specific manner nor provide standardised interfaces.

On-board Credentials

A new approach to the credential storage and use, especially for (but not limited to) mobile devices, are On-board Credentials (ObC) by Nokia Research Center, Finland [64]. ObC combines the high security of general-purpose secure hardware, like TI M-Shield [104, 52], with the flexibility of virtual credentials, like passwords. The secure hardware provides hereby an isolated secure execution environment, secure storage, and means to ensure the integrity of the secure environment. This approach is *open* in the sense that 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 very suitable option as credential platform for this thesis. But on the other hand, it is complex and requires, in comparison to the MTM code base [35] available to us, a rather complicated set-up. Moreover, ObC does not provide standardised interfaces for applications in contrast to MTM. Thus, although ObC would be more suitable in terms of functionality, MTM is chosen as credential platform for this thesis for reasons of feasibility and compliance to standards.

3.2 Security based on Isolation and TCB Minimization

TruWallet

A wallet solution without the delegation of the identity management, that offers strong protection of credentials and binding to the user's platform configuration, is presented in [41]. This solution, named "TruWallet", is based on virtualisation via the Turaya security microkernel [93], which provides a secure GUI for trusted paths, support for Trusted Computing based on a TPM, secure execution environments for applications, and is interoperable with existing Operating Systems and applications.

In the TruWallet architecture, the wallet application executes inside its secure environment (*compartment*) and acts as a web proxy that performs the user's login at websites. Hereby, TruWallet implements protocols that establish a shared secret between the user's wallet and a server and additionally offer migration of the credentials based on TPM functionality.

The security of the execution and of the stored credentials is guaranteed through the security kernel, its TC capabilities for sealing, and the compartmentalisation of the applications. Only the untampered kernel and wallet application, measured by an authenticated boot, are able to unseal the credentials. Thus, the security kernel has to be fully trusted at runtime and forms the bulk of the Trusted Computing Base (TCB) in the TruWallet architecture.

The isolation of the credential execution environment is provided by the kernel and hence depends on the guaranteed integrity of the kernel at runtime. Advanced software attacks against the kernel might break this guarantee and subvert the kernel in order to reveal secrets. The TruWallet authors argue that *"[the] TCB cannot be compromised during runtime because of the assumption that it is small enough to be verified and tested thoroughly"* [41, p. 25].

Although the security kernel is substantially smaller than commodity Operating Systems and thus in the size range for formal verification, this assumption is at this time only conditionally valid. The currently most famous example for a formally verified microkernel is seL4, which has approximately 7500 LoC [60, 2]. The verification implies the fulfilment of critical security properties like the absence of code injection attacks, buffer overflows, unchecked user-arguments, or NULL pointer access, but the actual goal of the verification was to prove functional correctness and not the security. Thus, certain security-relevant properties, such as covert channels, were not included in the proof.

The verification was achieved in an interactive manner based on the theorem prover Isabelle/HOL [85]. In total, the proof comprises about 200,000 LoC. But the authors of [60] state that they assumed certain design simplifications of the kernel. For instance, the proof was done for an ARMv6-based platform without Symmetric Multiprocessing (SMP). The SMP version or the x86 port of seL4 has yet to be verified.

Another current issue is that the TCB is not fully covered by the proof, because the memory management, which is part of the TCB, is pushed off the kernel into user-space and has to be verified separately. Further potential issues for an x86 port of the proof are pointed out in the security discussion of [56, Section 3].

Hypervisors and Microkernels

Although a formally verified microkernel is not yet available for the x86 platform, which is targeted in this thesis, microkernels and hypervisors offer good security benefits through a decreased perimeter of the TCB compared to the commodity OSes and through a high level of isolation for applications that deal with sensitive information like credentials. This principle has, for instance, been shown by TruWallet. The designs of today's commodity operating systems are not based on microkernels or hypervisors, but are monolithic, meaning almost all of the functionality, including the drivers, executes in kernel privileged mode. Applications operating on top of such kernels have to include the bulk of the OS into their TCB. Considering the growing complexity and size of monolithic kernels, it is clear that the number of security issues rises as well, thus advocating new approaches to secure OSes and architectures [68].

The microkernel/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. For example, the Terra architecture [42] implements a Trusted VMM that partitions the platform into ordinary VMs and "closed box" VMs that provide the functionality of a dedicated, Trusted VMM protected closed platform to which software can tailor their software stacks based on their security demands.

The already mentioned Turaya security kernel of the EMSCB project [93] and the Qubes architecture [56], based on the Xen VMM, both compartmentalize applications and provide a trusted GUI to the compartments.

Both the Xen based approach described in [125] and the Overshadow architecture [27] protect the application data privacy through an en-/decryption mechanism of the applications' memory pages.

The Nizza architecture [102] extracts the security sensitive parts (AppCores) at the system software level and at the application software level and runs them as isolated trusted processes on top of the TCB, while the untrusted parts run on top of an untrusted legacy OS.

In [109] the authors describe their Xen based Proxos architecture, which allows applications that run in private VMs to configure their trust into the legacy OS by partitioning the system calls into trusted and untrusted ones. Proxos then routes the untrusted system calls to the legacy OS and trusted system calls to a private OS inside the private VM.

The NOVA architecture [105] aims specifically at minimising the TCB through decomposing the virtualisation architecture and implementing virtualisation in user-space (microhypervisor approach).

Also Microsoft's Next-Generation Secure Computing Base (NGSCB) [38] concept distinguishes between a trusted and an untrusted partition for software. The NGSCB design hereby envisions and exploits an improved platform hardware support.

In general, the major chip vendors have improved their hardware support for virtualisation and platform security. This includes Intel's VT/TXT [54] or AMD's AMD-V/SVM [12], which are adopted by projects like Xen or VMWare. In their Safer Computing Initiative, Intel describes approaches to secure computing leveraging the improved hardware support [44, 94]. Research projects that make specific use of the improved hardware features are presented by the CMU [100, 71]. The authors of [100] describe an architecture that implements a tiny hypervisor (about 1100 LoC), which uses the CPU supported memory virtualisation capabilities in order to isolate the memory region of a commodity OS kernel from the rest of the platform software, thus protecting the integrity of the kernel at all times from software based attacks. The TrustVisor architecture [71] 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 execute then in total isolation from the rest of the system's software and from 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 μTPM. The main motivation for the TrustVisor design is to avoid the performance penalty of Flicker, which has to be paid in order to achieve its extremely minimal TCB.

Microkernel-based architectures, e.g., the L4 kernel [4], split both the OS and the applications up into multiple components, which run in their own process. The microkernel is responsible for isolating the processes from each other.

Common to all the above mentioned approaches based on a hypervisor or a microkernel is that they are suitable for providing a high level of isolation for the execution environment of an application and to provide further resistance against software based attacks. Furthermore, it has already been shown that such compartmentalised applications are suitable to implement standardised credential platforms like the MTM [121, 50, 99] and due to projects like IBM Research's vTPM [20] VM's have access to the physical platform TPM.

Compartmentalised applications are also easy to measure, enabling advanced security features like TPM based sealed storage or remote attestation, as shown, e.g., by the TruWallet architecture.

The essential prerequisite for all these guarantees is the integrity of a trusted hypervisor/microkernel at runtime. While SRTM or DRTM can ensure the trustworthiness at boot-time, it is far more complex to guarantee integrity at runtime – Time of Check to Time of Use (ToCToU) problem – and this issue is investigated in contemporary research [69, 101, 19, 28]. Thus, the hypervisor or microkernel has to be part of the TCB and is a potential security risk. Moreover, in contrast to the Flicker architecture, compromised software at the highest system privilege level (i.e., the kernel or the VMM) can jeopardise the secrecy of applications' credentials.

A good overview of the benefits and risks of hypervisor/microkernel based architectures and examples for successful attacks are given in [84, 122, 62, 14, 87, 47, 88, 15, 16].

3.3 Platform Security

Mandatory Access Control

Access control to credentials stored on persistent storage, or in case of the Flicker based architecture to the Secure Loader Blocks (SLBs), can be achieved through Mandatory Access Control (MAC) systems. On Windows, MAC is part of the Mandatory Integrity Control [75]. For Linux exist several MAC schemes, most of which are based on the Linux Security Modules (LSM) architecture [124]. The most famous scheme is the Security-Enhanced Linux (SELinux) [6], but other approaches exist, like AppArmor [81], TOMOYO Linux [83], or SMACK [98]. In [37] Enck *et. al.* present PinUP, an LSM based approach to pin files to applications in form of an access control overlay. A role-based access control scheme, not based on LSM, is part of grsecurity [1] for Linux kernels.

Regarding our stated problem, SLB images, and thus PALs, can be bound to applications via access control policies. However, common to all these approaches is that the access control scheme relies on file system properties like file attributes or the file path. These properties are prone to passive attacks, such as changing the file attributes. Moreover, only the SLB image is bound to the application, but not the PAL execution environment or the PAL state. Our architecture implements exactly this link, thus providing a stronger binding than mandatory access control. However, MAC can be used as a supplementary mechanism in our architecture.

System Integrity

One approach to ensure the system integrity is binary signing. While this approach is, for example, heavily used in the Symbian OS security architecture (see Section 3.4), it is not yet very common for desktop systems. The latest Windows OSes or the Ubuntu/Debian Linux package manager make use of signed binaries in order to verify the integrity and authenticity of downloaded installer executables, but it is not used at load-time of already installed software. Binary signing implementations for Linux are for instance IBM Research's solution [119] or the newer, open-source DigSig [17] infrastructure. In both cases, executables are only executed if their signature can be successfully verified. Similarly, libraries can only be loaded successfully if their signatures could be verified. In contrast to binary signing on Symbian OS, both these systems are built on top of an ill defined TCB, i.e., the bulk of the Linux kernel is part of the TCB. Considering the size of the Linux kernel (several million lines of code) it is obvious that this TCB is easily vulnerable to attacks.

An alternative approach that is based on TCG trusted computing features is the already outlined Integrity Measurement Architecture (IMA) [95] (Section 2.1.1 on page 9). In contrast to binary signing, IMA does not verify the integrity of the executables, but solely records their measurements at load-time. However, IMA is able to use the TPM in order to verify the integrity of and to remotely attest its measurements.

Both solutions, binary signing and IMA, are suitable mechanisms to retrieve the identity, i.e., hash, of an application in this thesis' architecture. Here, IMA was chosen for two important reasons. For one thing, IMA is already a default part of the latest Linux kernels, whereas DigSig requires a kernel patch. Secondly, the enforced integrity at load-time, which binary signing offers, is less important in our architecture than the possibility to securely attest the measurement with TCG TC means to a remote party.

3.4 Symbian OS Platform Security

The Symbian OS v9 introduced an enhanced security architecture that enables the protection of the interests of stakeholders on the mobile phone, such as the consumers, the network operators, or the software developers [46]. The key concepts of the security architecture are a small trusted computing base, application capabilities, and data caging. The security model is strongly oriented towards the principles defined by Saltzer and Schroeder [96], for example open design and least privilege. The architectural goals comprise the provisioning of a trust base and of a lightweight security model that supports openness.

Trust Computing Base

At the core of the Symbian OS is a microkernel that acts as reference monitor. The microkernel together with the computer hardware, the file server process, and the software installer form the minimal TCB in the Symbian OS.

The Symbian OS system model assumes that there is only one single user on the system. This simplifies the security model, which hence focuses only on the processes in the system. Processes form the unit of trust in the security model and they are classified in four different tiers of trust that correspond to the privilege level associated with the process. The TCB is at the highest privilege level and controls the access of system server processes inside the Trusted Execution Environment (TrEE) to low-level operations, like the screen hardware or the communications device driver. Code in the TrEE is trusted but it is not running on the highest privilege level. Add-on software that requires services from the TrEE or TCB has to be signed by a trusted party with the necessary privilege level to gain access to the TrEE or TCB. The software installer checks the requested privileges of the software and grants them based on the privilege level of the certifying party. It further ensures that the software can not affect the system integrity and protects sensible low-level operations. The certifying authority is also responsible for validating the code correctness and trustworthiness before signing. Although unsigned software can be installed, this software has only the least privilege level, since it is considered untrusted.

Additionally, Symbian OS v9 introduced identifiers for every software and software vendor. The SID number identifies uniquely a particular application and its version. The VID number on the other hand identifies a vendor. A specific range of SIDs and VIDs is reserved for applications that have been signed by a trusted authority, thus indicating the raised trust level of this applications. In general, the decision to grant access to specific system resources can also be based on this special SID and VID values and the trust the installer puts into the specific application (version) or vendor.

Furthermore, modern devices provide the technological means to protect the TCB from being modified or erased while the system is offline. An attacker or legitimate user could, for example, reflash the system and install modified firmware that breaks the security of the TCB. Newer devices therefore support the secure boot concept (as explained in 2.2) to protect the TCB. However, despite its protected TCB SymbianOS v9 is not free of possible attacks, especially on old CPU cores like ARMv5, as shown by, e.g., Collin Mulliner [76].

Capabilities

Symbian OS implements a capability-based security model that follows the concept introduced by Dennis and Van Horn [29]. The privilege level a process requires is determined by the capabilities that it carries along. A capability is defined as a token that needs to be presented in order to gain access to a system resource, which are accessible in Symbian OS via service APIs. Capabilities in Symbian are discrete and orthogonal. Although there are currently only 20 different capabilities defined, they can be classified on a broad scale in TCB, system, and user capabilities. As mentioned above, the software installer validates via a digital signature at install time that a program has the necessary rights to get to its requested capabilities. Software that is already installed on the platform, i.e., in ROM, has been checked manually. Following the principle of least privilege, a process can only load libraries that have at least the privileges as the process does.

Data Caging

The third key concept of the Symbian OS security architecture is data caging as the file access control model. Data caging is used to prevent unauthorised access to system-critical or to confidential files. The access control is path-based and does not implement explicit access control lists in order to save the scarce storage space on mobile devices, to extend the battery life, and to improve performance. To provide compatibility with existent software, the access restriction apply only to certain special directories and all their sub-directories, namely: `\sys`, `\resource`, and `\private`. Of special interest for applications is the `private` directory, within which every application has its own sub-directory based on its SID. The file server ensures that each application has access to its own sub-directory, while the access to other applications' sub-directories requires a certain system capability.

Unified stores

Finally, the Symbian OS platform security includes unified key- and certificate stores [108]. Applications are able to create their own store that provides them with the possibility to manage their own keys and certificates via an API.

The API for the key store enables the application to create, import, and export RSA, DSA, and DH key pairs (in PKCS#5 and PKCS#8 format); to protect keys by two-factor user authentication; and to perform common key operations like signing or de-/encryption.

Via the API of the certificate store, applications are able to manage certificates in one of the supported formats (X.509, WTLS, X.968); to construct and validate certificate chains; and to assign a trust status and applicability (i.e., purpose) to the certificates.

Based on the small TCB and the overall security concept to protect the microkernel in Symbian OS v9, the stores provide a good solution to store and use key and certificate credentials in an application-specific manner. However, although the stores provide standard algorithms and formats, the interface to them is proprietary to the Symbian OS and not standardised like, for example, the TSS for TPM/MTM. Furthermore, the stores provide protection against malicious software on the phone, but not against simple hardware attacks.

3.5 Hardware Tokens

A number of general-purpose secure hardware types have already been mentioned up to this point. For instance, the Trusted Platform Module (TPM) or processor security extensions like ARM TrustZone [18], Texas Instruments M-Shield [104, 107], Intel's TXT [54], and AMD's SVM [12]. This type of secure hardware provides fundamental capabilities, like secure storage or isolated execution environments, on which security architectures can be built on.

Further, there exists dedicated secure hardware, such as smart cards (e.g., SIM card [39] or JavaCard [106]) or RSA SecurID [90]. Usually these types of hardware offer high level of security, standardised interfaces like PKCS#11, and a two-factor authentication for granting access. On the other hand, they are designed for a dedicated purpose, e.g., SSO, and are often inflexible in their applicability for other credentials than the intended one. Furthermore, they target a human user and not the binding to a specific application.

4

Architecture

In this chapter the architecture is presented that implements the application-specific means to use and store credentials. The architecture leverages the Flicker infrastructure and makes use of software-based Mobile Trusted Module (MTM) as its credential platform, both presented in Chapter *2 Technical Background and Problem Statement*.

The basic idea behind the architecture is to provide all applications with the possibility to securely execute their own MTM inside a Flicker session, thus under the protection of DRTM, and with a minimal Trusted Computing Base (TCB). The MTM can then be used by applications to store and to use their credentials in a secure and isolated manner.

Additionally, the architecture supports the notion of Trust Domains (TDs). Each Trust Domain is centred around a TTP which acts as the trusted remote entity for the applications in its domain. Any application that wants to make use of this architecture must chose its TTP and thus enters the TD of the chosen TTP.

This chapter first explains the big picture of the presented architecture (Section 4.1). The subsequent sections (Sections 4.2 through 4.8) elaborate in detail on how the architecture is designed in order to solve the problem stated Section 2.5.

4.1 Big Picture

In this section the general idea for the architecture is explained in order to convey the big picture. With this big picture in mind, the following sections will present the details of the architecture and elaborate more specifically on the exact functionality.

As a prerequisite several Trusted Third Partys are assumed, e.g., the application vendors or independent parties, that establish their Trust Domains on the platform by means of a special bootstrapping procedure. Applications can then enter the Trust Domain of their selected TTP during the instantiation their own MTM. Figure 4.1 depicts the Trust Domain concept with several Trusted Third Partys and a central vendor who provides the architecture and the Mobile Trusted Module. In this setup the applications trust their chosen TTP and transitively the architecture vendor. The architecture does not impose the active involvement of the vendor during the trust establishment, i.e., a restricting central trust authority, but is open to alternative establishment methods that exclude an active vendor.

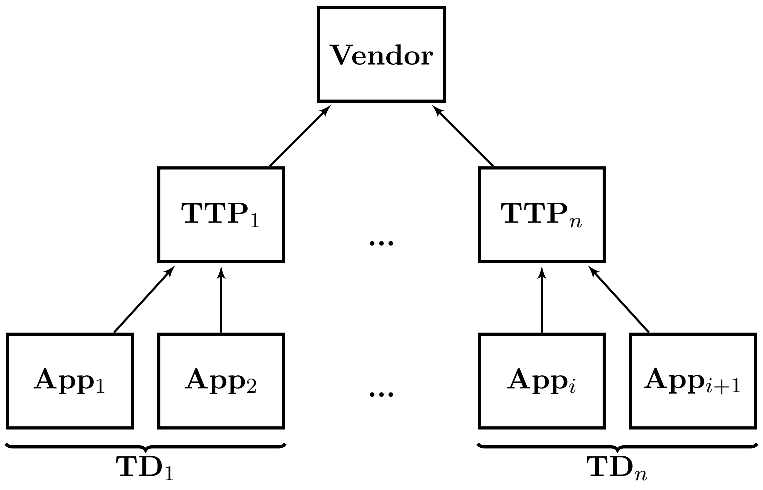

Figure 4.1: *Applications in different Trust Domains*

Figure 4.2 depicts the big picture for any application. The general motivation for the architecture is to provide each application with its own MTM instance. The applications leverage their specific MTM to use their credentials securely in protocols involving a remote party and to store their credentials securely when they are passive. Moreover, applications make use of the attestation functionality of the TPM to attest the MTM execution in the same manner as described for PALs in Section 2.4 (p. 20), but additionally include the application ID in the quote.

In contrast to the platform TPM, which is shared among all software on and users of the platform, the MTM is application-specific. The MTM is implemented in software, in compliance with the MTM specifications, and therefore requires strong isolation from other software and the operating

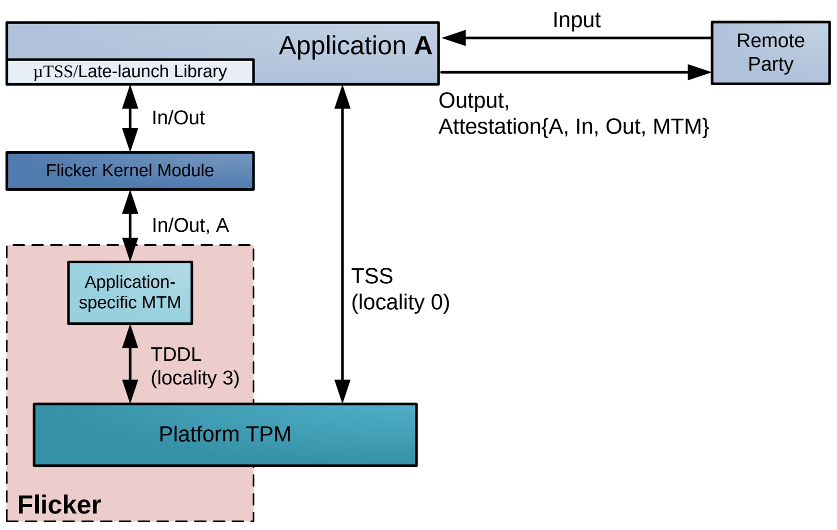

Figure 4.2: *Big picture of the credential storage and usage by applications*

system on the platform in order to protect the security sensitive computations and data (i.e., fulfill the Roots of Trust (RTs)). For that reason, the MTMs are put under the HW protection of DRTM by means of a Flicker session. Applications that want to make use of their specific MTM require a special library, or *shim*, which is responsible for the communication with the Flicker kernel module and which implements specific parts of the TCG Software Stack[1], denoted here as μTSS, since the MTM expects its commands in a specific format defined by the MTM and TPM specifications [115, p. 8].

This library communicates with the Flicker kernel module in order to set and retrieve I/O parameters for the MTM and to trigger the Flicker session. 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 the execution to the application.

The MTM makes use of the platform TPM as well (e.g., as randomness source or to read/extend PCR17), but in contrast to the legacy OS and applications, which access the TPM at locality 0 (or the legacy locality), the MTM accesses the TPM at locality 3. Therefore, any conflict in access to the TPM is avoided in the exact manner as described by the TPM specifications[115, Section 9].

For the MTM implementation in software, it is important to notice that an MTM consists of an immutable part, the MTM command logic, and a mutable

[1]a very compact explanation of the rather complex TSS is given in [26]

part, the MTM state. In this architecture, this fact is exploited in the same way as in [35] by splitting the MTM in these two parts. For this architecture, that means that only the mutable MTM state is application-specific, while the MTM functionality is identical for all application-specific MTMs. For that reason, the commands are implemented as PAL, that takes the state on which to operate as input.

The payload of each SLB is quite limited and in fact in this architecture the MTM consists of a set of PALs, each implementing a certain subset of the MTM logic. For each Trust Domain, these PALs share a domain-specific secret key, only known to PALs currently operating in the specific TD (illustrated in Figure 4.3). Certain PALs in this set play a special role, like the bootstrapping of an TD or creating a new, application-specific MTM state in a TD, and they will be explained separately in more detail in subsequent sections of this chapter.

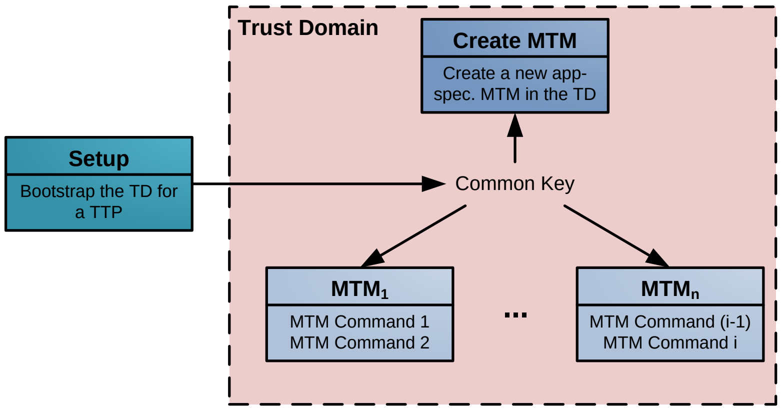

Figure 4.3: *Set of MTM PALs, bootstrapped for a particular Trust Domain*

The life-cycle of each application-specific MTM consists of the following three steps:

1. **Instantiation:** The MTM state is created by a special PAL. The created state is bound to the application and thus can only be operated on by this particular application. The created state includes application-specific information, that can be used later for verification and attestation purposes. Version information for each created state ensure that no old state data can be replayed.

2. **Operation:** The application utilises its MTM to store and use its credentials, e.g., sealed storage, cryptographic functions, or attestation.

3. **Deletion:** The application deletes its MTM by removing any stored MTM related data, like its MTM state, and informing its Trust Domain about the deletion in order to free resources that had been allocated for this MTM, like the state version information.

4.2 Application Identification

The first problem to solve is the secure and unambiguous identification of the application that initiates a Flicker session. In this architecture this problem is solved based on IBM Research's Integrity Measurement Architecture (IMA) [95].

An overview of the functionality of IMA was already given at the end of Section 2.1.1 on page 9. IMA is available by default in the Linux kernel since version 2.6.29.

For this thesis the IMA kernel module and the Flicker kernel module were patched such that Flicker is able to enquire the measurement of a process from the IMA subsystem. In particular, the IMA module was extended with a simple API call that returns the latest stored measurement for a specific inode in the file system. The additional interface adds solely 8 lines of code to the IMA code base. The Flicker module was patched such that it computes the aggregate hash for the calling application just before executing the late-launch. To do so, the module inspects the task structure of the calling process inside the kernel runtime system for the inodes of the executable, all memory mapped files (e.g., dynamically loaded libraries), and configuration files. It then requests the latest measurements for these files from IMA via the new interface and computes the aggregate hash of the returned values in a fashion similar to PCR extension (see page 6 in Section 2.1.1). The computed aggregate hash forms then the (load-time) identity of the application at call-time of the execution (or instantiation) command of its MTM. This hash is passed on by the Flicker module as an input to the PAL, where it is used as the identity to which the MTM is bound, i.e., under which the MTM operates application-specific.

4.3 Cryptographic Primitives

Although the MTM PALs have access to the platform TPM and its cryptographic functions, those functions are either to restrictive (e.g., created wrapped keys should be bound just to the MTM, but not additionally to the platform TPM) or are too inflexible (e.g., no symmetric cryptographic primitives) for the architecture. For these reasons, the MTM PALs include a small software library that provides such cryptographic primitives to the MTM.

Asymmetric Schemes

The asymmetric schemes provided for both encryption and signing are PKCS#1 v1.5 and v2.1 as defined in [57]. In particular RSAES-PKCS1-V1_5 and RSA with Optimal Asymmetric Encryption Padding (RSAES-OAEP) are available as encryption schemes, while RSASSA-PKCS1-V1_5 and RSA with Probabilistic Signature Scheme (RSASSA-PSS) are available as signing schemes with appendix.

Symmetric Scheme and Padding

The only provided symmetric scheme is AES [77] with a 128 bits key-length and in Cipher-Block Chaining (CBC) mode [78]. If the input length does not equal a multiple of the block length, padding is required. The library implements an unambiguous padding mechanism as specified in [49, p. 27].

Hash Functions

The sole hash function provided by the library is SHA-1 (Secure Hash Algorithm [34, 79]).

Message Authentication Code

The only Message Authentication Code implemented in the library is the Keyed-Hash Message Authentication Code (HMAC) [51] based on SHA-1.

Key Derivation

Key derivation is achieved using a *Key Derivation Function* (KDF) as specified e.g., in [80] or [58, 66]. In this implementation the recommendations of RFC2898 [58] are adhered to, i.e., using HMAC as the *Pseudorandom*

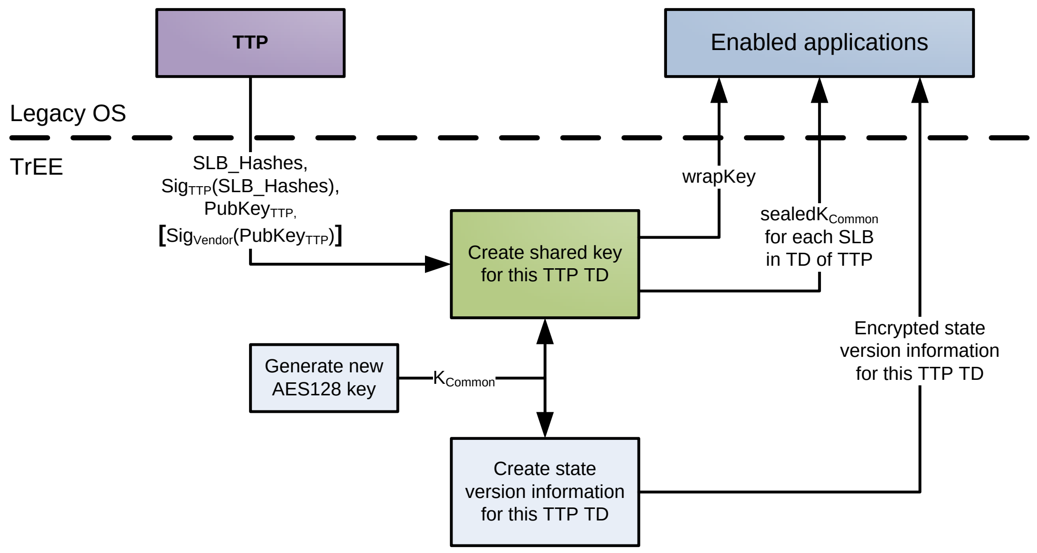
TTP
Enabled applications
Legacy OS
TrEE
SLB_Hashes, Sig_TTP(SLB_Hashes), PubKey_TTP, [Sig_Vendor(PubKey_TTP)]
wrapKey
sealedK_Common for each SLB in TD of TTP
Create shared key for this TTP TD
Encrypted state version information for this TTP TD
Generate new AES128 key
K_Common
Create state version information for this TTP TD

Figure 4.4: *High-level view of the bootstrapping of a Trust Domain*

Function (PRF) and PBKDF2 as derivation function. The used passwords for KDFs in this implementation are throughout AES-128 keys and thus the key length is bigger than half of the block-size of the used hash function (160 bits for SHA-1) and thus fulfills the recommended minimum key length for secure KDFs.

4.4 Trust Domain Bootstrapping

As described earlier in Section 4.1, the MTM PAL set contains special PALs that do not implement MTM functionality, but necessary mechanisms like bootstrapping a particular Trust Domain on a platform. In this section this bootstrapping process is explained in detail.

The PALs that belong to the Trust Domain of a particular Trusted Third Party share a secret key, denoted as K_{Common}. This key is set up for each TD by the bootstrapping PAL. The architecture design makes use of the sealing capabilities of the platform TPM and the special role of PCR17 during late-launch in order to guarantee the confidentiality, integrity, and authenticity of this shared keyK_{Common}. This means, that only the MTM PALs operating in the correct TD are able to access this key and that they have the means to ensure the authenticity and integrity of this key.

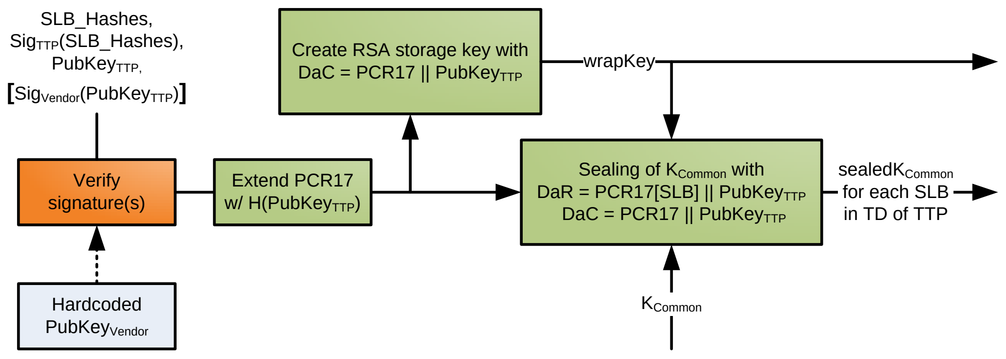
SLB_Hashes, Sig_TTP(SLB_Hashes), PubKey_TTP, [Sig_Vendor(PubKey_TTP)]
Verify signature(s)
Hardcoded PubKey_Vendor
Extend PCR17 w/ H(PubKey_TTP)
Create RSA storage key with DaC = PCR17 || PubKey_TTP
wrapKey
Sealing of K_Common with DaR = PCR17[SLB] || PubKey_TTP DaC = PCR17 || PubKey_TTP
sealedK_Common for each SLB in TD of TTP
K_Common

Figure 4.5: *Establishment of a common key for a set of PALs in the same Trust Domain*

Figure 4.4 gives an overview of the bootstrap PAL internals. The newly created K_{Common} is sealed once for each MTM PAL selected by the SLB hash array. The resulting seals and the wrapped sealing key are returned to the legacy environment for use by the applications. Moreover, the bootstrap PAL creates a central data structure that holds the version information for each application-specific MTM in this TD. This data is protected by a key derived from K_{Common} and is also returned to the legacy environment for persistent storage.

The exact mechanisms implemented for the creation of the shared key and for the protection of the MTM state version information are explained in the following.

The sealing of K_{Common} is illustrated in Figure 4.5. The PAL requires certain inputs: a) an array of hashes, each identifying an MTM SLB that operates in this domain; b) a signature of this array by the Trusted Third Party of the Trust Domain; c) the public key of the same Trusted Third Party; and d) optionally a signature of the TTP public key by the vendor of the MTM SLBs.

The bootstrap PAL first verifies the signature of the TTP for the hash-array based on the public key of the TTP. It further verifies the authenticity of the TTP public key based on a signature of the same by the vendor and the vendor public key, that is hardcoded in the PAL (and is thus also part of the PCR17 value of the Setup SLB). This second verification is optional in order to enable a more open architecture, since TTPs are not required to be authenticated by the vendor. Although the vendor signature mechanism provides the highest assurance, it is also restrictive, because all TTPs have to securely request the signature for their public key from the vendor. Hence, the vendor would be in absolute control of who is able to establish a TD. Alternative approaches for bootstrapping trust could be based on a secure user interface that shows, for instance, the fingerprint of the TTP public key whereby the user can verify this fingerprint manually against the correct value and abort if necessary. The setup could also be based on a secure remote attestation of the bootstrap PAL results.

Afterwards, PCR17 is extended with the SHA-1 hash of the TTP public key. The hash of the public key identifies the Trusted Third Party and thus the Trust Domain. The PAL then leverages the platform TPM to create a wrapped RSA storage key as child of the TPM SRK. A special information field *DigestAtCreation* (DaC) in the key structure, that indicates the composite value of selected PCRs at the time the key was created, is used to store the PCR17 value at the creation time. Thus, a verifier with knowledge of the hash of the Setup SLB and (the hash of) 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 Trust Domain of the particular TTP.

K_{Common} is an AES-128 key. It is sealed once for each MTM SLB with the created TPM storage key in order to protect its confidentiality and to further guarantee the authenticated access to it solely by the selected MTM SLBs of this Trust Domain. This authenticated access is accomplished through binding of the seals to the PCR17 values of the respective SLBs extended by the TTP public key hash (based on the *DigestAtRelease* (DaR) information of the sealed data structure). Those MTM SLB hashes are provided to the Setup as the hash-array in the input. For the same reason as for the wrapped key, the DigestAtCreation information of the seals is set to the PCR17 value at sealing time.

Figure 4.6 depicts the protection of the MTM state version information for the currently bootstrapping Trust Domain. The PAL derives a special key $K_{Version}$ from K_{Common}. The label used for the derivation is `version`. The derived key is then used for a symmetric encryption of the MTM state information for this Trust Domain.

The state version information is a special data structure to maintain the versions of all MTMs in one TD. This data structure requires a protected and dedicated counter in the TPM as a reference counter. Because of the restriction that only one TPM monotonic counter can be incremented (possibly multiple times) between two system reboots and thus prohibiting more than one active TD between reboots, we opted for a counter deployed in the TPM

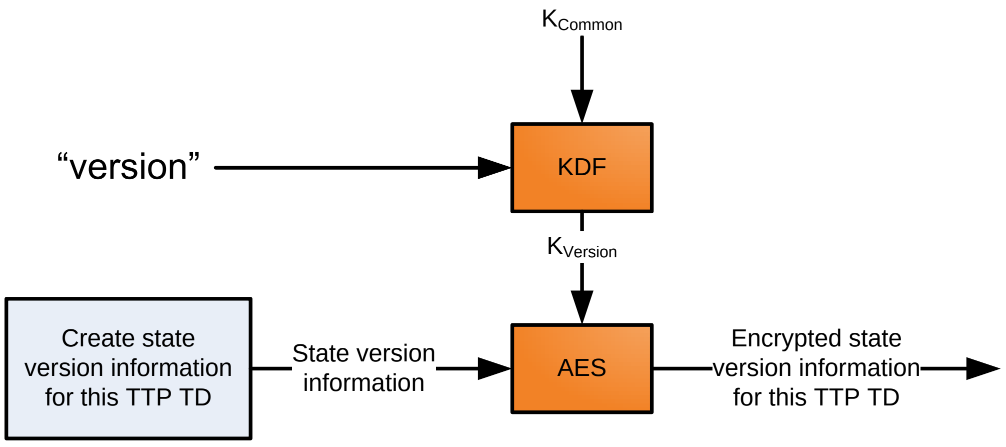
K_{Common}
"version"
KDF
$K_{Version}$
Create state version information for this TTP TD
State version information
AES
Encrypted state version information for this TTP TD

Figure 4.6: *Creation of the version information for MTM state in the same Trust Domain*

NVM. Since this memory is not maintained by the TPM as a monotonic counter, but simply as memory region, our architecture has to implement the necessary steps to control access to this memory and to implement the monotonic counter functionality in the SL code, using the provided memory. The index of the TPM NVM space allocated for the counter is stored in the state version information.

Based on the reference counter and the state version information, it is possible to determine the freshness of an MTM state and, moreover, these information are required to compute the encryption keys for MTM states.

We elaborate in Section 4.6 in more detail on how we control access to the reference counter and on how our architecture implements replay prevention for MTM states.

In total, the bootstrapping PAL returns the wrapped storage key, the seals of K_{Common}, and the encrypted state version information to the legacy execution environment, where they are persistently stored for further use by the MTM PALs.

4.5 Common Key Extraction

Except the bootstrapping PAL, every MTM PAL selected by the TTP should be able to access the shared key for this specific TTP Trust Domain, namely

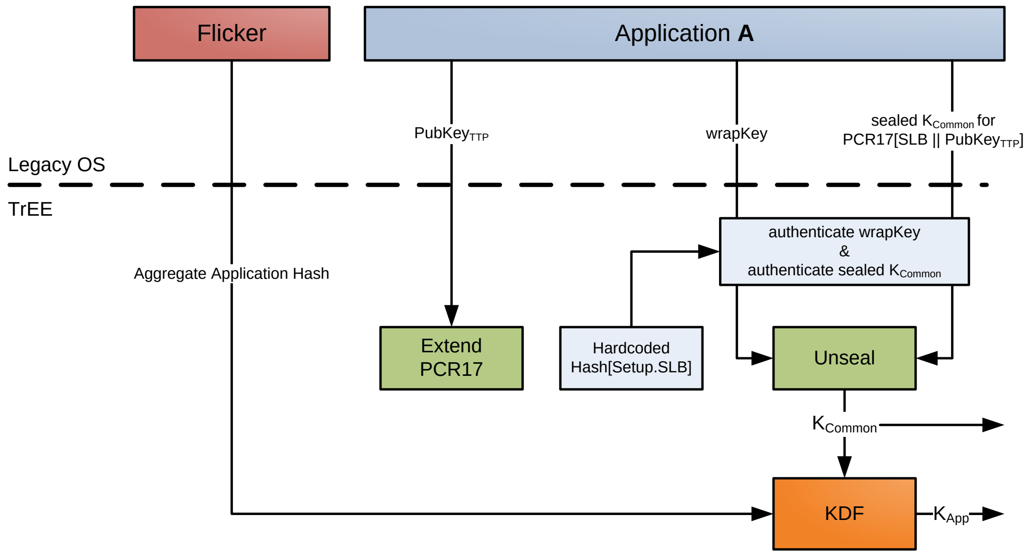
Flicker
Application A
PubKey$_{TTP}$
wrapKey
sealed K$_{Common}$ for PCR17[SLB || PubKey$_{TTP}$]
Legacy OS
TrEE
Aggregate Application Hash
authenticate wrapKey & authenticate sealed K$_{Common}$
Extend PCR17
Hardcoded Hash[Setup.SLB]
Unseal
K$_{Common}$
KDF
K$_{App}$

Figure 4.7: *Deriving the application-specific key*

the K_{Common} created during the bootstrapping of the TD. This shared key is the root key used to derive different dedicated keys, e.g., an application-specific key or an encryption key for the MTM state version information of the respective TD.

K_{Common} was sealed by the bootstrapping PAL and hence has to be unsealed by the other MTM PALs before usage. The common procedure for the shared key extraction, that includes ensuring the authenticity and integrity of the key, is illustrated in Figure 4.7 and explained in the following.

Each application that initiates a late-launched MTM execution has to provide a certain set of inputs, namely the wrapped storage key created by the bootstrapping PAL for the Trust Domain of the application, the sealed K_{Common} of this MTM PAL, and a hash of the public key of the TTP of the TD the application executes in. Further, the MTM PAL receives the aggregate hash of the application from the Flicker kernel module.

First, each MTM PAL extends the PCR17 with the hash of the received public key hash. Thus, the PCR17 value corresponds to the measurement of the executing MTM SLB extended with the public key hash and hence identifies the MTM PAL and the Trust Domain for the MTM execution.

Next, the PAL verifies the authenticity of the provided storage key and sealed K_{Common} based on the DigestAtCreation (DaC) value of both and on the hardcoded hash value of the trusted bootstrap SLB. If the hardcoded value equals the DaC value with only PCR17 being selected, the storage key and the sealed AES key were created by the correct bootstrapping PAL and can be trusted. Because the hash value is hardcoded into the code of each MTM PAL, it is part of the PCR17 measurement of each MTM SLB.

The integrity of the storage key and the sealed key are verified by the TPM at load and unsealing time, respectively. The unsealing of K_{Common} is only successful if the PCR17 value corresponds to the DigestAtRelease (DaR) of the sealed key, i.e., both the measurement of the MTM SLB and the public key hash must correspond to the values used at creation time during the bootstrapping.

After the successful authentication of the storage key and of the seal, the MTM PAL unseals the shared key by means of the TPM and the storage key. The unsealed key is common for all MTMs in this TD and is available for further usage by the MTM PAL on a domain wide level.

Additionally, an application-specific key K_{App} is derived from K_{Common}, based on the aggregate application hash. This derived key is unique for a specific application in a specific Trust Domain.

In the next sections, a successful extraction of K_{Common} and derivation of K_{App} is assumed and thus these keys are supposed to be available for the functions described.

4.6 MTM State Integrity and Versioning

In this section, the details of the state versioning are explained. The basic idea for the proposed solution are virtual monotonic counters using a TPM as described in [97]. In this approach a virtual counter is maintained for every instantiated MTM and freshness of the MTM states is verified based on these counters.

Figure 4.8 depicts the details of how the encryption key for MTM states, called K_{State}, is derived and the role of the virtual counters in this context. First, $K_{Version}$ is derived from K_{Common} and is used to decrypt the state version information for this Trust Domain. The state version information consists of a limited number of virtual counters, e.g., three 32-bit counters, which could be encrypted in exactly one block with AES-128 (4 bytes are reserved for the TPM NVM index of the reference counter). The maximum counter value is set to 2^{31}, because the most significant bit is used to indicate if the counter is assigned or free. Every instantiated MTM, i.e., MTM state, is tagged with a

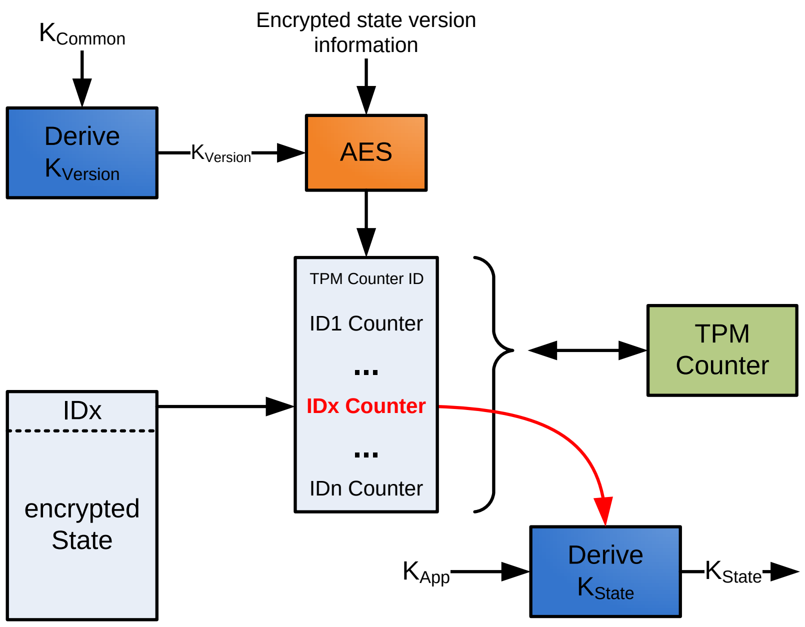

Figure 4.8: *Derivation of the encryption key for an MTM state within a particular TD*

domain-unique ID during the instantiation and this ID forms the index of the virtual counter assigned to this MTM. The counter ID and the corresponding virtual counter value are used in conjunction 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 and thus in failed sanity checks of the state.

As proposed in [97], 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 in synchronisation, and thus, if the sum and the reference counter value correspond to each other, the freshness of the virtual counters is assured, provided the virtual counters are integrity protected.

The TPM_DefineSpace command, for creating a new NVM space inside the TPM, offers the possibility to set a fine grained access control for the newly created space. For instance, the read and write access can be bound to selected PCR values, the locality, and the entity (i.e., either the owner or any other entity with knowledge of the correct authorisation value). In our implementation, different PALs require access to the NVM in order to read and write the counter value, thus prohibiting the binding to a specific PCR17 value. Therefore, 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"*. The authorisation value is hence only available to the PALs that are able to unseal K_{Common}. $K_{Version}$ was chosen as the derivation key, because it is domain specific just like the reference counter.

An approach based on TPM monotonic counters would require a similar setup, because monotonic counters require a particular authorisation value for incrementation (set during counter creation). However, reading the counter does not require any authorisation. In contrast to the NVM based approach, in which the counter functionality is implemented in the PAL and the TPM provides solely a secure non-volatile storage location, TPM monotonic counters are maintained by the TPM and no additional counter functionality, except the necessary TPM commands, is required in the PAL.

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.

In Figure 4.8 only a one-level deep hierarchy of virtual counters is illustrated, but further levels can easily be achieved with a mechanism similar to Merkle hash trees [43]. Virtual monotonic counters based on Merkle hash trees, as presented in [97], have already been shown to be suitable to establish storage with forking and replay detection [118]. In this tree-like counter data structure, only the leaves are dedicated virtual counters, while the intermediate and root counters contain the sum of their children. To verify the freshness of a specific leaf counter, one simply traverses the tree up to the root and checks on every intermediate node if the parent counter value equals the sum of its children, whereby the NVM reference counter is seen as the parent of the root node. Crucial for this solution is that every node in the tree is integrity protected in order to prevent the reallocation of counter values.

4.7 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

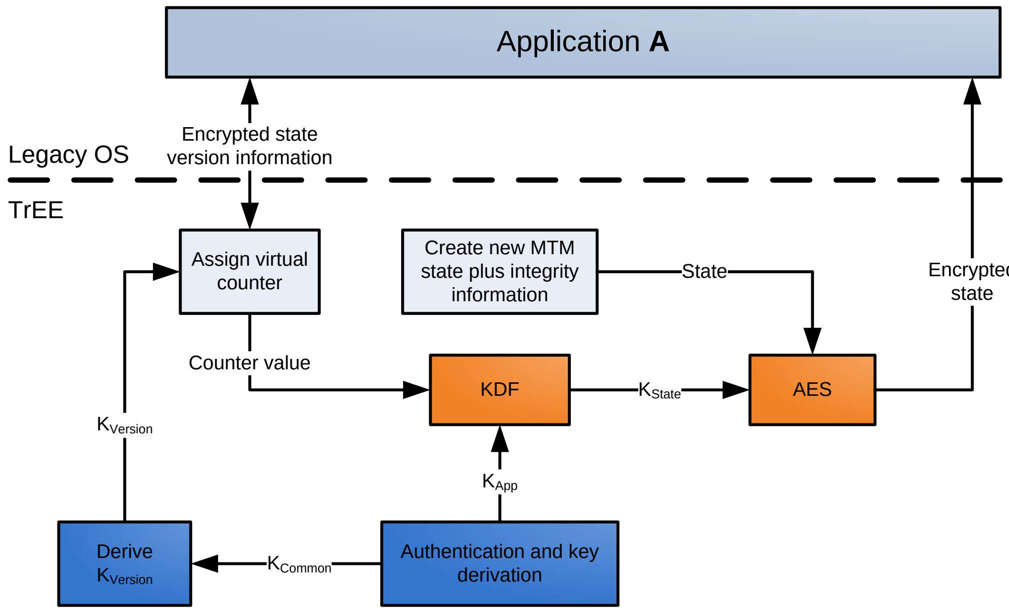

Figure 4.9: *Instantiation of a confidential, integrity protected, and application-specific MTM state*

application. The process of how an application creates an MTM state via an PAL, that is dedicated to this purpose, is illustrated in Figure 4.9.

The application has to provide the MTM state version information for its Trust Domain. This information was created by the bootstrapping PAL and is encrypted with $K_{Version}$, which is derived from K_{Common}. The PAL for the state instantiation assigns a virtual counter to the new state, Further, this counter is incremented by one. Thus, even if the identical application releases its MTM and creates a new state with the identical counter assigned, no replay attack is feasible.

Next, the PAL creates a new MTM state for the application and appends information for the integrity validation, i.e., a hash of the state. The state plus its integrity information is then symmetrically encrypted with 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-specific bound. The state is tagged with its plaintext virtual counter ID.

The state is returned to the application, which then is able to issue MTM commands on its specific MTM state.

A particular problem with the state creation for software-based MTMs is the secure provisioning of certain keys to the MTM. Normally, each TPM or MTM is shipped out by the vendors with a pre-installed *Endorsement Key* (EK), that identifies the module and hence the platform. The EK is used to enrol new identities, i.e., AIKs, on the platform, which are used for attestation purposes instead of the EK., because of privacy concerns.

Our implementations employs an MRTM for which we already possess a code-base [35] that meets the requirements for a deployment in execution environments with constraint resources, like the security extensions of contemporary System-on-Chip architectures or the late-launched environments of modern CPUs. The MTM specifications identify use-cases in which the enrolment of additional identities is not required and in which the EK for MRTMs is declared as optional, as long as one AIK, the SRK, and a Root Verification Authority Information (RVAI) (that defines the root of the verification key chain for RIM certificates) are installed during manufacturing of the MTM. The size optimisations for our MRTM include the abandonment of an EK.

Pre-installation of the mandatory credentials implies that they are already deployed in the MTM prior to shipment of the MTM to the user. In our architecture, however, the MTM state is created on the user platform, thus prohibiting a feasible pre-installation. In our architecture exist different approaches to this problem. For instance, the required credentials, like the AIK, are created in the context of the state instantiation. 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 an AIK by means of the AIK's 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 for the AIK or the AIKs certificate.

4.8 MTM Execution

Any application that has successfully created its own MTM state is able to use its MTM to store and use its credentials. While the MTM state forms

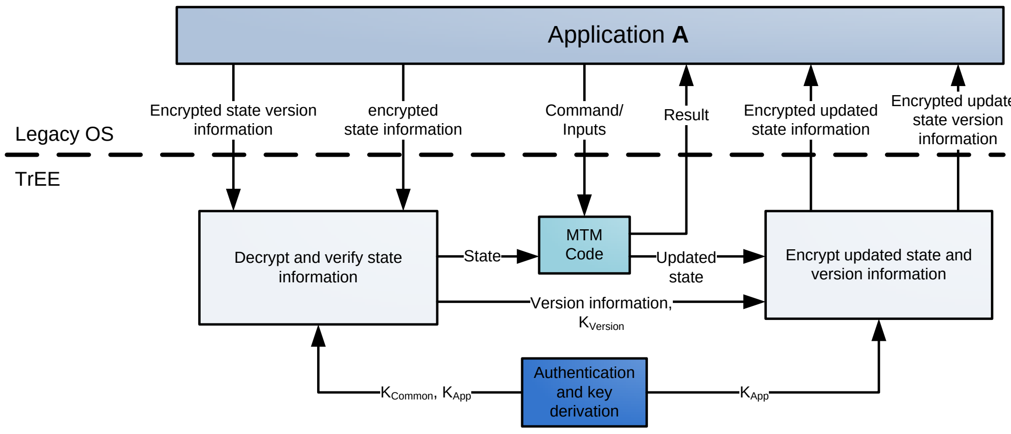
Application A
Encrypted state version information
encrypted state information
Command/ Inputs
Result
Encrypted updated state information
Encrypted updated state version information
Legacy OS
TrEE
Decrypt and verify state information
State
MTM Code
Updated state
Encrypt updated state and version information
Version information, $K_{Version}$
Authentication and key derivation
K_{Common}, K_{App}
K_{App}

Figure 4.10: *Execution of an application-specific MTM*

the application-specific data, the MTM functionality is implemented in one or several PALs. Each such PAL 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 PAL for the command that shall be executed.

Figure 4.10 illustrates the general workflow of an MTM PAL execution. Beside the application-specific MTM state information and the MTM command inputs, the application has to provide the usual input for the common key extraction and derivations as explained in Section 4.5. The workflow consists of three major steps (after the K_{Common} and K_{App} have been extracted/derived as illustrated in Figure 4.7 on page 47): 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 re-encryption of the updated MTM state.

Figure 4.11 depicts the details the first major step, the decryption and verification of the input MTM state. The key K_{State} is derived as explained in the preceding Section 4.6. The key $K_{Version}$ is further kept in memory for later usage. K_{State} is then used to decrypt the state and its appended integrity information. The decrypted state is afterwards checked for sanity. If the decrypted state is sane, this implies that it is fresh as well, since the used K_{State} is derived from the virtual counter information for the state version. The decrypted state version information for this TD is kept in memory as well for a later usage.

After the successful decryption of the MTM state, the received MTM command is executed on this state. The MTM results of this execution are returned to

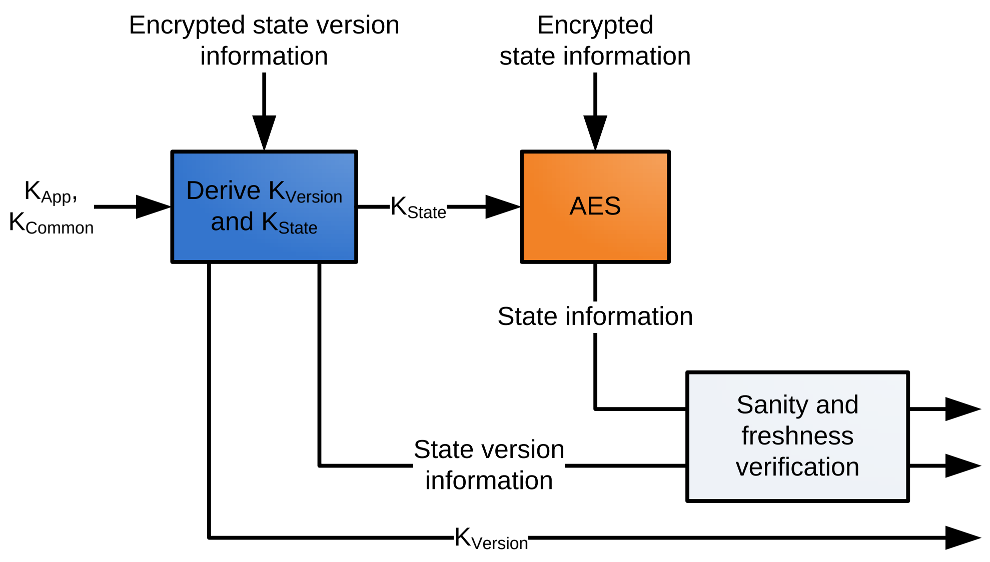
Encrypted state version information
Encrypted state information
K_{App}, K_{Common}
Derive $K_{Version}$ and K_{State}
K_{State}
AES
State information
Sanity and freshness verification
State version information
$K_{Version}$

Figure 4.11: *Decryption of the MTM state on entry to an MTM PAL*

the application.

As an effect of the MTM operation, the MTM state for this application has been modified and this updated state has to be re-encrypted before it can be returned. Figure 4.12 illustrates this state version update and re-encryption. Essentially, this step is the reversal of the decryption step, except that the state version information are updated, i.e., the virtual counter value for the current state (and all virtual counters depending on it) are incremented as well as the reference counter. A new key K_{State} is derived from the updated version information plus K_{App}. The updated version information are re-encrypted with the in-memory kept $K_{Version}$. The encrypted state version information and the MTM state, encrypted with the new K_{State}, are returned to the application.

Before returning to the legacy environment, the MTM PAL extends PCR17 with the hashes of the MTM input and output, as well as with the application ID. Furthermore, PCR17 is extended with a well-known value, that marks the end of the Flicker session and enables remote parties to distinguish between measurements taken in the secure environment and measurements taken afterwards in the untrusted legacy execution environment. The application is thus able to attest, via the platform TPM, the exact MTM SLB, its parameters, and the application it belongs to. Moreover, such an attestation, if successful, confirms the integrity of the MTM result to the remote verifier.

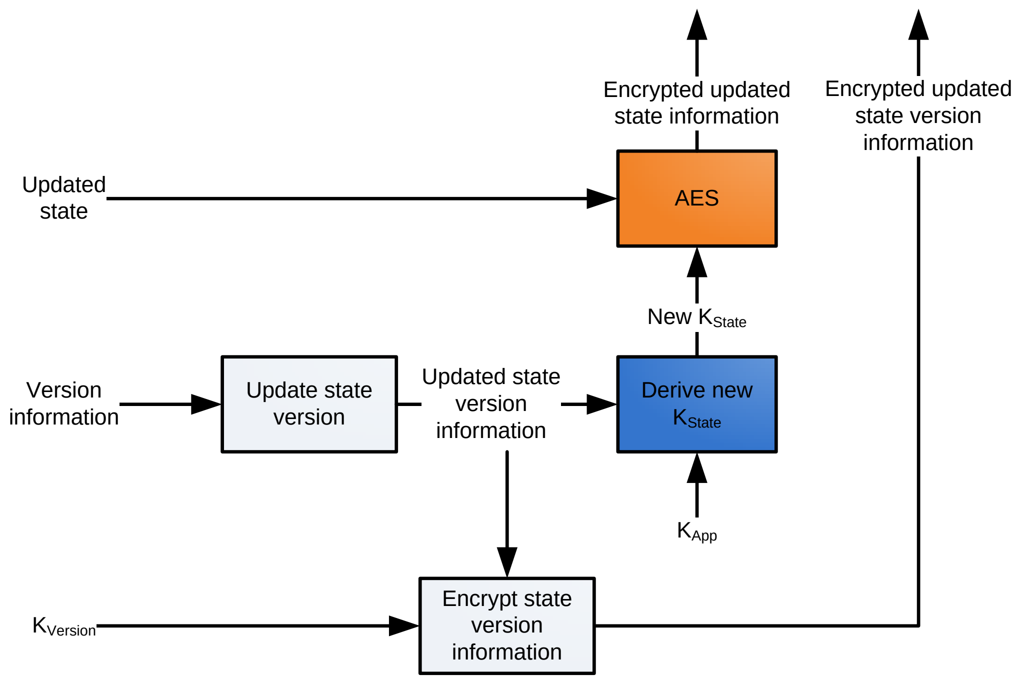
Encrypted updated state information
Encrypted updated state version information
Updated state
AES
New K_{State}
Version information
Update state version
Updated state version information
Derive new K_{State}
K_{App}
$K_{Version}$
Encrypt state version information

Figure 4.12: *Update of the MTM state version and encryption of the MTM state before the exit of the MTM PAL*

5

Security Analysis

In this chapter, the security properties of the proposed architecture are analysed. A special focus is put on the mechanism by which the architecture achieves the link between applications and the secure execution environment.

5.1 Assumptions

Certain assumptions regarding the cryptographic primitives, the DRTM technology, the TPM tamper-resistance, and the utilised MTM PALs have to be established in order to allow for a reasonable analysis of the architecture.

First, the cryptographic primitives are assumed to be secure against cryptanalytic attacks. In particular the encryption and digital signing algorithms can not be broken by an attacker that has no knowledge of the secret (private) key. Based on current publications on cryptanalysis of AES and RSA the assumed secure thresholds are 128 bits for AES and 1024 bits for RSA [40, 22, 21, 23, 61]. SHA-1 is used extensively in this architecture as it is the only supported hash algorithm by the current TPM specification. Although there have been recent advances towards breaking SHA-1 [120, 24, 25], it is assumed to be cryptographically secure in the context of this thesis.

The second assumption concerns the DRTM security. Recently an attack was published that circumvents the DRTM security features of Intel's TXT during a late-launch of the Xen hypervisor [123]. The attack is based on infecting the System Management Mode (SMM) handler of the platform, which than bypasses all security measures of the TXT late-launch. This is possible, since the SMM at the highest security level on the Intel platform and therefore remains unmeasured and unmodified by the DRTM. A solution proposed by

Intel, called SMM Transfer Module (STM), is to sandbox the SMM in order to prevent it from accessing protected resources [44, pp. 311-315]. Although STM is not available yet, in this thesis DRTM is assumed to provide the specified security features, which are fundamental for the proposed architecture, and SMM based attacks are not considered.

We further assume that the TPM chip is tamper-resistant as declared by the specifications. Recently, Christopher Tarnovsky has presented a successful attack on an Infineon TPM that disclosed secret keys protected by the chip [110][1]. The attack required a very high level of expertise and also special-purpose hardware, but it confirmed that TPM chips are not resistant against sophisticated hardware-based attacks. For this thesis, the security of TPMs as described in the TCG specifications is assumed, i.e., data stored within the TPM is considered as secrecy and integrity protected. Further, it is assumed that the TPM chip adheres to the specifications and does not violate them in a way that jeopardises the security of the architecture. This assumption is motivated by the fact that recent investigations revealed that some TPM models are not specification compliant [91].

The last assumption concerns the software security of the MTM PALs. The software-based MTM (including all required libraries) comprises roughly 10,000 lines of code. Considering this amount of code lines and the simplifications of the execution environment, e.g., no virtual memory, no peripheral devices except the TPM, and no scheduling, we assume that the unmodified deployed MTM PALs contain safe code. We argue that under this circumstances the code safety could be proven with formal methods or through code inspection. By definition, the Trusted Third Party (TTP) only signs hashes of these safe and trustworthy MTM SLBs.

5.2 Attacker Model

For the attacker we assume the same model as in the Flicker infrastructure and TPM specifications. The goal of the attacker is to disclose application's credentials either directly or by gaining access to secret keys that protect the credentials. He mounts attacks at runtime of the MTM PALs as well as at a time when the architecture is idle, i.e., against the sealed storage. Thereby, the attacks are targeted against the DRTM mechanism directly, but more commonly against the information protected by the MTM.

[1]The official slides are unfortunately extremely scarce regarding information, but more details can be found on several news websites

In terms of software-based attacks, the attacker is able to execute code as either user applications (ring 3), or with the highest privileges (ring 0) in the system, or as a separate late-launched PAL (ring -1). The attack code can run all operations on the target platform or make use of the resources of remote processes. Moreover, this includes that the attacker might compromise any application on the platform that makes use of an application-specific MTM as well as the OS.

Regarding hardware attacks, the attacker has physical access to the target platform and is able to perform simple hardware-based attacks ("open case attacks"), such as attaching malicious DMA devices or other attacks involving only commodity hardware tools and no special expertise.

This entails, that he does not have sophisticated tools like electron microscopes or other special probes which enable him to mount sophisticated hardware-based attacks. DRTM does not provide, by specification, resistance against this type of attacks and an attacker model that allows these, basically renders DRTM and thus the here proposed architecture ineffective.

Furthermore, certain TPM commands require authorisation, e.g., defining a new NVM space requires the owner authorisation or the usage of a TPM key, including the SRK, requires the key-specific authorisation value. In our architecture, unless explicitly stated different, all authorisation values are implicitly set to a well-known value, like 20 nulls. The attacker is thus able to issue these commands, because he has knowledge of these well-known values. However, it will be shown that this enables him only to mount denial-of-service attacks, e.g., changing the well-known authorisation values to secret ones or to consume all TPM resources. He will, however, not be able to disclose or misuse the application-specific MTM credentials based on this knowledge, as we will demonstrate in the following attack scenarios.

Although remote attestation plays an important role for the security that can be provided by our architecture, we exclude, for the sake of simplicity, attacks on network communications from this analysis. A mighty network attacker, e.g., Dolev-Yao [31], is able to mount attacks against the communication between the application and the remote party, but a thorough analysis would exceed the feasible boundaries for this thesis and we assume the existence of secure channel between these two parties.

5.3 Attack Scenarios

In the following, possible attacks scenarios based on the above described attacker model are explained and analysed. The scenarios are categorised into hardware-based and software-based attacks.

5.3.1 Hardware-based Attacks

The first attack scenarios that we analyze are hardware-based attacks [44, Chapters 17 & 20; pp. 131-132]. As explained in the attacker model, the attacker has physical access to the machine and can mount simple attacks. In this section, we base the protection against the examined attack vectors on the security features of DRTM. We analyse them from a general point of view that is not limited to AMD's SVM alone, but also includes Intel's TXT.

Hardware debuggers and malicious DMA devices: Two already mentioned scenarios are the attachment of a hardware debugger in order to read sensitive information directly from the hardware or a malicious DMA device that gains access to memory regions containing secret information. Both of these scenarios are countered by the late-launch implementations of Intel and AMD. As explained in Section 2.3, on AMD the `SKINIT` instruction disables all hardware debugging features, thus preventing an attacker from disclosing secrets with a hardware debugger. Moreover, it disables DMA access to the SLB region via the DEV vector. Similar measure that provide the same security benefits are taken on Intel platforms for the `SENTER` instruction.

Reset attack: Another scenario is based on the system reset. The attacker issues a system reset while the SL executes and boots directly afterwards attack code. Considering that the physical memory may sustain its content for several seconds after a power loss, it is possible for an attacker to read secret information from the memory after the reset. Intel's TXT counteracts this based on information stored in the persistent memory of the input/output controller hub (ICH, commonly known as “Southbridge”) and the TPM. This approach is necessary, since the CPU and the memory controller hub (MCH, commonly known as “Northbridge”) lose all information regarding the protection during reset. Both the TPM and the Northbridge are mandatory parts of the TCB on modern platforms. When booting after a system reset, the bootstrap processor checks, before the BIOS is entered, if previously a secure environment was executing and, if so, it executes a special module that cleans the memory regions that have been declared protected and thus all sensitive information.

ACPI attack: A similar attack is based on the S2/S3/S4 sleep states specified in the Advanced Configuration and Power Interface (ACPI). Events that are triggered during these states may cause some devices to perform a reset while other devices do not reset. For example, in S3 the CPU is reset, while the memory is still supplied with power, i.e., while secret information is still in memory, the CPU loses track of which memory regions have to be protected. In Intel's solution the responsibility is partly shifted to the code in the secure environment, which has to react to these events. It either could ignore them and prohibit the CPU to reset or it could take care of the necessary steps to protect the secrets, exit the secure environment, and allow the sleep state. If the code in the secure environment does not catch these events, while being active, the events cause the chipset to assume the above described situation in which sensitive information is left unprotected in the memory. It thus triggers a complete platform reset during which the secrets in memory are erased.

LPC bus attack: In a different scenario, the attacker tries to sniff secret information from the buses that interconnect the CPU, memory, and TPM. Since only simple attacks are allowed, the only bus left for sniffing is the Low Pin Count (LPC) bus, that connects the Southbridge with the TPM. The LPC is a slow bus and can be eavesdrop with commodity, low-cost equipment and little expertise. To counteract this attack, it is possible to establish a secure channel to the TPM, called *transport session* in the TCG specifications. The LPC is considered to be a network connection between two platforms and the transport session provides a protocol to setup a secure channel between the TPM and the PAL.

Attacks against the applications/OS: Hardware-based attacks that do not target the late-launched secure environment itself, but the applications or OS, could be for example hardware debuggers in order to reveal secret information kept in memory. In our architecture we prevent this type of attacks by not storing sensitive information in the application and rather keeping it in the application-specific MTM. Thus, the sensitive information is protected at runtime by the security capabilities of the secure environment and at passive time by the sealed storage capabilities of the TPM.

However, there is an exceptional scenario, namely the sealed storage use-case for applications. An application, that wants to seal information before storing them on the persistent storage, is required to encrypt these information themselves (e.g., AES) and then use their MTM to seal the used encryption key. This procedure is necessary due to the fact that the MTM only supports asymmetric sealing keys without any chaining, thus a very limited plaintext size. Moreover, even with chaining, the input buffer size of the secure environment sets a size limit that is most likely smaller than the size of the sensible information. The same procedure is described by the TPM specifications for users that want to make use of the platform TPM's sealing function. Our architecture is thus in this respect on the same security level than the TPM, except that our MTMs are application-specific and not shared.

Excluded advanced attacks: To conclude this section, a few advanced hardware-based attacks are listed. Again, this attacks are excluded from the attack vectors that the attacker model in this analyse is capable of, since they are explicitly excluded from the DRTM protection capabilities by the TCG specifications.

One thinkable attack would be the faking of the special bus-cycles issued by the CPU at locality 4 [89]. The attacker inserts these bus-cycles onto the LPC bus, thus causing the TPM to reset the dynamic PCRs. The attacker is than able to simulate a trusted PAL execution although malicious code executed. This attack requires special equipment and a certain expertise.

Other bus-based attacks could target the connection between the CPU and the Northbridge, i.e., the Front Side Bus, or the connection between memory and Northbridge, i.e., the memory bus. On both buses the attacker could intercept secret information. But both of them are high-speed buses and eavesdropping them requires a high expertise and expensive, special-purpose hardware tools.

Also an scenario involving a rogue CPU is possible. The rogue CPU is modified so that it reveals sensitive information to the attacker. If the BSP is not the rogue CPU but one of the application processors, the attack might still be successful if the rogue AP is activated by the SL (after certain preparations) in order to enable multiprocessing. But this scenario requires the attacker to obtain or build a rogue CPU and thus requires a very high level of expertise and tool support.

5.3.2 Software-based Attacks

The second type of attacks that we analyse are software-based attacks. In this scenarios the attacker tries to gain secret information by mounting software-based attacks against the secure execution environment, our deployed architecture, the system-level software, or the applications. The attack code can either execute as normal application with ring 3 privileges, with system level ring 0 privileges, or as PAL in a Flicker session.

Attacks against the secure environment: These attacks require that the malicious code runs in parallel or concurrent to the secure environment. But these prerequisites are explicitly prevented by the DRTM design. A parallel execution is not possible, because the application processors must be halted previous to the late-launch and the SL executes on the only remaining processor, the BSP. In our architecture, the BSP stays the sole active CPU during the PAL execution and the halted applications processors are only re-activated after the resumption of the legacy OS, i.e., when the credentials and keys are sealed again and any secret information is cleaned from the memory.

A concurrent execution is prevented by disabling the global interrupts flag during late-launch. The SL executes therefore atomically and no malicious code can regain control during the SL execution.

Malicious code that executes as PAL in separate Flicker sessions poses no threat to other sessions, since the sessions are mutually exclusive and the strong isolation properties of DRTM and TPM, i.e., the PAL measurement mechanism and sealing bound to this measurements, prevent that any information from one session can leak to another (or the legacy OS, for that matter) as long as the SLs clean their memory before exiting.

Attacks against the architecture: In this scenario, we analyse attacks targeted against the implementation of our architecture, that means the data deployed by our architecture on the persistent storage. These data comprises the SLBs and all information that are required as input parameters by the PALs, like the information supplied by the TTP or the wrapped/sealed keys. The information about the MTM state versions are analysed separately in Section 5.3.4.

The essential capabilities of the TPM and DRTM that are used to establish trust among the PALs and keys in our architecture are 1) the integrity protected information in TPM wrapped keys and seals about their respective origin (DigestAtCreation); 2) the possibility to bind keys and seals to a specific platform state (DigestAtRelease); 3) the unique PCR17 values for SLBs measured during late-launch; 4) the secure attestation of PCR values to a remote party.

Figure 5.1 illustrates how these capabilities are applied in our architecture in order to resist attacks based on malicious PALs and TTP input. The three components that exchange information in the architecture are the Trusted Third Party of the Trust Domain, in which the MTM should operate, the Setup PAL that bootstraps the Trust Domain, and the MTM PALs that implement the MTM functionality and state instantiation. Guaranteeing

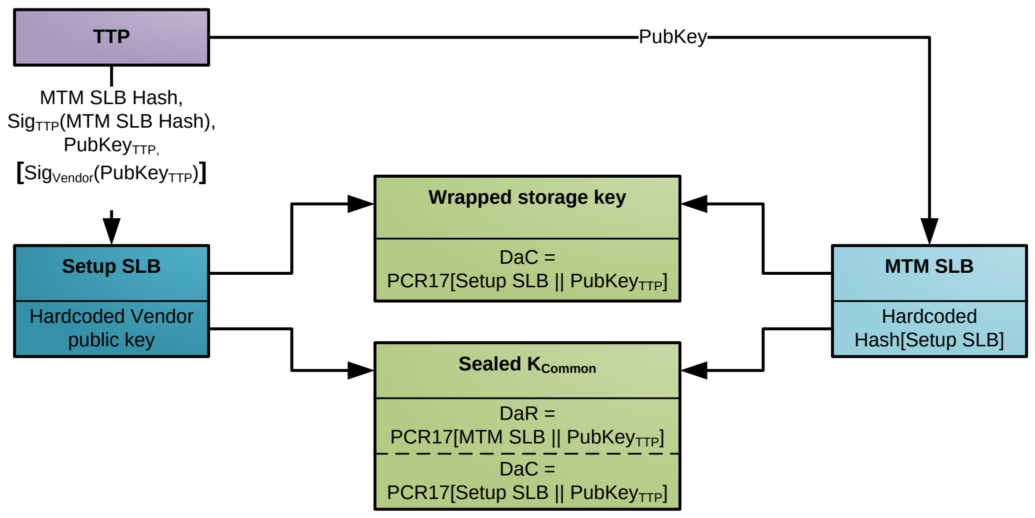
TTP
PubKey
MTM SLB Hash,
Sig$_{TTP}$(MTM SLB Hash),
PubKey$_{TTP}$,
[Sig$_{Vendor}$(PubKey$_{TTP}$)]
Wrapped storage key
DaC =
PCR17[Setup SLB || PubKey$_{TTP}$]
Setup SLB
Hardcoded Vendor public key
MTM SLB
Hardcoded Hash[Setup SLB]
Sealed K$_{Common}$
DaR =
PCR17[MTM SLB || PubKey$_{TTP}$]
DaC =
PCR17[Setup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

Figure 5.1: *PAL trust establishment within the architecture*

the basic trustworthiness of the PALs is the fundamental requirement for providing the necessary trusted resources for a software-based Mobile Trusted Module.

The following list gives an overview of the possible attack scenarios and how our architecture utilises the above mentioned capabilities to counter them:

- **Tampered TTP input:** The attacker is able to modify the input to the Setup PAL and exchange the public key of the TTP with his own. This is possible, because the vendor signature of the TTP public key is optional for reasons of openness. However, he does not gain any secret information, but solely exchanged the identifier of the TTP and thus the TD. The consequences could be denial of service at most, since the wrong TD would be detected by a remote attestation of the MTM results.

- **Malicious Setup PAL:** The attacker modifies the Setup PAL such that it behaves maliciously, e.g., it uses a well-known value as K_{Common} so that the attacker is able to derive all keys and disclose the application-credentials. Hereby, the modified Setup PAL uses the original input from the TTP. However, the MTM PALs expect a wrapped storage key of the TPM that has been used to seal K_{Common}. Wrapped keys can provide information about the PCRs at creation time (DigestAtCreation). Here in particular the DRTM measurement of the Setup SLB extended with the hash of the TTP public key is expected as DaC in PCR17. Since this specific value can only be achieved through the late-launch of the unmodified Setup SLB, the MTM PALs are able to detect when the storage key was not created by an unmodified Setup PAL (or if the storage key and TTP public key do not belong to the same TD). If the check fails, they consequently abort the potentially harmful operation.

If the Setup PAL was modified after it bootstrapped the TD, and hence the correct wrapped storage key is already created, the modified version is able to use this storage key to seal another, insecure K_{Common}. In this case the MTM PALs accept the DaC of the storage key, but they detect the untrusted DaC value of the sealed K_{Common}. Again, this is based on the fact, that the correct DaC of the seal can only be achieved through a late-launch of the unmodified Setup SLB with the correct TTP public key. If the authenticity and trustworthiness of the seal can not be verified, the MTM PALs abort the operation before any information can leak.

The same argumentation also holds, if the attacker additionally modifies the input from the TTP to the Setup PAL, since the crucial point is the detectable modification of the Setup SLB. Software is not able to forge these specific PCR17 values, due to the PCR extension mechanism (p. 6) and the fact that only a late-launch can reset PCR17 (p. 14). Thus, PCR17 attests in a trustworthy manner the exact identity of the software executing in the secure environment.

- **Modified MTM PAL:** The MTM PALs are also available to the attacker and so he is able to modify them such that they leak sensible information to him. In this scenario he leaves the Setup SLB and TTP input unchanged. The modified MTM PALs are able to load and use the wrapped storage key. However, they are not able to unseal K_{Common}. The seal is bound to the value of PCR17 in such a way that only the MTM PALs that were selected by the TTP (MTM SLB Hashes) are able to unseal the common key. This is enforced by the TPM unsealing function (DigestAtRelease) and the measurement of the MTM SLBs into PCR17 by DRTM.

  If an attacker additionally modifies the input from the TTP such that he provides the MTM SLB hashes of his modified versions to the Setup, signed them with his own key pair, and omits the vendor signature, the malicious MTM PALs would operate successfully together with the correct Setup PAL. However, when the credentials are used in conjunction with a remote party, this party would detect the modified

TTP public key during the remote attestation.

- **Modified Setup and MTM PALs:** In this scenario, the attacker modifies the Setup PAL and the MTM PALs, but not (necessarily) the TTP input. The modified Setup PAL ignores the TTP input and creates a storage key, which it uses to seal a (well-known) key as K_{Common} but without any selected PCRs for the DaR of the seal. The modified MTM PALs ignore the DaC of the storage key and seal and are able to unseal the K_{Common}. Although, in this scenario malicious PALs are operating successfully and with the correct public key of the TD TTP, the malicious MTM PALs would be detected by the remote party, because their measurement is part of PCR17 which in turn is the basis for the remote attestation. The applications are hence able to abort their operations. Thus, the attacker can disclose only credentials that are rejected by the remote party.

- **Parallel malicious installation:** The remote attestation is an essential security factor in the architecture in order to detect modified PALs and inputs before the credentials are actually used between the application and a remote party. To achieve a successful remote attestation, while still deceiving the application to use malicious code, the attacker could install his malicious version of the architecture in parallel to the trusted one. The applications operate on the modified version, thus revealing the credentials, and the attacker's goal is then to produce the same, trustworthy attested output with the unmodified architecture as he produced with his malicious version. But since he has no insights into the isolated PAL executions and MTM states, the chances for this attack to succeed are minuscule.

Attacks against the applications and OS: Besides the secure execution environment and our architecture, an attacker could target the platform's applications and OS. In this respect the proposed architecture can not provide a better level of protection than its building blocks Flicker and TPM. Both of them provide a secure execution environment or shielded storage location respectively, but also require OS involvement to some extent. Flicker provides a kernel module through which applications write their inputs, read the outputs, and select the SLB to execute. Similarly, the TPM is accessed through drivers within the kernel and optionally through a software stack running in user- and kernel-space. Our architecture requires additional components of the kernel, namely IMA and the process management, for the application identification.

Although software-based attacks are not able to disclose any information at runtime and the MTM states are sealed for secure storage, malware and malicious code are able to misuse the application-specific MTMs or log, in certain use cases, application-level secrets.

As an example for the latter case, consider the secure storage use case in which an application wants to seal some data by encrypting it with a symmetric key and than sealing this key with its MTM. In this case, malware could log the symmetric key before it is sealed, because this operation is outside the protection perimeter of the architecture. In this scenario neither Flicker nor TPM would provide better protection, since in both cases the application had to perform the same operation and give the symmetric key to a Flicker PAL or the TPM as input via the OS.

Certain malware types, such as rootkits, are able to misuse credentials stored in an application-specifc MTM by undermining the complete OS security. Rootkits, for example, use operating system methods like hooking or replacing parts of the OS in order to compromise the OS (also known as kernel-level rootkits [48]) or exploit hardware features of the platform to bypass security capabilities of the OS (hypervisor rootkits [55] or user-level rootkits [111]). A disclosure of the credentials is, however, even for rootkits not possible, because the OS is not part of the TCB of our architecture due to Flicker.

Some other attack scenarios on how a MTM could be misused by other (malicious) software are also possible. For instance, if the IMA subsystem of the kernel is compromised, the application ID acquired by the Flicker kernel module is untrusted. The MTM execution and the MTM state are hence bound to the untrusted value. E.g. if the acquired ID is constant independently from the actually executing application, the MTM is not application-specifically bound anymore, but global to all applications. In this case, our architecture provides the same security level as Flicker, since the link between applications and the secure environment is lost. The same result could be achieved with a compromised Flicker kernel module that discards the IMA measurements or with a compromised process management in the kernel that provides false or incomplete information about the applications.

A further issue is the Time of Check to 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. But the applications are not part of the TCB of their MTM and although they can use their credentials, they are not able to access them in unencrypted form. Thus, the compromised application is able to misuse

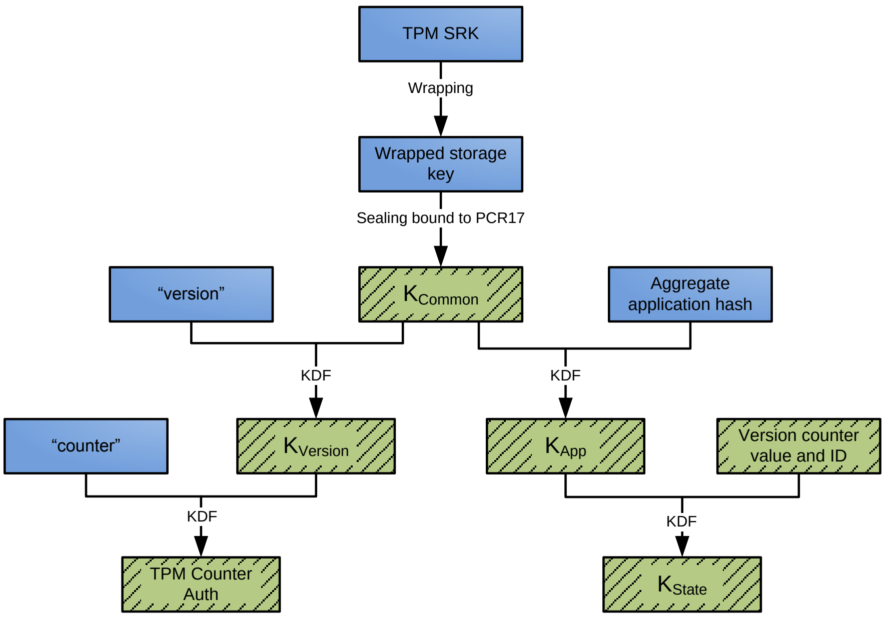

Figure 5.2: *Key-chain in the architecture*

its credentials, but not disclose them. This mechanism is similar to TPMs, where users can create and use keys on the module, but because the keys are wrapped, the user will never see the private portion of their keys in plaintext.

5.3.3 Key-Chain

In this section we analyse the confidentiality of the secret keys that are required for the MTM execution. Figure 5.2 depicts the relationships between the individual keys. Because the keys are either created by the TPM, sealed with a TPM key, or are derived from another key, they form a chain with the TPM Storage Root Key (SRK) as anchor. Keys unshaded are publicly available, while the shaded keys are protected such that they are only available to the legitimate MTM PALs.

The TPM SRK is on top of the key chain and it is by specification protected inside the TPM's shielded memory. However, the SRK can be used by any entity, i.e., user or application, that has knowledge of the SRK authorisation value.

The wrapped storage key, created during the bootstrapping of a Trust Domain, is created is a child of the SRK. Thus, the private portion of the wrapped key is never revealed in plaintext outside the TPM. The wrapped key is used domain-specific and thus can not be bound to a specific platform state via PCR17, since all MTM PALs in this domain require access to this key. Consequently, this key can be loaded and used by any party on the platform.

The wrapped key is used to seal the secret key K_{Common}. The important step that guarantees that only the legitimate MTM PALs of this domain have access to the confidential K_{Common} is that the sealed K_{Common} is bound to the specific PCR17 values of the unmodified MTM SLBs in the TD. Because the late-launch of each SLB results in a different, specific PCR17 value, K_{Common} is sealed once for every MTM PAL. Theoretically, the Setup PAL could create one wrapped storage key for each MTM PAL, bind these keys via PCR17 to the MTM SLBs, and then seal K_{Common} once with every wrapped key. Thus, the wrapped storage keys would not be usable by illegitimate entities. However, the storage key creation is a very expensive operation, and the increased costs in bootstrapping time and management of the additional keys outweigh the increase in security. The attacker is in our architecture not able to unseal K_{Common}, as argued earlier in Section 5.3.2. He is, however, able to use the storage key to seal and unseal other data and thus to mount cryptanalytic attacks against the key, like known-plaintext attack. But we assume and argue that the RSA implementation of the TPM with 2048 bits key-length is resistant against these attacks and that the costs of breaking one TD are most likely higher than its reward. A different approach to increase the security of the wrapped storage key would be to use a secure authorisation value, e.g., 20 random bytes, to authenticate the usage of the key and prevent unauthorised entities from doing so. However, the authentication value has to be mediated securely to the MTM PALs and the wrapped storage key is created to fulfil just this task. Thus, an authentication for the wrapped storage key yields a circular dependency which is not feasible without to implement a huge management overhead that again outweighs the benefits.

K_{Common} is used by the MTM PALs to derive other keys from it. First of it all, K_{Common} is domain-specific and not bound to any specific application in its domain. Therefore, the MTM uses K_{Common} to derive the encryption key for the domain-wide MTM state version information from it. The string used for the derivation is publicly known, namely *"version"*, but K_{Common} is confidential and thus the derived key $K_{Version}$ is secret as well. If the attacker is able to break $K_{Version}$, he is able to decrypt the version information for this domain and thus to replay old states by modifying the counter values. The HMAC-SHA-1 based KDF, however, provides the security benefit, that he is not able able to compute K_{Common}, which he requires to disclose credentials.

The authorisation value required for access to the NVM space of the reference counter, that is assigned to this Trust Domain, is derived from $K_{Version}$ as previously explained in Section 4.6. An attacker with knowledge of the authorisation value is, however, not able to compute $K_{Version}$ due to the properties of HMAC-SHA-1, used for the derivation.

The second key derived from K_{Common} is K_{App}, which is domain- and application-specific. K_{App} is bound to the application via the application's aggregate hash used for its derivation. Again, since K_{Common} is secret, so is K_{App}.

K_{App} itself is never used in an encryption or authentication operation, but serves as the secret key from which K_{State} is derived, which is used for the encryption of the MTM state and thus the secure storage of the credentials. In addition to the domain and the application, K_{State} is bound to the version of the latest MTM state. It thus offers a more fine-grained security and perfect forward secrecy, since an attacker might break the K_{State} of one MTM state and disclose the credentials of that version, but he is not able to break a long-term secret like K_{App} or K_{Common} this way. Although the version counter is secret as well in this derivation, it does not lower the need for strong protection of K_{App}, since an attacker with knowledge of K_{App} is able to easily brute force the very small value space of the version counter in order to break the state encryption.

Concluding, we want to emphasise again the importance of K_{Common} in our architecture. An attacker is only able to disclose credentials by breaking keys, if he gains knowledge of K_{Common} or K_{App}. K_{State} also provides him this benefit, but solely for one specific state version, while $K_{Version}$ enables him to replay old states. Bearing in mind, that K_{App} can only be computed with knowledge of K_{Common}, this common key becomes the single point of failure in the key hierarchy under the assumptions that the attacker is not able to break the TPM and DRTM protections.

5.3.4 MTM State Replay Attacks

In this section we want to elaborate in more detail on the mechanics of how our architecture prevents replay attacks of old MTM states.

The architectural design of the replay prevention is explained in detail in Section *4.6 MTM State Integrity and Versioning*. The key $K_{Version}$ is used to protect the domain-specific version information, that are implemented as an array of virtual counters. Each MTM in the domain is assigned one virtual counter, that reflects the latest state version of its MTM. The virtual counter integrity is assured based on a monotonic NVM based reference counter and the encryption of the version information. The virtual counters are used to derive K_{State} for the en-/decryption of the application-specific MTM states.

Under the assumption, that K_{Common} and hence the other derived keys are unknown to the attacker and further under the assumption, that the operations depicted in Figure 4.8 (p. 49) are protected by the Flicker session, the attacker is only able to attack the information stored on the persistent storage, i.e., the MTM state file and the version information, or the counter value stored in the TPM NVM.

In the first scenario the attacker targets the MTM state file saved on persistent storage. He modifies the state file and, since the data structure of the state is known, the attacker knows the offsets of specific information in the state, so that he is able to manipulate them specifically. The state file is encrypted with AES-128 in CBC mode. This means, that the attacker can not determine the new values of the changed data. This further entails, that he is not able to successfully adapt the encrypted integrity value attached to the state. Thus, the integrity check of the decrypted state fails and MTM execution aborts.

Each MTM state is tagged with an ID, that indicates the virtual counter assigned, and this ID is attached in plaintext. Thus, the attacker is able to easily change the ID. However, the K_{State} derived in this case is always incorrect and thus the decrypted state will fail the sanity and integrity checks. The reason that the derived K_{State} is incorrect is that the derivation takes the counter value and the counter ID into account. Thus, even if the modified ID points to a counter with the same value as the correct one, the changed ID value will cause the derivation of an incorrect K_{State}.

A similar detection of an attack occurs if the attacker targets the version information. These are also encrypted with AES-128. The attacker knows the offsets for the particular counter and the NVM space index of the reference counter. He is thus able to modify specific counter values. The consequences, however, are twofold. If the sum of the virtual counters in the modified version does not equal the value of the referenced NVM counter, the version information are considered untrusted and the MTM aborts execution. On the other hand, if the sum and the counter equal to each other, there are again two possible consequences depending on the changed value of the counter assigned to the current MTM. If this value is incorrect, the derived K_{State} is wrong and the integrity check for the state fails. If the value, however, is correct, the modifications to the version information remain undetected during this session. The MTM executes normally and returns the updated, modified version information. The attacker has not gained any information, but it is very likely that one or more of the other MTMs in this domain are now non-functional, because of incorrect counter values in the modified version information. Although this last case could be prevented through additional integrity information of the version information, we argue that this overhead is unnecessary, because this scenario is extremely improbable and, more importantly, the attacker does not gain any information, but just achieved a denial-of-service attack, which he can mount more easily, as he has access to the state file.

The last element, that the attacker could target, is the NVM space of the reference counter. Although, the space index of the TD is located within the encrypted state version information, it is easy for an attacker to obtain this index, because he can request a list of all defined NVM spaces and their corresponding attributes from the TPM (without any authorisation) and thus investigate which index belongs to the TD. Reading and writing to the NVM, however, does require authorisation. TPM authentication values are 20 bytes long. In our architecture, the Setup PAL, which creates the space for the TD reference counter, uses an HMAC-SHA-1 derivation of $K_{Version}$ as authentication value. An attacker could consequently try to acquire the $K_{Version}$ either by a cryptanalytic attack on the version information or by a brute-force attack against the NVM space authentication. The former attack is already excluded, because we assume the resistance of AES against this type of attacks. The latter attack is infeasible for two reasons: first, the key space to search through is sufficiently big (2^{160}), and second, the TPM implements *"anti-hammering"* methods, that block the TPM for a certain time period if too many failed authentication attempts occurred within a short time frame.

An attacker with knowledge of the owner authentication value, which is well-known in our architecture, is able to delete and to redefine NVM spaces. However, because our PALs expect to be able to read and write the NVM space of the reference counter based on the derived authentication value, which is unknown to the attacker, a redefined NVM space can have the same index as before, but a different authentication value. Therefore, a redefinition equals a space deletion and causes a denial of service, but does not provide any attack vectors that may disclose credentials.

5.4 MTM Roots of Trust

To conclude the security analysis, we analyse how our architecture meets the required Roots of Trust (RTs) for a secure implementation of a software-based MTM. To understand the meaning and importance of the RTs we first have to elaborate a little bit on the definitions of resources in an MTM architecture.

The MTM specifications define different kind of resources available for MTM implementations. The resources are distinguished by the way trust is placed onto them or by their availability.

The RTs are also denoted as “Trusted Resources”, because their trustworthiness is established solely on the surety by another entity. The second way to establish trust in a generic instantiation of a resource is by measurement through a reliable entity. These resources are therefore denoted as “Measured Resources”.

In terms of availability, the specifications distinguish between dedicated and allocated resources. The former ones are automatically available when the platform powers up, e.g., hardware devices or processors. Dedicated resources exist therefore by definition parallel to and isolated from each other. The RTs are specified as dedicated resources. Allocated resources are functions, that are created and maintained by dedicated or other allocated resources. Each stand-alone trusted mobile platform is required have some dedicated resources that can protect data from attacks.

Important for our architecture is the fact that the specifications states that a dedicated RT can also be described as an allocated RT that has been measured by a trusted building entity. Consequently, we are able to implement our software-based MTM, if our platform provides dedicated resources that form an RT and that are able to measure and vouch for allocated resources and thus build other Roots of Trust.

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

An RT composed of dedicated resources is required to perform a self-test and shut-down in the failure case. The RTE states, that the integrity and authenticity of the MTM execution environment has to be guaranteed with platform-specific means. This self-test is in our architecture implemented as the check for the correct DaC of the wrapped storage key, the correct DaC/DaR of the sealed K_{Common}, the remote attestation of the MTM SLB, and the replay prevention of MTM states. If either of these checks fails, the MTM is not functional and shuts down, because it is not able to access its state or because the failed attestation indicates a potential attack on the integrity of the PALs.

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. If the RTE passed the self-test, that means the trusted MTM PAL is able to unseal the K_{Common} and thus to derive all other keys, it has access to the MTM state. The state implements the PCRs of the MTM and the protected storage is provided by K_{Common} and its successors in the key chain, because only the legitimate MTM PALs are able to derive this key.

Root of Trust for Reporting (RTR): The RTR reports the measurements stored by the RTS. It further signs them with a key that is also stored by the RTS. Thus 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 Reference Integrity Metrics (RIMs) and extending the integrity metrics into PCRs. Some immutable or protected code forms the RTV and, exactly like for the RTR, this code is implemented in the MTM PALs and is hence protected by the RTE.

6 Implementation

In this chapter, we provide details about the actual implementation of our architecture in terms of design, code size, performance, and constraints. The details include the Secure Loader Blocks as well as a simple software stack for applications that use an MTM.

The implementation was tested on two different platforms. The first platform is an HP Compaq dc5750 with a Broadcom BCM0102 TPMv1.2 chip[1] and an AMD Athlon64 X2 4400+ Dual Core CPU (2,3 GHz). The second platform is 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 thesis.

Unfortunately, this thesis can not present a fully functional prototype implementation due to unresolved hardware problems with the test platforms during late-launch. An analysis and log of the attempted approaches to resolve these problems are provided in this chapter.

6.1 Hardware Problem Analysis

The problem encountered with both test machines is an incorrect extension of PCR17 during late-launch of the Flicker session. On the first platform (HP dc5750) it seems that the Broadcom TPM does not extend but only set the PCR17 value. We observed this behaviour for SLB lengths up to 40 bytes[2].

[1]Integrated on the network card

[2]The actual SLB is longer, but the Flicker kernel module was modified such that it sets a selected number as length value in the second word of the SLB

Although this does not comply with the specifications, the values for an SLB could be still precomputed for sealing purposes and as hash parameters to the Setup SLB. But, moreover, the PCR17 value is quite unstable if the SLB length is bigger. For an identical SLB, the value changes on every call to `SKINIT` if the length is more than approximately 160 bytes. Beneath this threshold, the value is more stable, but still invalid values occur in a sequence of subsequent Flicker sessions with only one SLB. The authors of Flicker confirmed this behaviour and suspect that this machine model contains a faulty Nouthbridge that causes this behaviour. We suspect that a potential reason for this error could be the deployment of the TPM on the network card, but could neither verify or disprove this assumption.

The Flicker authors, however, recommended the second machine (Dell T105) as a substitute that works correctly during their experiments. Unfortunately, Dell removed this machine from their European stores and it thus came to a very long delay in the delivery of this machine. Moreover, the delivered machine is a Quad Core platform, while the machine referenced by the Flicker authors is a Dual Core. Although, this should not have any impact on the DRTM operation according to the specifications and the Flicker infrastructure required only minor adaptions, e.g., halting three instead of one application processor, this machine also exhibited faulty behaviour.

At first, late-launch did not trigger a reset of the dynamic PCRs. We located the cause for this behaviour in the locality management of the TPM and could correct it[3]. Although the dynamic PCRs are now correctly reset, PCR17 is not extended with the measurement of the SLB, but with the measurement of an empty blob. The Flicker authors indicated that this faulty behaviour might be caused by incorrect Inter-Processor Interrupts in the kernel module. These interrupts are required to put the application processors into a state in which they can, on request, join the secure environment and are also mandatory part of the preparations for late-launch on a multi-core AMD platform. The original Flicker infrastructure implemented these necessary protocol already according to the specifications and even after a number of variations of this implementation we were unable to remedy this error. We further tested for other possible causes like the TPM driver and the state of the bootstrap processor, but were still unable to find the cause. It also has to be noted that debugging on this low abstraction level on a platform is more complex and time-consuming than the common application code debugging, especially due to easy crashes in the secure environment and required reboots of the system as well as the lack of advanced software debugging tools. Hardware

[3]To be precise: In contrast to the original Flicker version, the OS has now to relinquish locality 0 before `SKINIT` is executed

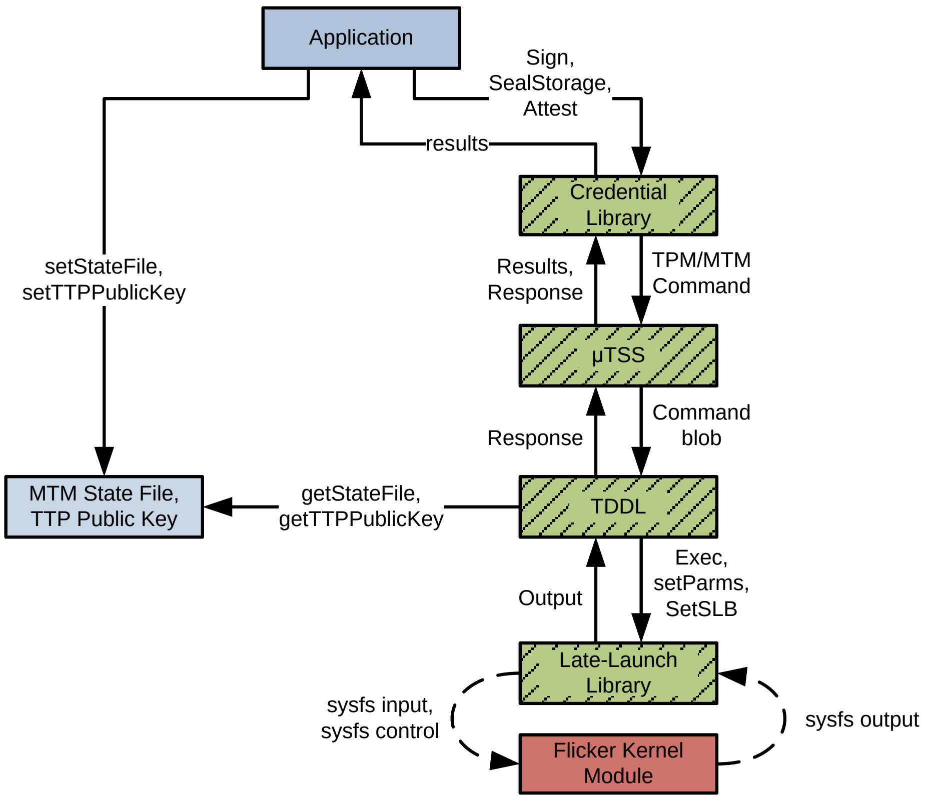

Figure 6.1: *Architecture software stack*

debugging tools, e.g., an LPC bus sniffer, were not available to us.

Furthermore, the availability of AMD based systems that are suitable for late-launch in terms of TPM version, chipset, and processor seems quite limited. To the best of our knowledge, the above mentioned models are the only off-the-shelf platforms currently available.

A reasonable alternative would be the implementation on an Intel based platform. However, although DRTM is identical on both vendors on a high level, they differ extremely when it comes to the implementation details. Thus, at this point in time in the progress of the thesis, such a switch was unfeasible considering the time required for the re-implementation.

6.2 Software Stack

Figure 6.1 illustrates the implemented software stack for applications that make use of their specific MTM for credential storage and usage. The shaded boxes on the right hand side (Credential Library, μTSS, TDDL, and Late-Launch Library) depict libraries that are linked into the application. Arrows indicate the input and output parameters for the library interfaces between two layers.

The Credential Library provides the application with the highest level of abstraction and small interface that is currently limited to the functionality to sign data, seal data, and attest the MTM state. The Credential Library implements these functionality based on the MTM services offered by the μTSS library.

The μTSS is compared to the TSS specified by the TCG extremely minimised and does not offer the abstraction level of the specified. It focuses on the Trust Service Provider layer of the TSS and solely implements the required MTM commands according to the specifications, i.e., it forms the command blobs that are sent to the MTM. Furthermore, it parses the MTM response blob and returns the required parameters to the Credential Library, which forwards them together with the result value to the application

The TDDL library implements the device driver functionality, thus the interface to sent an command blob to the MTM. It thereby forms plus loads the complete input for the MTM PAL, e.g., the selected MTM state file, TPP public key, and domain-specific date. It also determines and loads the MTM SLB that has to be loaded for the MTM command that shall be executed. To do so, it utilises the Late-Launch Library to communicate with the Flicker kernel module via the module's sysfs entries. Furthermore, it parses the output buffer of the kernel module, saves the updated MTM state and version information, and returns the MTM response to the μTSS for further parsing.

The motivation to implement the libraries layered like this is the provisioning of standardised interfaces in between the layers that provide more flexibility for changing the lower levels of the stack. For example, the DRTM based architecture could be substituted by an MTM implementation based on a hypervisor architecture or on secure environments like M-Shield and TrustZone.

6.3 Code Size

The main programming language used for the implementation is ANSI C99. The only part written in another language is the SLB core library, provided by Flicker, which consists of Assembly x86 code.

In the following we list the sizes in bytes as well as in lines of code of the different components of our architecture. To determine the LoC we use the open-source SLOCCount tool[4].

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 6.1: *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 [35], but adapted/extended)*

Table 6.1 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 software TCB for the MTM PAL is solely the SLB Core, which is minimal with 223 LoC. The rest of the components implement optional features for a PAL, but are necessary for the MTM PALs in our architecture. Thus, they form together with the SLB Core the TCB for the applications that utilise an MTM.

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

SLB	Description	Size (KB)
Setup	Bootstrap a TTP Trust Domain	37594
State Instantiation	Instantiates a new MTM state	39124
MTM 1	Implements command subset for RIM certificates, monotonic counter, and capabilities	45504
MTM 2	Implements command subset for session management and sealed storage	44820
MTM 3	Implements command subset for key management, signing, and attestation	46168

Table 6.2: *Sizes in kilobyte of the implemented SLBs*

Table 6.2 lists the sizes in kilobyte for each of the SLBs in our architecture. Considering, that in Flicker the highest 4kB of the 64kB of the SLB are reserved for the runtime stack and 12kB more used during the building of the page tables, then the PALs take full advantage of the provided space.

Layer	LoC
IMA[a]	1330
Flicker kernel module[b]	694
Late-Launch Library	444
TDDL	111
μTSS	1185
Credential Library	142

Table 6.3: *The implementation sizes in lines of code for the software stack ([a]provided by the kernel, but extended; [b]provided by Flicker, but modified/extended)*

In Table 6.3 the sizes in LoC of the software stack layers, as presented in Figure 6.1, are listed. Additionally, the modified IMA kernel subsystem and Flicker kernel module are measured. The μTSS forms the bulk of the stack, because it is responsible for the implementation of the MTM commands. The TDDL and the Credential Library are on the other hand rather pure APIs between their neighbouring layers. The modified Flicker kernel module is 66 LoC bigger than the original module, because it now implements the measuring of the application and because it was patched to run under a version 2.6.33 Linux kernel. The IMA merely grew 14 LoC in order to implement the necessary interface for the Flicker kernel module to enquire logged measurements.

Table 6.4 lists the sizes in bytes of the required input and output parameters in our architecture. The MTM state, although already minimised [35], forms

Parameter	**Size (Bytes)**
TTP Public Key	274
TTP Public Key Hash	20
Wrapped storage key	613
Sealed K_{Common}	322
State version information	16
MTM state	2030
TTP/Vendor signature	128
Hashes (for 3 MTM SLBs)	60
Application identity	20
CPU state	116

Table 6.4: *Sizes of the input and output parameters in the architecture*

the bulk of the I/O parameters. Both the wrapped storage key and the sealed K_{Common} are remarkably bigger than their actual content (RSA-2048 key pair and AES-128 key, respectively), due to additional integrity and usage information added by the TPM. The CPU state consists of a data structure that contains all necessary information of a CPU, e.g., register values, in order to dump and reconstruct the CPU state.

6.4 Performance

Despite problems with the DRTM measurement, it works sufficiently enough to estimate the performance with very high accuracy. For these tests, the prototype runs in insecure mode, that means that we accept the false measurements of DRTM stored in PCR17 as correct and use them in the context of our architecture. The only difference during the operation that affects the performance measurements is the size of the SLB that the TPM measures during late-launch. If the late-launch would measure correctly, the TPM would measure an SLB of one of the sizes listed in Table 6.2, while the faulty late-launch measure an SLB of length 0. Thus, the faulty implementation has a slight performance benefit.

Performance measurements of the faulty `SKINIT` instruction yield a mean execution time of 0.003 *ms* on our Dell machine[5]. This corresponds to the performance measurements conducted by the Flicker authors [73, Section 7] and confirms that the TPM indeed performs an SHA-1 hash of null bytes. The estimated performance benefit of the faulty execution, compared to the

[5]mean of 10 measurements

correct one, is approximately 133.35 ms, if we assume an SLB size of 48kB and a linear duration increase in Table 2 of [73].

In the following we provide the runtimes of the particular PALs and analyse how much each particular operations contributed to the overall runtime.

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

Table 6.5 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 Dell machine. Although the benchmark program avoids the complete TrouSerS stack and conducts I/O communication directly via the character device of the platform TPM, these measurements form the upper bound for the performance. However, since both the TPM driver in our architecture and in the OS kernel are extremely similar, these upper bounds are extremely close to the actual performance in the secure environment.

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

Table 6.6 lists the mean execution times and the corresponding standard deviation of the cryptographic primitives in the cryptographic library used by our architecture. The measurements were taken with the `gettimeofday`

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

command inside a benchmark program running on top of the OS. Therefore these measurements form the upper bound for the performance of the primitives executed in the secure environment. However, the upper bound is very close to the actual performance, because the system was kept idle and the only overhead was OS scheduling, which should have been rare on a quad core system. The cryptographic primitives execute on one CPU with 2.1GHz clock rate and perform therefore very fast and also stable. The only exceptions are, similar to the TPM commands, the functions involving private key operations.

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

Table 6.7: *Mean SLB runtimes and their standard deviation (10 measurements, excluding MTM command execution time)*

The mean runtimes and the corresponding standard deviations of the particular SLBs in our architecture are listed in Table 6.7. The measurements were computed based on the CPU clock rate and the number of elapsed clock cycles (`rdtsc` command). We excluded the MTM command execution time from the measurements for the MTM SLBs, because it greatly depends on the kind of command. The upper bound, however, is approximately the RSA-1024 private key generation execution time, which is required for the creation of a wrapped key by the software-based MTM.

	Setup		
Steps	$\bar{x}$ (ms)	σ (ms)	%
Derive $K_{Version}$	0.008	0.000	0.000
Derive counter auth	0.004	0.000	0.000
Encrypt state version info	0.021	0.000	0.000
Create TPM wrap key	22667.598	20390.260	71.621
Create reference counter	442.320	3.867	1.398
Load wrap key	2662.846	4.311	8.414
Seal K_{Common} (4 times)	1523.762	0.001	4.814
Flush loaded wrap key	39.350	0.000	0.124
Verify TTP signature	507.788	0.000	1.6

Table 6.8: *Mean performance, standard deviation, and share on overall mean runtime for each major step in the workflow of the Setup PAL (10 measurements)*

Tables 6.8, 6.9, and 6.10 list the mean execution times of the major steps in the workflow of the particular PALs. The measurements were computed based on the CPU clock rate and the number of elapsed clock cycles (`rdtsc` command). Additionally, the standard deviations of the performances and the share of each step on the overall mean execution time are provided. The MTM PAL measurements exclude the MTM command performances for the above stated reason.

In general, the commands that involve the platform TPM, especially private key operations or wrapped key generation, form the bulk of the runtimes. The derivation of $K_{Version}$ in the tables 6.9 and 6.10 includes also the loading of the wrap key, unsealing of K_{Common}, and flushing of the loaded key. Both the creation of a new virtual counter during the state initiation as well as the incrementation of a virtual counter both require one read from and one write to the TPM NVM. The MTM state initiation requires one RSA-1024 private key generation for the state's AIK.

6.5 Constraints

In this section we want to provide some information about constraints in terms of size and time in our architecture.

First of all, the size of the SLB is very limited with 64kB. Considering the original purpose of an SL, namely to prepare the execution environment for bootstrapping a VMM or secure OS, this limitation is not unanticipated. The

	MTM state initiation		
Step	$\bar{x}$ (ms)	σ (ms)	%
Initial PCR17 extension	19.239	0.000	0.326
Derive $K_{Version}$	3805.279	3.452	64.549
Derive counter auth	0.004	0.000	0.000
Decrypt state version info	0.020	0.001	0.000
Encrypt state version info	0.014	0.000	0.000
Verify virtual counter values	14.173	0.000	0.240
Create new virtual counter	139.506	0.000	2.367
Derive K_{State}	0.008	0.000	0.000
Initiate new MTM state	1741.708	1159.523	29.552
Encrypt state	0.101	0.002	0.002
Input/Output parameters PCR17 extension	153.961	0.000	2.612

Table 6.9: *Mean performance, standard deviation, and share on overall mean runtime for each major step in the workflow of the MTM state initiation (10 measurements)*

	MTM execution		
Step	$\bar{x}$ (ms)	σ (ms)	%
Initial PCR17 extension	19.239	0.000	0.456
Derive $K_{Version}$	3801.656	0.843	90.115
Derive counter auth	0.004	0.000	0.000
Decrypt state version info	0.021	0.000	0.000
Encrypt state version info	0.019	0.000	0.000
Verify virtual counter values	14.173	0.000	0.336
Read virtual counter	0.004	0.000	0.000
Derive K_{State}	0.007	0.000	0.000
Encrypt state	0.088	0.001	0.002
Decrypt state	0.108	0.002	0.003
Increment virtual counter	139.298	0.452	3.302
Input/Output parameters PCR17 extension	211.699	0.000	5.018

Table 6.10: *Mean performance, standard deviation, and share on overall mean runtime for each major step in the workflow of the MTM execution (10 measurements, excluding MTM command)*

limit is impaired by the fact, that the space can not be fully utilised, because the memory region between the end of the PAL and the 64kB offset is used as runtime stack, which grows towards the PAL region. The PAL developer has to take this into account in order to leave sufficient space for the stack so that it does not overwrite the PAL's code segment.

Another constraint is the size of the input and output parameters. In Flicker by default 4kB is allocated for each buffer. If we consider the sizes of the required inputs and outputs in our architecture (see Table 6.4), that leaves 411kB for the MTM command blob as input. As a consequence, in our architecture the I/O buffers were augmented to 5kB, thus leaving 1435kB for the MTM command. Although this size is much smaller than the specified 4kB size of the TPM I/O buffer, it is big enough to cater for all command and response blobs in our MTM implementation. The input and output buffer sizes could, however, be further increased if necessary.

The maximum number of Trust Domains on our Dell machine is limited to three. Although our architecture allocates only four bytes of NVM for each reference counter, the overhead induced by the TPM, e.g., information about read and write permissions or a bound platform state, depletes the scarce amount of available NVM.

The critical factor for the maximum runtime of a Flicker session are the disabled interrupts during the session. The SLB execution suspends the legacy OS without preparing the OS specifically for the suspension, thus unhandled interrupts may cause a system crash during resumption. For instance, on both our test platforms we experienced the behaviour that a Flicker session that lasted longer than approximately 8 seconds resulted in a system crash when the legacy OS resumes operation. This is a practical problem for PALs that require more time, for example the Setup PAL. However, this behaviour occurres only when the network cable is plugged in. If unplugged, no problems were experienced, even with sessions longer than 2 minutes. We suspect therefore the cause for the crash in unhandled interrupts from the network card.

7

Future Work

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

A first enhancement of the security would be to enable multi-factor authorisation. The current architecture is explicitly designed to provide the credential platform in an application-specific manner. This is motivated by the fact that personal platforms possess in most cases only one user but several applications with interest in securing their particular credentials. In this spirit, the current design excludes any user interaction. Nevertheless, user authorisation and physical presence can 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, e.g., on public computer platforms. The prerequisites for secure user I/O are a trusted channel and a trusted path between the user and the secure environment. First experiments with the Flicker architecture have shown that a polling PS/2 keyboard driver for the secure environment is possible. Further work in this direction would be improved driver support for the secure environment, e.g., a graphics driver based on memory mapped VGA, and especially means to establish a trusted channel and indicate a trusted path. An example for a recent research project is the Bumpy architecture of CMU [74], which utilises an external trusted monitor.

Although, our architecture provides the capabilities for a secure execution and storage of applications' credentials, an attacker is still able to misuse the credentials. In this thesis the TCB was defined as the hardware and software basis that is critical for the system security and we implicitly excluded the credential misuse from this definition. In order to provide stronger protection for credentials, the architecture also has to prevent their misuse. The two prevalent factors that enable an attacker to misuse the credentials are the involvement of OS components in the application identification and the problem of ensuring runtime integrity. The first factor concerns the IMA subsystem and the Flicker kernel module, which are responsible for measuring the application, but are part of the OS and thus prone to software attacks. The open question here is how these components could be removed from the architecture while still providing a trusted application measurement. Alternatively, these components could remain part of the architecture, but their integrity has to be guaranteed. An authenticated boot (SRTM) is one approach, but it suffers from the Time of Check to Time of Use problem, since its measurements are not necessarily valid after the boot process. The authors of Flicker describe in [73] briefly the design of a rootkit scanner implemented as a PAL. This scanner measures the kernel text segment, the loaded kernel modules, and the system call table. This measurement could also be part of our architecture, for example to bind the credentials to a specific kernel or to attest the kernel measurement at runtime and rely on the TTP to evaluate the trustworthiness of the kernel. However, using this measurement for binding purposes contradicts the meaningful attestation design goal of Flicker, because it inflates the measurement chain to almost all of the ill-defined, huge TCB that Flicker avoids. Other recent approaches to achieve runtime integrity have been listed in Section 3.2 on page 32.

A further open issue for the application identification relates to applications consisting of interpreted scripts. The kernel's process management lists the script interpreter, for instance, Bash shell or Python, as executables, but does not provide information about the script file that is interpreted. Since the script file obviously forms the application, it has to be part of the aggregated application measurement computed by the Flicker kernel module and IMA. Since we currently can not identify this file via the kernel process management, it is omitted from the application ID, which is hence incomplete.

In terms of applicability, the application identification ideally is based on some more flexible mechanism than a hash measurement. Currently, application and library measurements change when they are upgraded and thereby the application loses its bound credentials. Some form of secure migration of the credentials after an application/library update would be one option. Another option is property-based attestation as presented in contemporary research [86, 92, 65, 63].

A further practical optimisation relates to support for longer Flicker sessions. As described earlier, long sessions cause a system crash at legacy OS resumption, presumably because of unhandled interrupts. One possible solution would be to add interrupt handlers to the PAL, but this seems hardly feasible considering the limited space of SLBs and the complexity of the handler routines. A better solution would be to prepare the OS for the long session, e.g., suspend interrupt critical devices like the network card.

Open research regarding the HW support would be an improvement of the resources available to the secure environment. Bearing in mind the original purpose of DRTM, the bootstrap of a secure OS/VMM, the current resources in terms of space or DRTM execution environment are sufficient. However, with respect to the use cases enabled by the Flicker architecture, the available resources could be improved. For example on mobile devices, the processor security extensions like M-Shield or TrustZone provide secure ROM and RAM, a firewalled entry to the environment based on signatures, cryptographic accelerators, or a unique hardware key. Projects like On-board Credentials [64] already demonstrated the enhanced possibilities for credential architectures based on such security hardware.

Future optimisations and open issues regarding our MTM credential platform concern the secure, specification compliant instantiation of the MTM states and the reset of the MTM. Our architecture utilises a MRTM implementation, which does not include the concept of a (physically present) MTM owner and does not provide an EK. Thus, an AIK together with its credentials are assumed to be pre-installed by the manufacturer. But since we now instantiate the state on the user's platform, an AIK has to be securely provisioned to the new MTM. A possible solution would be to add a protocol that leverages the platform TPM to protect and deliver the AIK to the MTM PAL for the instantiation. Another solution would be to implement an EK and allow MTM owners, e.g., on an MLTM, but this would entail more MTM PALs with a bigger input buffer because of the increased state size and MTM code-base. A second issue concerns the MTM reset. The specifications define that certain volatile data has to be reset if the MTM platform reboots. For instance, the static PCRs are reset to zero and all loaded keys and sessions are cleared. In our implementation, the MTM preserves its state even over a system reboot due to the secure storage. The open questions here include, e.g., when to reset the MTM state: at application "reboot" or at system reboot, or how to reset the MTM states of several applications without actually involving the applications, e.g., by splitting the state in a resettable and non-resettable part.

Another issue specific to MTM and TPM is the long-term security of the utilised cryptographic primitives. For example, SHA-1 is not considered secure anymore. The essential problem is the inflexibility of the TPM and MTM specifications regarding these primitives. In our architecture, the benefit of implementing the MTM as software instead of as dedicated hardware, make it possible to upgrade the MTM to newer, stronger cryptographic algorithms. However, the TPM, which is fundamental for the security of our architecture, is inflexible in this matter and thus in the long-term constitutes a possible security risk.

8 Conclusion

In this thesis, we presented our solution for an architecture that provides application-specific credentials, deployed by means of TCG Dynamic Root of Trust for Measurement on an AMD based platform. The rationale to use DRTM was to approach the basic vulnerability for the security of applications' credentials, namely the ill-defined and huge Trusted Computing Base on commodity operating systems.

We employed the Flicker infrastructure by Carnegie Mellon University as the fundamental implementation of DRTM in our architecture. Flicker provides a hardware-isolated and measured execution environment and the possibility to easily resume the legacy OS. The essential problem in this setup is the statelessness of Flicker sessions, which implicates that the deployed credentials are neither bound to their particular application nor protected from misuse by other applications, such as malware.

Although we utilised our architecture in this thesis to implement a software-based TCG Mobile Trusted Module as the application-specific credential platform, it is generic enough to form a framework that provides the link between applications and the secure environment for different types of Flicker PALs. Trust Domains were introduced as a concept to form logical units consisting of multiple PALs.

Our architecture design solves the statelessness problem by binding to each application one specific MTM. The MTM can be leveraged for securely executing and storing the application's credentials in an environment that provides a minimised TCB and isolation from the legacy OS. We thereby utilise the Integrity Measurement Architecture subsystem and the process management of the kernel to identify applications.

In this context we explored related work regarding possible solutions for credential platforms or security architectures that minimise the TCB of applications.

We analysed hardware- and software-based attack scenarios against DRTM, our architecture, the operating system, and the applications. As result of our analysis we conclude that hardware-based attacks and software-based attacks against late-launch are prevented by the design of DRTM. Software-based attacks against our architecture are detected through remote attestation based on the measured execution of the MTM. The most vulnerable attack point is the system- and application-level software. Attacks that compromise the IMA subsystem, the process management in the kernel, or the Flicker kernel module may lead to a successful misuse of the application's credentials. However, the disclosure of credentials is not possible.

Possible solutions to alleviate the misuse problem, future optimisations, and open research issues were briefly discussed. We also provided size and performance figures of a prototype implementation and an analysis of encountered hardware problems in the context of our prototype.

In conclusion, we present a new approach to securely using and storing credentials. The design of our architecture is oriented towards the mobile device and personal device sectors in terms of motivation to bind credentials to applications rather than to users. Our approach to leverage a secure execution environment, provided by the platform's hardware, in a minimised manner also resonates well with the tight memory and energy constraints of mobile devices. An attacker in control of the highest privilege level is not able to disclose – but only misuse – credentials. This is in contrast to other approaches like hypervisor based architectures, where a privileged level attack provides full credential access. In comparison to the original Flicker architecture, this work mitigates credential misuse, since the attacker is forced to mount sophisticated software-based attacks against the kernel at run-time.

Bibliography

[1] grsecurity. `http://www.grsecurity.net/`.

[2] L4.verified - The Proof. `http://ertos.nicta.com.au/research/l4.verified/proof.pml`.

[3] Liberty Alliance. `http://www.projectliberty.org/`.

[4] Open Kernel Labs. `http://wiki.ok-labs.com/`.

[5] OpenID Foundation. `http://openid.net/`.

[6] Security-Enhanced Linux. `http://www.nsa.gov/research/selinux`.

[7] Shibboleth. `http://shibboleth.internet2.edu/`.

[8] TrouSerS - The open-source TCG Software Stack. `http://trousers.sourceforge.net/`.

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

[10] Windows CardSpace. `http://www.microsoft.com/windows/products/winfamily/cardspace/default.mspx`.

[11] *2010 IEEE Symposium on Security and Privacy (S&P 2010), 16-19 May 2010, Oakland, California, USA* (2010), IEEE Computer Society.

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

[13] Advanced Micro Devices. AMD64 Architecture Programmer's Manual Volume 2: System Programming, September 2007. Publication no. 24593, rev. 3.14.

[14] Anderson, M. J., Moffie, M., and Dalton, C. I. Towards trustworthy virtualisation environments: Xen library os security service infrastructure. Tech. rep., HP Laboratories Bristol, 2007.

[15] Andrews, J. Virtualization Security. `http://kerneltrap.org/OpenBSD/Virtualization_Security`, October 2007.

[16] anonhacker. Xbox 360 hypervisor privilege escalation vulnerability. `http://www.securiteam.com/securitynews/5MP040AKUA.html`, 2007.

[17] Apvrille, A., Gordon, D., Hallyn, S., Pourzandi, M., and Roy, V. DigSig: Runtime Authentication of Binaries at Kernel Level. In *LISA '04: Proceedings of the 18th USENIX conference on System administration* (Berkeley, CA, USA, 2004), USENIX Association, pp. 59–66.

[18] ARM. TrustZone-enabled processor. `http://www.arm.com/pdfs/DDI0301D_arm1176jzfs_r0p2_trm.pdf`.

[19] Azab, A. M., Ning, P., Sezer, E. C., and Zhang, X. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 461–470.

[20] Berger, S., Cáceres, R., Goldman, K. A., Perez, R., Sailer, R., and van Doorn, L. vTPM: virtualizing the trusted platform module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2006), USENIX Association.

[21] Biryukov, A., Dunkelman, O., Keller, N., Khovratovich, D., and Shamir, A. Key Recovery Attacks of Practical Complexity on AES Variants With Up To 10 Rounds. Cryptology ePrint Archive, Report 2009/374, 2009.

[22] Biryukov, A., and Khovratovich, D. Related-key Cryptanalysis of the Full AES-192 and AES-256. Cryptology ePrint Archive, Report 2009/317, 2009.

[23] Boneh, D., Cryptosystem, T. R., Rivest, I. R., Shamir, A., Adleman, L., and Rst, W. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the AMS 46* (1999), 203–213.

[24] Cannière, C. D., Mendel, F., and Rechberger, C. Collisions for 70-Step SHA-1: On the Full Cost of Collision Search. In *Selected Areas in Cryptography* (2007), C. M. Adams, A. Miri, and M. J. Wiener, Eds., vol. 4876 of *LNCS*, Springer, pp. 56–73.

[25] Cannière, C. D., and Rechberger, C. Preimages for Reduced SHA-0 and SHA-1. In *CRYPTO* (2008), D. Wagner, Ed., vol. 5157 of *LNCS*, Springer, pp. 179–202.

[26] Challener, D., Yoder, K., Catherman, R., Safford, D., and Van Doorn, L. *A Practical Guide to Trusted Computing.* IBM Press, 2007.

[27] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. K. 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, Mar. 2008).

[28] Davi, L., Sadeghi, A.-R., and Winandy, M. 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* (New York, NY, USA, 2009), ACM, pp. 49–54.

[29] Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Commun. ACM 9*, 3 (1966), 143–155.

[30] Dietrich, K. An Integrated Architecture for Trusted Computing for Java Enabled Embedded Devices. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing* (New York, NY, USA, 2007), ACM, pp. 2–6.

[31] Dolev, D., and Yao, A. C. On the security of public key protocols. Tech. rep., Stanford, CA, USA, 1981.

[32] Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., and Neugebauer, R. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2003).

[33] Dyer, J. G., Lindemann, M., Perez, R., Sailer, R., van Doorn, L., Smith, S. W., and Weingart, S. Building the IBM 4758 Secure Coprocessor. *Computer 34*, 10 (2001), 57–66.

[34] Eastlake 3rd, D., and Jones, P. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), Sept. 2001. Updated by RFC 4634.

[35] Ekberg, J.-E., and Bugiel, S. 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* (New York, NY, USA, 2009), ACM, pp. 9–18.

[36] Ekberg, J.-E., and Kylänpää, M. Mobile Trusted Module (MTM) - an introduction. Tech. rep., Nokia Research Center, 2007. NRC-TR-2007-015.

[37] Enck, W., McDaniel, P., and Jaeger, T. PinUP: Pinning User Files to Known Applications. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 55–64.

[38] England, P., Lampson, B., Manferdelli, J., Peinado, M., and Willman, B. A Trusted Open Platform. *Computer 36*, 7 (2003), 55–62.

[39] European Telecommunications Standard Institute. Specification of the Subscriber Identity Module - Mobile Equipment (SIM-ME) Interface, 1999. Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module - Mobile Equipment (SIM-ME) Interface (3GPP TS 11.11 version 8.14.0 Release 1999).

[40] Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., and Whiting, D. Improved Cryptanalysis of Rijndael. In *FSE '00: Proceedings of the 7th International Workshop on Fast Software Encryption* (London, UK, 2001), Springer-Verlag, pp. 213–230.

[41] Gajek, S., Löhr, H., Sadeghi, A.-R., and Winandy, M. TruWallet: Trustworthy and Migratable Wallet-Based Web Authentication. In *STC '09: Proceedings of the 2009 ACM workshop on Scalable trusted computing* (New York, NY, USA, 2009), ACM, pp. 19–28.

[42] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 193–206.

[43] GASSEND, B., SUH, G. E., CLARKE, D., VAN DIJK, M., AND DEVADAS, S. Caches and Hash Trees for Efficient Memory Integrity Verification. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2003), IEEE Computer Society, p. 295.

[44] GRAWROCK, D. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009.

[45] GUSTAFSON, D., JUST, M., AND NYSTROM, M. Securely Available Credentials (SACRED) - Credential Server Framework. RFC 3760 (Informational), Apr. 2004.

[46] HEATH, C. *Symbian OS Platform Security*. John Wiley & Sons, 2006.

[47] HERNICK, J. Virtualization Security Heats Up. `http://www.informationweek.com/story/showArticle.jhtml?articleID=201803212`, September 2007.

[48] HOGLUND, G., AND BUTLER, J. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[49] HOUSLEY, R. Cryptographic Message Syntax (CMS). RFC 5652 (Draft Standard), Sept. 2009.

[50] HWANG, J.-Y., SUH, S.-B., HEO, S.-K., PARK, C.-J., RYU, J.-M., PARK, S.-Y., AND KIM, C.-R. 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* (Jan. 2008), pp. 257–261.

[51] INFORMATION TECHNOLOGY LABORATORY (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY). The Keyed-Hash Message Authentication Code (HMAC), Mar. 2002.

[52] INSTRUMENTS, T. M-Shield Mobile Security Technology: making wireless secure (White Paper). `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`.

[53] INTEL CORPORATION. tboot, 2009. `http://tboot.sourceforge.net/`.

[54] INTEL CORPORATION. Trusted eXecution Technology (TXT) – Measured Launched Environment Developer's Guide, December 2009.

[55] Invisible Things Lab. BluePillProject. `http://bluepillproject.org/`.

[56] Joanna Rutkowska. Qubes OS Architecture, Jan. 2010. Invisible Things Lab; Version 0.3.

[57] Jonsson, J., and Kaliski, B. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), Feb. 2003.

[58] Kaliski, B. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), Sept. 2000.

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

[60] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an OS kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 207–220.

[61] Kleinjung, T., Aoki, K., Franke, J., Lenstra, A. K., Thomé, E., Gaudry, P., Montgomery, P. L., Osvik, D. A., Riele, H. T., Timofeev, A., and Zimmermann, P. Factorization of a 768-bit rsa modulus. Cryptology ePrint Archive, Report 2010/006, 2010.

[62] Kortchinsky, K. Cloudburst: Hacking 3D and Breaking Out of VMware, Aug. 2009. Black Hat USA; `http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-PAPER.pdf`.

[63] Korthaus, R., Sadeghi, A.-R., Stüble, C., and Zhan, J. A practical property-based bootstrap architecture. In *STC '09: Proceedings of the 2009 ACM workshop on Scalable trusted computing* (New York, NY, USA, 2009), ACM, pp. 29–38.

[64] Kostiainen, K., Ekberg, J.-E., Asokan, N., and Rantala, A. On-board credentials with open provisioning. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer,*

and Communications Security (New York, NY, USA, 2009), ACM, pp. 104–115.

[65] Kühn, U., Selhorst, M., and Stüble, C. Realizing property-based attestation and sealing with commonly available hard- and software. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing* (New York, NY, USA, 2007), ACM, pp. 50–57.

[66] Labs, R. PKCS#5 v2.1: Password-Based Cryptography Standard, Oct. 2006.

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

[68] Loscocco, P. A., Smalley, S. D., Muckelbauer, P. A., Taylor, R. C., Turner, S. J., and Farrell, J. F. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *In Proceedings of the 21st National Information Systems Security Conference* (1998), pp. 303–314.

[69] Loscocco, P. A., Wilson, P. W., Pendergrass, J. A., and McDonell, C. D. Linux kernel integrity measurement using contextual inspection. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing* (New York, NY, USA, 2007), ACM, pp. 21–29.

[70] Marchesini, J. *Shemp: secure hardware enhanced myproxy.* PhD thesis, Hanover, NH, USA, 2005. Chair-Smith, Sean W.

[71] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy* [11].

[72] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Seshadri, A. How low can you go? Recommendations for Hardware-Supported Minimal TCB Code Execution. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2008), ACM, pp. 14–25.

[73] McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An Execution Infrastructure for TCB Minimization. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), ACM, pp. 315–328.

[74] McCune, J. M., Perrig, A., and Reiter, M. K. Safe passage for passwords and other sensitive data. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)* (Feb. 2009).

[75] Microsoft MSDN Library. Mandatory Integrity Control. `http://msdn.microsoft.com/en-us/library/bb648648%28VS.85%29.aspx`.

[76] Mulliner, C. Exploiting Symbian: Symbian Exploitation and Shellcode Development. `http://www.mulliner.org/collin/academic/`.

[77] National Institute of Standards and Technology. FIPS 197, Announcing the ADVANCED ENCRYPTION STANDARD (AES), Federal Information Processing Standard (FIPS), Publication 197. Tech. rep., DEPARTMENT OF COMMERCE, Oct. 2001.

[78] National Institute of Standards and Technology. Recommendation for Block Cipher Modes of Operation: Methods and Techniques, NIST Special Publication 800-38A. Tech. rep., DEPARTMENT OF COMMERCE, 2001.

[79] National Institute of Standards and Technology. FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2. Tech. rep., DEPARTMENT OF COMMERCE, August 2002.

[80] National Institute of Standards and Technology. Recommendations for Key Derivation Using Pseudorandom Functions, NIST Special Publication 800-108. Tech. rep., DEPARTMENT OF COMMERCE, Oct. 2009.

[81] Novell. AppArmor Application Security for Linux. `http://www.novell.com/linux/security/apparmor/`.

[82] Novotny, J., Tuecke, S., and Welch, V. An Online Credential Repository for the Grid: MyProxy. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 2001), IEEE Computer Society, p. 104.

[83] NTT Data Corporation. TOMOYO Linux. http://tomoyo.sourceforge.jp/index.html.en.

[84] Ormandy, T. An empirical study into the security exposure to hosts of hostile virtualized environments, 2007.

[85] Paulson, L. C. The foundation of a generic theorem prover. *J. Autom. Reason. 5*, 3 (1989), 363–397.

[86] Poritz, J., Schunter, M., Herreweghen, E. V., and Waidner, M. Property Attestation - Scalable and Privacy-friendly Security Assessment of Peer Computers, May 2004. Research Report RZ3548, IBM Research.

[87] Ray, E., and Schultz, E. Virtualization security. In *CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research* (New York, NY, USA, 2009), ACM, pp. 1–5.

[88] Root Labs. How the PS3 hypervisor was hacked. http://rdist.root.org/2010/01/27/how-the-ps3-hypervisor-was-hacked/.

[89] Root Labs. TPM hardware attacks (part 2). http://rdist.root.org/2007/07/17/tpm-hardware-attacks-part-2/.

[90] RSA Security. RSA SecurID. http://www.rsa.com/node.aspx?id=1156.

[91] Sadeghi, A.-R., Selhorst, M., Stüble, C., Wachsmann, C., and Winandy, M. TCG Inside?: A Note on TPM Specification Compliance. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing* (New York, NY, USA, 2006), ACM, pp. 47–56.

[92] Sadeghi, A.-R., and Stüble, C. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *NSPW '04: Proceedings of the 2004 workshop on New security paradigms* (New York, NY, USA, 2004), ACM, pp. 67–77.

[93] Sadeghi, A.-R., Stüble, C., and Pohlmann, N. European Multilateral Secure Computing Base - Open Trusted Computing for You and Me, 2004. Whitepaper. www.emscb.org/content/pages/turaya.htm.

[94] Sahita, R., Warrier, U., and Dewan, P. Dynamic Software Application Protection, 2009. Intel Coporation.

[95] Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 16–16.

[96] Saltzer, J. H., and Schroeder, M. D. The Protection of Information in Computer Systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[97] Sarmenta, L. F. G., van Dijk, M., O'Donnell, C. W., Rhodes, J., and Devadas, S. 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* (New York, NY, USA, 2006), ACM, pp. 27–42.

[98] Schaufler, C. SMACK: Simplified Mandatory Access Control Kernel for Linux. `http://schaufler-ca.com/`.

[99] Schmidt, A. U., Kuntze, N., and Kasper, M. On the deployment of Mobile Trusted Modules, 2007.

[100] Seshadri, A., Luk, M., Qu, N., and Perrig, A. 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* (New York, NY, USA, 2007), ACM, pp. 335–350.

[101] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. 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* (New York, NY, USA, 2005), ACM, pp. 1–16.

[102] Singaravelu, L., Pu, C., Härtig, H., and Helmuth, C. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. *SIGOPS Oper. Syst. Rev. 40*, 4 (2006), 161–174.

[103] Smith, J. M., Smith, S. W., and Zhao, M. Keyjacking: Risks of the Current Client-side Infrastructure. In *In 2nd Annual PKI Resarch Workshop. NIST* (2003).

[104] SRAGE, J., AND AZEMA, J. M-Shield Mobile Security Technology, 2005. TI Whitepaper. `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`.

[105] STEINBERG, U., AND KAUER, B. Nova: a microhypervisor-based secure virtualization architecture. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), ACM, pp. 209–222.

[106] SUN MICROSYSTEMS. Java Card Specifications Version 3.0.1, May 2009. `http://java.sun.com/javacard/`.

[107] SUNDARESAN, H. OMAP platform security features, July 2003. TI White paper. `http://focus.ti.com/pdfs/vf/wireless/platformsecuritywp.pdf`.

[108] SYMBIAN SOFTWARE LTD. Symbian Developper Library v9.4.

[109] TA-MIN, R., LITTY, L., AND LIE, D. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 279–292.

[110] TARNOVSKY, C. Deconstructing a 'Secure' Processor, Feb. 2010. Black Hat DC; `http://www.blackhat.com/presentations/bh-dc-10/Tarnovsky_Chris/BlackHat-DC-2010-Tarnovsky-DASP-slides.pdf`.

[111] TERESHKIN, A., AND WOJTCZUK, R. Introducing Ring-3 Rootkits, July 2009. Black hat USA; `http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf`.

[112] TRUSTED COMPUTING GROUP. Mobile Trusted Module (MTM) Specification. Version 1.0 Revision 6, 26 June 2008.

[113] TRUSTED COMPUTING GROUP. TCG Mobile Reference Architecture Specification. Version 1.0 Revision 1, 12 June 2007.

[114] TRUSTED COMPUTING GROUP. TCG PC Client Specific Implementation Specification For Conventional BIOS. Version 1.20 FINAL, Revision 1.00, July 13, 2005, For TPM Family 1.2; Level 2.

[115] Trusted Computing Group. TCG PC Client Specific TPM Interface Specification (TIS). Version 1.2 FINAL, Revision 1.00, July 11, 2005, For TPM Family 1.2; Level 2.

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

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

[118] van Dijk, M., Rhodes, J., Sarmenta, L. F. G., and Devadas, S. Offline Untrusted Storage with Immediate Detection of Forking and Replay Attacks. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing* (New York, NY, USA, 2007), ACM, pp. 41–48.

[119] van Doorn, L., Doorn, V., Ballintijn, G., and Arbaugh, W. A. Signed Executables for Linux. Tech. rep., UMD, June 2001. CS-TR-4259.

[120] Wang, X., Yin, Y. L., and Yu, H. Finding Collisions in the Full SHA-1. In *CRYPTO* (2005), V. Shoup, Ed., vol. 3621 of *LNCS*, Springer, pp. 17–36.

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

[122] Wojtczuk, R. Subverting the Xen Hypervisor, Aug. 2008. Black Hat USA; `http://www.invisiblethingslab.com/bh08/papers/part1-subverting_xen.pdf`.

[123] Wojtczuk, R., and Rutkowska, J. Attacking Intel Trusted Execution Technology, Feb. 2009. Black Hat DC; `http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf`.

[124] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 17–31.

[125] YANG, J., AND SHIN, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2008), ACM, pp. 71–80.

[126] ZHANG, X., ACIIÇMEZ, O., AND SEIFERT, J.-P. A trusted mobile phone reference architecture via secure kernel. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing* (New York, NY, USA, 2007), ACM, pp. 7–14.