On Attestable and Trusted Services for the Cloud

by

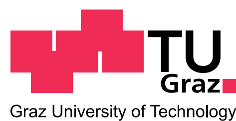
Martin Pirker

A PhD Thesis
Presented to the Faculty of Computer Science and Biomedical Engineering,
Graz University of Technology (Austria),
in Partial Fulfillment of the Requirements for the PhD Degree

Assessors

Prof. Dr. Roderick Bloem (Graz University of Technology, Austria)
Prof. Dr. Allan Tomlinson (Royal Holloway, University of London, UK)

January 2015



Institute for Applied Information Processing and Communications (IAIK)
Faculty of Computer Science and Biomedical Engineering
Graz University of Technology, Austria

Only oh-so-many grand projects fit into one man's mortal life.

This is one of them.

Abstract

The digital devices that have become part of our daily lives constantly evolve, and with them the tasks we use them for. Today, devices such as smartphones and tablets boast a processing power that was until recently only available on a desktop PC. Furthermore, these new devices connect to the internet via inexpensive wireless data connections, and perform countless everyday tasks—sometimes even with the support of remote server services in the cloud. In this thesis, we study an assortment of security and privacy challenges connected with this distributed data processing scenario.

First, we study the challenge of determining and reporting a platform's state. This query is motivated by the assumption that if a platform is in a certain known state, it may be trusted to behave in a certain known way. We develop prototype platforms that integrate two Trusted Computing techniques, namely the Trusted Platform Module (TPM) and Intel's Trusted Execution Technology (TXT). On our Linux-based platform the platform administrator can stipulate that applications run only in a certain platform state, otherwise they remain inaccessible in an encrypted form. Furthermore, using Android as a base platform, we proceed to construct a more lightweight platform. This Android-based platform may join a cloud network as a processing node, however only if the platform is in a certain state. For this purpose, we develop a custom join protocol anchored in Trusted Computing.

Secondly, we explore the challenge of reducing the complexity of a platform and service. We design as prototype and implement a compact PrivacyCA, a service that issues certificates for Trusted Computing keys. These keys sign platform state reports in Trusted Computing.

Thirdly, we ensure that a client using a remote cloud service is able to pay for it in a privacy-preserving way. To this end, we adopt a scheme which enables privacy in an anonymous and unlinkable way, and evaluate its performance on smartphone class platforms. Our results show that such a scheme runs sufficiently fast on existing hardware platforms.

Fourthly, the v2 generation of TPM chips is now becoming more widely available, and it would be helpful to have a TPM v2 simulator for this new generation available. We demonstrate semi-automatic processing of the TPM v2 specification with the aim of extracting and generation of a full TPM v2 simulator that may run just as software, or become, ported to an FPGA platform, a basis for simulating a hardware TPM chip.

Acknowledgements

I'm deeply grateful to IAIK for providing the research environment and resources that enabled me to perform the work presented in this thesis. A heartwarming "Thank you!" to all collaborators over the years. This especially applies to all my colleagues in the Secure and Correct Systems (SCoS) research group at IAIK, but also to all partners and collaborators in all the individual research projects.

I also want to thank my PhD supervisor, Roderick Bloem. His feedback, patience, expertise and insights were an invalueable contribution to my academic growth and this thesis. Also, I want to thank Allan Tomlinson for being the external assessor of this thesis, and for every insightful discussion every time we met at a Trusted Computing event, somewhere in Europe.

Martin Pirker
Graz, January 2015

Table of Contents

Abstract **v**

Acknowledgements **vii**

List of Tables **xiii**

List of Figures **xv**

Glossary **xvii**

1 Introduction **1**
1.1 Introduction and Motivation 1
1.2 Obstacles and Challenges 3
1.3 Contribution 4
1.4 Overview 10

2 Background **11**
2.1 Trust 11
2.2 TCG's Trusted Computing 14
2.2.1 Introduction 14
2.2.2 Trusted Platform Module 15
2.2.3 TCG Software Stack 17
2.3 Platform State 18
2.3.1 Introduction 18
2.3.2 The PC Platform 19
2.3.3 Chain of Trust 19
2.3.4 Measurements and Complexity 20
2.3.5 Virtualization for Measurements Reduction 22
2.4 Dynamic Root of Trust 23
2.4.1 Intel's Trusted Execution Technology 23
2.4.2 Intel's tboot 24
2.5 A Support PKI for Trusted Computing 26
2.5.1 Credentials 26
2.5.2 Attestation Identity Keys (AIK) and PrivacyCAs 27
2.6 Aspects of Trusted Platforms 30
2.6.1 Security of Trusted Computing Technology 30
2.6.2 Input/Output as Attack Surface 32

2.6.3 Practical Chain of Trust . . . 32
2.6.4 Integrity of Code and Data . . . 32
2.6.5 Open Source . . . 33
2.6.6 Boundary of Security Breaches . . . 33
2.6.7 Software Management Usability . . . 34
2.7 Linux Platform Technology . . . 34
2.7.1 Logical Volume Manager . . . 34
2.7.2 Linux Unified Key Setup Encryption . . . 35
2.8 ARM Platforms . . . 35
2.9 An Anonymous Accounting Scheme . . . 37
2.10 Related Work . . . 38

3 A Platform with Enforcement of Application Integrity 45
3.1 Introduction . . . 45
3.2 Architecture . . . 46
3.2.1 Secure Boot . . . 47
3.2.2 Base System . . . 48
3.2.3 Trusted Virtual Application Manager . . . 48
3.2.4 Virtualization Partitions . . . 49
3.3 Operating the Platform . . . 49
3.3.1 Installation . . . 49
3.3.2 Update and Application Mode . . . 50
3.3.3 Base System Rebuild and Resealing . . . 50
3.3.4 Small Updates . . . 50
3.3.5 Application Management . . . 51
3.3.6 External Reboot . . . 51
3.4 Implementation . . . 51
3.4.1 Components . . . 51
3.4.2 Disk Storage Management . . . 53
3.4.3 Performance . . . 57
3.4.4 Shared TPM Access . . . 57
3.5 Platform Security . . . 57
3.5.1 Attacker Model . . . 57
3.5.2 Discussion . . . 58
3.6 Summary . . . 58

4 Lightweight Distributed Attested Clouds 61
4.1 Introduction . . . 61
4.2 Scenario . . . 62
4.2.1 Cloud Node Properties . . . 62
4.2.2 Entities . . . 63
4.3 Node Operation . . . 64
4.3.1 Cloud Node Joining Cloud Control . . . 64
4.3.2 Node Update / Cloud Rejoin . . . 67
4.3.3 Node Local Storage . . . 67
4.4 Implementation . . . 68

4.4.1 Overview . . . 68
4.4.2 x86 PC Platform . . . 69
4.4.3 ARM Platform . . . 71
4.4.4 Summary . . . 74
4.5 Aspects of Platform Security . . . 74
4.6 Summary . . . 75

5 A PrivacyCA for Anonymity and Trust 77
5.1 Introduction . . . 77
5.2 Building a Trustworthy Service . . . 78
5.3 First Iteration and Building Blocks . . . 79
5.4 Second Iteration: A Compact PrivacyCA Protocol and Service . 82
5.4.1 Communication Protocol . . . 82
5.4.2 Reduced Software Image . . . 84
5.5 Use Cases . . . 86
5.6 Summary . . . 88

6 Privacy-Preserving Payment for Constrained Clients 89
6.1 Introduction . . . 89
6.2 Scenario and High Level Description . . . 91
6.2.1 Entities . . . 91
6.2.2 Scenario . . . 91
6.2.3 High Level Description of Operations . . . 94
6.2.4 Privacy Considerations . . . 95
6.3 Practical Anonymous Payment . . . 97
6.4 Implementation . . . 99
6.4.1 Setup . . . 99
6.4.2 Results . . . 100
6.4.3 Implementation Discussion . . . 102
6.5 Discussion . . . 103
6.6 Summary . . . 104

7 Semi-Automated Prototyping of a TPM v2 107
7.1 Introduction . . . 107
7.2 From TPM Specification to Implementation . . . 108
7.3 Assembly of a TPM v2 Software Simulator . . . 110
7.3.1 Input Data Transformation . . . 110
7.3.2 Information Extraction . . . 111
7.3.3 Advanced Code Generation and Additional Steps . . . 113
7.4 Hardware TPM v2 Simulation Platform . . . 114
7.4.1 Introduction . . . 114
7.4.2 Implementation . . . 116
7.4.3 Prototyping Results . . . 117
7.5 Conclusion and Outlook . . . 118

8 Conclusion and Outlook **119**
8.1 Review of Challenges and Achievements 119
8.2 Future Work 121

A Publications and Cooperations **123**

Bibliography **129**

List of Tables

5.1 PrivacyCA prototype component sizes 87
6.1 Execution time of Activate and Spend 101

List of Figures

2.1 PC chain of Trust 20
2.2 Stored measurements log 21
2.3 Booting Linux with tboot. 25
2.4 AIK certificates, PrivacyCA and requester interaction 29

3.1 Overview of major platform blocks and execution flow 48
3.2 Platform prototype runtime architecture 53
3.3 TVAM commands overview 54
3.4 Detailed platform disk layout 55
3.5 Concept of main platform file system 55

4.1 Node-to-Cloud join protocol 65
4.2 Experimental hardware setup 72
4.3 Block diagram of TPM attachment. 72

5.1 XKMS-based certificate query 81
5.2 Formal specification example of PrivacyCA command 83
5.3 PrivacyCA client commands list 84
5.4 PrivacyCA implementation software layers 85

6.1 Cloud credits accounting flow overview 92
6.2 Scenario overview 92
6.3 High-level operations of credit flows 94
6.4 Spend execution time 102

7.1 FODG intermediate format 112
7.2 YAML intermediate format 112
7.3 Development setup with TPM hardware simulator. 115
7.4 Prototype hardware setup 117

Glossary

ACM Authenticated Code Module (Chapter 2.4.1), 24, 25, 52, 54, 58, 59, 70

AIK Attestation Identity Key (Chapter 2.5.2). 8, 27–30, 41, 64, 66, 68, 69, 77, 80–83, 86, 88, 126

DRTM Dynamic Root of Trust for Measurement (Chapter 2.4). 23, 24, 36, 39

EK Endorsement Key (Chapter 2.5.1). 26–28, 31, 64, 66, 69, 77, 80, 83, 88, 126

FPGA Field Programmable Gate Array. 71, 108, 115–117, 121, 126

LCP Launch Control Policy (Chapter 2.4.1). 24, 25, 50, 53

LPC Low-Pin-Count bus. 71, 72, 114–117

LUKS Linux Unified Key Setup (Chapter 2.7.2). 35, 54

LV Logical Volume. See LVM. 34, 54

LVM Logical Volume Manager (Chapter 2.7.1). 34, 54

MLE Measured Launch Environment (Chapter 2.4.1). 24, 25, 48, 50, 52, 53, 58

PCR Platform Configuration Register (Chapter 2.3). 16, 17, 19, 21, 22, 26, 27, 29, 32, 39, 50, 52, 53, 64, 66, 70, 73, 114

PKI Public Key Infrastructure (Chapter 2.5). 26, 80, 82

SINIT SINIT is the name of the ACM provided by Intel. 52–54, 70

SRTM Static Root of Trust for Measurement (Chapter 2.3.2). 23, 73

TCG Trusted Computing Group (Chapter 2.2). 2, 4, 8, 9, 15–18, 22, 26–28, 35, 41, 43, 57, 64, 78, 80, 107–110

TEE Trusted Execution Environment (Chapter 2.8). 36, 37, 121

TPM Trusted Platform Module (Chapter 2.2.2). v, 2, 4, 5, 7–10, 15–17, 19, 21–23, 25–28, 30–33, 35, 36, 38–41, 43, 45–47, 50, 52, 54, 57–59, 64–71, 73–75, 77, 81, 83–87, 107–118, 120–122, 125, 126

TSS TCG Software Stack (Chapter 2.2.3). 17, 18, 64, 69, 74, 85, 109, 114, 115, 122

TVAM Trusted Virtual Application Manager (Chapter 3.4.1). 39, 48–54, 56, 59, 124

TXT Intel Trusted Execution Technology (Chapter 2.4.1). v, 2, 5, 7, 19, 23–25, 31, 36, 38–40, 45–48, 50–52, 57–59, 69–71, 73, 74, 93, 103, 120

VLP Verified Launch Policy (Chapter 2.4.2). 25, 50, 53

1 Introduction

1.1 Introduction and Motivation

Digital devices such as smartphones and tablets are now a part of our daily life, and we have come to rely on their ability to use online internet services. Proportional to the upgrading of the underlying technology the capabilities of these devices are becoming ever more powerful. These enhancements also impact the tasks we use these devices for in our daily life. As an extension of this, we tend to entrust them increasingly with our personal data, which gives rise to the question of security and privacy of (private) data in the global internet of devices—covering client devices as well as servers.

Looking back a few years, the concept of a computer in every home at first appeared ridiculous at the time, but then it started to become affordable, and then in a very short time the personal home computer became both widely spread and well accepted. Sustained advances in microprocessor technology mean that today's mobile phones are rightly known as "smartphones", and these now boast the processing power of a desktop PC not many years ago. The result is that nowadays a smartphone is quite capable of performing many of the everyday tasks confronting end-users that previously required a full-blown desktop PC. If users wish to look something up on the internet, use an electronic messaging platform, or participate in social networks, all these tasks can nowadays be accomplished on any smartphone.

Due to size constraints a smartphone's capabilities are limited by nature in terms of battery life and storage size, and consequently not all tasks can be achieved on such a device alone. If a task is too demanding for a smartphone, parts of it or even the whole task need to be executed on a more powerful

platform. One approach suiting itself to remote data processing is the cloud computing option, which involves large data centres taking advantage of the economics of scale and leasing computing power and storage capacity to clients on demand. This combination of limited client devices (such as smartphones and tablets) and the availability of cheap wireless network connections and remote processing power readily available via cloud computing gives rise to a distributed processing scenario: Relatively minor tasks are executed directly on limited client devices, however extensive, more complex and resource consuming task are delegated to powerful remote cloud computing services. The data is externally processed, the results are sent back to the client and the computing resources consumed are billed to the client.

Since the use of this distributed data processing scenario is on the rise, the question of data security and privacy on local and remote platforms is of concern to all users. Increasingly, users own a smartphone, tablet, or similar device to access all kinds of internet services, and more and more businesses are leveraging the cloud for data processing by off-loading tasks to a remote cloud service. Specific security solutions[1] may be applicable in limited business scenarios, however solutions providing enhanced security for mass-market platforms and scenarios tend to have immediate appeal, as more platforms and users are positioned to take advantage of them.

One approach in improvement of security for mass-market platforms is Trusted Computing, which was developed and specified by the Trusted Computing Group (TCG) [128] industry consortium. At the core of Trusted Computing is the Trusted Platform Module (TPM) [140, 144], which is a dedicated hardware chip providing a set of functions for cryptography and cryptographic key management and additional primitives that support the implementation of higher-level security functions such as the reporting of a platform's state, so-called (remote) platform state attestation. The TPM chip has already sold in the hundreds of millions[2], however the application areas where a TPM is used are still limited. Furthermore, there are even newer technologies like Intel's Trusted Execution Technology (TXT) [46] that promise to improve the assessment of a platform's security state with dedicated cooperation from the platform CPU, chipset and integrated TPM.

The mass-market availability of such recent advances in security technology, as they are now being integrated into platforms and devices, inspires a closer look, which includes prototypical applications and experimentation therewith in the distributed processing scenario—small and mobile clients and remote (cloud) data processing.

[1] For example expensive, dedicated high-security modules (HSMs) on servers.

[2] The last official statement on the TCG homepage dates from as early as 2011 [135]. In private conversations with TCG members the estimate is that now over one billion TPMs have been shipped and integrated into a variety of platforms.

1.2 Obstacles and Challenges

There are numerous obstacles and challenges to be overcome in the distributed processing scenario. The work presented in this thesis does not, and indeed is not intended to, tackle all of them. Instead, we focus on a selection of platform security and user data privacy challenges posed by the distributed processing scenario:

- As computation tasks are delegated to remote cloud nodes and servers, is it possible to determine what is being executed on the remote platform? It would be good to ensure, by running some form of attestation process and protocol—that the intended software version also set up with the correct configuration is working with the data. This question concerns both operating system software layer and higher-level application software (image, possibly provided by a client).

  Furthermore, on the assumption there is a working attestation process to analyse the software installed on a platform (see above), how complex is the reporting in this process? Is the state report so complex that this process requires expert knowledge and special tools, or can this be simplified to the comparison of a value operation or question whether this state is good, and all others are bad—that an administator or common end-users can act upon or manage?

- The internet is a network connected across the world, so while on one side of the world capacity is being used, on the other, where it is night, processing capacity is idle. Given, then, that certain platforms are idle for a number of hours every day, it would surely be a beneficial idea to put idle capacity to good use. To quote a practical example, in the case of an idle internet-connected PC, what effort is necessary to turn it into a temporary, attestable (meaning "known") platform state processing node in a data processing network?

- Remote network services do not necessarily have to be complex, but should be reliable and attestable as to their state and functions provided. A network service required by entities interacting on a network may only be performing a simple processing operation, however the most important property of the service may be that it can also perform an attestation protocol on its own state. This protocol produces a proof that the service provided is exactly as expected by the clients. What is a prototype example of such a minimal, but sorely-needed security service?

- Although costs of cloud data processing continue to fall, services consumed in the cloud do still attract charges. The simplest approach to performing the accounting is through a processing cost account with a cloud provider, which lies within a client account and is in turn tied to the client's credit-card, to which all costs for resources consumed are billed. This basic setup leads to the natural assumption that, whenever costs need to be paid, a

real person or company (in effect the credit-card holder) must form part of this accounting process. As a result of this, paid cloud services cannot be bought anonymously.

It is interesting to reflect on this accounting problem and ask whether the default "no anonymous clients" scenario is really the only practical one. A cloud provider does not and cannot provide services free of charge, but if there is a way to pay anonymously for these services, this feature becomes the enabler for cases where the actual identity of the client does not matter. This leads to the conclusion that, if a practical, anonymous accounting and payment scheme for cloud resource consumption existed, this would expand the potential uses of cloud services to scenarios where client anonymity is important—which translates to increased business opportunities for a cloud provider.

- As the "v1.2" TPM chip generation currently available on the market has not enjoyed widespread adoption, a new "v2" TPM generation has been prepared by the Trusted Computing Group for mass-market release. Learning from the previous generation, the TCG implemented a set of new primitives to support new functions and deployment scenarios[3], which are impossible or very hard to implement with the current generation. Unfortunately, there currently exists only a TPM v2 specification, but so far no official public TPM v2 emulator package or programming language (e.g., C or Java) bindings.

  A software TPM v2 emulator would be a great aid to exploration of the new functions and experimentation. This need applies to practical use cases on the client and server side of the distributed processing scenario. Furthermore, actual practical use and application of the specification probably identifies weaknesses and defects early and thus promises to produce significant feedback for the TPM v2 specification process.

1.3 Contribution

The obstacles and challenges outlined in Section 1.2 are a high-level introduction to and reflection on several specific challenges. We tackle each of these challenges in a separate chapter of this thesis, by drawing on and utilising technological advances and approaches from various areas. This results in novel insights and several experimental proof-of-concept prototypes. As a whole, the advances described in this thesis for these individual challenges contribute to the implementation and practical deployment of the distributed processing scenario[4].

[3]For example, the TPM v2 supports the need for distinct cryptographic key hierarchies for platform, operating system and application use. The current TPM generation (v1.2) has only one key hierarchy, in which all keys are anchored.

[4]Naturally, the stream of problems never stops as the distributed processing scenario and internet devices continue to develop rapidly—more thesis challenges...

This thesis is the culmination of several years of research work, the main interim results of which have previously been presented in papers at peer-reviewed international conferences and workshop meetings and published in those proceedings. A full, detailed list referencing the previous works this thesis is based on can be found in Appendix A of this thesis, and indeed some passages are included verbatim.

The challenges, solutions, approaches and prototypes presented in the chapters of this thesis are as follows:

- The work presented in **Chapter 3** explores the problem of defined platform state in the scenario of a modern, general-purpose operating system on a PC platform. More specifically, this comprises the problem of determination of the current state of a platform, how it reached that state, and reporting of the state to external parties (in a so-called remote attestation process) wishing to form an opinion of the state of the platform or remote cloud server.

  We study this problem with a full-size off-the-shelf operating system called Debian Linux. We run it on a mass-market PC platform with integrated Trusted Platform Module (TPM) [140] and Intel Trusted Execution Technology (Intel TXT) [46] support. These two technologies provide the necessary hardware support functions to allow us to integrate platform state measurement, code execution logging and reporting into the Debian Linux operating system and runtime environment.

  Also, we modify[5] Debian such that an administrator defines the boot process and platform start-up state. The platform administrator is able to require that the main Linux system becomes accessible only when a certain configuration boots on a specific hardware platform. Furthermore, the administrator can still install any Linux (Debian) software package and libraries desired onto the platform. With the help of automated scripts the platform can rebuild itself and the new configuration defined by the administrator becomes the new one upon next reboot.

  The core contribution of the prototype developed in Chapter 3 is the combination of state-of-the-art advances in mass-market PC platform security technology (TPM, TXT) with a common general-purpose desktop OS in the form of Debian Linux.

  The challenges are, first, enabling a system administrator to lock down the platform, so that only when a specific configuration boots does the platform and its data become accessible. Secondly, with the integration of Trusted Computing ensuring a remote query and reporting of the platform state can be performed—remote platform state attestation[6]. Thirdly,

[5] As Debian Linux is fully open-source, it is a perfect choice for experimental modifications of any component.

[6] See Sections 2.2.2 and 2.3 for more details.

the prototype platform presented still allows to install any Linux software package without restrictions, pending approval from the administrator. Fourthly, the platform can rebuild itself at runtime for a new administrator defined state that becomes the new default state at next reboot. To our knowledge, this combination of features has not been demonstrated in such a full-size integrated prototype before[7].

The work of Chapter 3 produces a working prototype, however at significant effort. As a general purpose operating system Linux is quite complex, and as a result a large number of components are involved in the boot process or part of the runtime environment. Consequently, this results in numerous modifications to Debian Linux and requires the development of custom code or scripts to achieve the working prototype of Chapter 3.

Building on the above-mentioned effort invested in Chapter 3, we focus the work of the following two chapters on the reduction of this effort, and consequently of prototype complexity, although naturally this implies less functions and features are either implemented or available.

The underlying insight is simple: Some problems and services profit from a platform that can report its current state, but do not necessarily require a general purpose platform. Reduction of platform features, service functions and consequently implementation complexity, however, provides benefits in multiple ways: First, there is a smaller amount of implementation effort. Secondly, fewer components and less complexity ease understanding of the platform, as the working memory capacity of the human brain is limited. This contributes to the third benefit: detection and location of bugs and reflection on security features and limitations of the platform become more efficient.

- The work of **Chapter 4** builds on the challenge of Chapter 3 to construct a platform which starts in a defined platform state, which can then be reported to a third party. However, the work of Chapter 4 is based in place of Linux on the Android operating system platform and runtime environment.

  Android's origin is in the smartphone and tablet market where platform resources are scare. In the case of a typical Android devices mass storage is provided through flash memory, which is constricted by a limit on read/write cycles, before it burns out. Consequently, Android separates read-only storage (code) and temporary storage (application data), in order to manage storage wear according to expected usage patterns[8]. This separation makes implementation of required support for platform state measuring and reporting to Android easier. Overall, the result is fewer

[7]Feedback on a live demonstration of the prototype platform to representatives of major commercial platform vendors in 2011 reinforces our belief.

[8]This does not mean system code can never be changed. Of course the Android core platform components can be upgraded to a newer version, but the fact that this is done rarely motivates explicit separation from frequently changing data.

modifications on Android compared to the number of modifications required for a full-size Linux—as in Chapter 3.

The first contribution of Chapter 4 is the proof-of-concept modification of Android_x86 [3]. The Android prototype with Trusted Computing modifications starts in a certain state and is then able to produce a report on the state it has booted into through use of the same TPM and TXT technology as used for the platform in Chapter 3. Our Android-based prototype is a stand-alone software image, which is capable of booting from a simple USB stick and does not depend on main platform mass storage.

In contrast to Chapter 3 there is no installation process onto a specific hardware, the USB stick can boot on any TXT-enabled PC platform[9]. Consequently, this property allows very fast (re)boot of any unused PC (plug-in USB stick and reboot) into this Android-based platform. Thus, as a second contribution we develop a concept on how such an attested processing node can become part of something bigger, part of a cloud of processing nodes.

We implement the attestation and reporting components in the platforms of Chapters 3 and 4 by taking advantage of platform support for Trusted Computing technology. More specifically, the Trusted Platform Module (TPM) (Section 2.2.2) uses standard RSA keys for cryptography, for signatures on data blocks etc. Nowadays the cryptography of RSA keys is a well-known domain. However, the RSA keys of a TPM possess additional descriptive attributes[10] that describe their specific properties and intended usage in Trusted Computing scenarios. One of these attributes specifies that a key is capable of performing a signature for a platform's state report (Section 2.5.2). Naturally, to preserve and document this specific attribute of a key, certificates provide a way to bind the key to its supplemental attributes. This demands a specific certification process, so that the key's signature on a platform's state report can be verified and reconstructed at a later time to have been done with the correct key with the correct attributes.

Unfortunately, traditional certification authorities (CAs) do not know about TPM keys and their special attributes, but in order to issue certificates for TPM keys a CA needs to be able to understand TPM key attributes and data structures and supplemental certificate types as they exist in a Trusted Computing environment.

- The work of **Chapter 5** addresses the lack of public[11] certification authorities for keys originating in a TPM. In this chapter we outline the design considerations and implementation trade-offs in our effort to construct a so-called *PrivacyCA* third-party certification service. In a Trusted

[9]Naturally, the respective hardware drivers for chipset generation and hardware devices must be supported by the software image.

[10]For example, whether a key is "migratable" or "not migratable", or whether it is to be used only for "signing" or "data encryption" operations—or is useable for both operations.

[11]Proprietary in-house setups are not known to us but probably exist in major companies.

Computing scenario as envisioned by the TCG a PrivacyCA service is responsible for the certification and validation of attestation identity keys (AIKs), which are the specific type of TPM keys designated to sign a platform state report data structure as produced by the TPM (for more information on AIKs see Section 2.5.2).

The prototype implementation developed and presented in Chapter 5 is a minimalistic service providing a basic set of commands. However, these commands are sufficient to produce and manage the necessary certificates for TPMs and their keys, which are at the core of the process of remote platform state reporting (attestation) in Trusted Computing.

The infrastructure specifications by the TCG suggest an XML-based protocol as one way of offering a certification authority interaction interface for clients connecting on the Internet. First we try the XML-based approach, however after some experience therewith we discard it in favour of a self-developed, custom protocol. The second revision of our PrivacyCA service implements our own custom protocol. This approach provides two important properties: First, a very compact implementation with only very few dependencies into the runtime environment. Secondly, the formal specification of the protocol enables the use of automation tools for robust generation of the protocol parser code for the security-critical, network input side.

Overall, in chapters 3 to 5 we present our work on how to build robust, compact, attestable services and processing platforms. Any external party can request from them proof that they are running the software expected to run on them, thanks to the integration of Trusted Computing.

When a client wants to use a (hopefully attestable) remote data processing resource, this leads to the question whether he can do this in a privacy-preserving way.

- The work of **Chapter 6** explores the challenge of privacy-preserving resource accounting in a distributed processing scenario: In our scenario a client possesses a smartphone and consumes cloud computing resources. He does not want to reveal his identity to the cloud provider and demands to pay anonymously for resources consumed.

  The approach we present in this chapter eschews the complexities of a general purpose e-payment architecture. Rather, our approach takes advantage of the limits of our scenario: The client wishes or is required to pay for the resources consumed from *one* specific provider.

  Consequently, our accounting scenario builds on a recent proposal of an anonymous payment scheme that ideally fits our scenario, since the scheme ensures the privacy of the client. This comprises two main properties: First, client anonymity and secondly, the unlinkability of each of the client's individual actions.

We map the scheme to our scenario and as a proof-of-concept evaluation we implement the most processing-intensive operations. Practical measurements of the scheme's execution speed on a set of hardware platforms suggest the scheme is fast enough for practical deployment in mass-market smartphone hardware.

The TPM v1.2 is a core technology used by the prototypes in Chapters 3 and 4 to support platform state reporting. Naturally, aside from this specific application the TPM can be used by application software in other cryptographic operations. Unfortunately, the features of the TPM v1.2 are already a bit dated, as it was introduced by the TCG in 2003. Consequently, TPM v1.2 features are not in line with the demands of current platforms and software[12].

In order to update the TPM feature set to better meet current demands the Trusted Computing Group has developed and released a specification of a new TPM v2 generation. First sample chips from vendors are available, however integration of TPMs v2 into mass-market platforms is a slow process. Naturally, support software is also still under development, which creates a chicken-and-egg situation surrounding the TPM v2, as with all new technology.

The TPM v2 specification advertises one prominent improvement over the v1.2 specification: The v2 specification is intentionally written in a specific format so that the specification is suited for automated tool consumption. This feature should allow automatic generation of code for the new TPM v2. Furthermore, once the specification is available as a parsed data structure, this enables the development of custom programs searching for errors and inconsistencies in the specification[13].

- The work of **Chapter 7** explores the specification of the new TPM v2 generation. We test the TCG promise on the suitability of the specification for automated tool consumption.

  We develop a semi-automated toolchain, which is a scripted process, and this accepts the published PDF format TPM v2 specification as input. As a result, first the textual content is extracted from the PDFs. Secondly, interesting portions are automatically identified, e.g. tables describing data structures. These tables are then parsed and their content interpreted to generate implementation code from them. Thirdly, the TPM v2 specification is more than 1400 pages in size, since it embeds code in the programming language C, so we extract it. This code illustrates the inner workings of a TPM v2 chip and its runtime environment. As a whole, extraction of the code along with the automated code generation produces a working TPM v2 emulator prototype.

[12] For example, while TPM v1.2 supports SHA-1 hashes, today SHA256 is preferred, as SHA-1 is no longer considered safe for a long time. Also, TPM v1.2 provides RSA cryptography, while today ecliptic curves are preferred over RSA due to their smaller key size.

[13] As a simple example inconsistencies or typos: if an identifier exits only once in the specification and is never referenced, there is the possibility of a mistyped identifier.

The resulting prototype allows anyone to develop for the TPM v2, while the hardware chip is still elusive in mass-market platforms. Furthermore, the intermediate data structures produced in our process are well suited as input for code generators producing TPM v2 interface code for various programing languages.

1.4 Overview

The chapters of this thesis are structured into the following major sections: Chapter 2 presents a succinct introduction to security and privacy/trust theory as well as to the technology and platforms used to develop the practical prototypes in the following chapters. Following this, Chapters 3 to 7 are outlined in Section 1.3 above. After that, Section 8 reflects on the work accomplished and speculates where the future might lead. Appendix A provides an overview of interim results and publications forming the basis of this thesis.

2 Background

This chapter provides a compact introduction to and overview of the technology, concepts, industry standards and related approaches in the problem domains touched on by this thesis. They provide the foundation for the contributions in this thesis—or more figuratively, they are the giant's shoulders those contributions stand on. The material in this chapter is naturally not exhaustive due to limited space. Rather, it provides a level of insight and understanding into the relevant problem domains, as most probably no one is well-versed in all of them. References to related works, publications and supplemental works are embedded in the text and so are pointers to additional material on specific topics when appropriate.

2.1 Trust

The term "trust" is for many an instinctive, natural concept. There is seldom an active, conscious thought expended on what it actually means. However, without a reasonable working concept of trust in daily social interactions our society would probably cease to function as it should:

> "Society runs on trust. We all need to trust that the random people we interact with will cooperate. Not trust completely, not trust blindly, but be reasonably sure (whatever that means) that our trust is well-founded and they will be trustworthy in return (whatever that means)." [111]

There are various social pressures, dilemmas, interests and moral norms which together enable this trust in our day-to-day social life. Discussion of the com-

plexities of social trust processes easily fill a book—[111] provides an extensive reflection.

Now that computers, computer networks and network services increasingly form part of our daily life, this raises the challenge of how concepts underlying our social experience of trust can be transferred and adapted for interaction with novel computing devices and services[1]. Why should someone *trust* all those new computers to do exactly what is claimed? Why should anyone trust a personal digital device, networked server or service to keep the user's data safe and do as expected in line with the user's preferences?

Comparison of the human/social aspect of trust to the technical/implementation side of trust in computing platforms crystallises into three problems [99], which are:

- The problem of unambiguous identification of a person or device.
- The problem of assessment whether a person or platform operates unhindered.
- The problem of expected/trusted reference behaviour of a person or platform.

In the following we explore each of these problems in more detail.

Unambiguous Identification

Humans beings are, by nature, very good at remembering other humans beings. One remembers a person's specific look and gait, for example. Similarly, almost every "thing" in our daily life is somehow unique, nothing is exactly the same. Also, as things are used, wear and tear makes them unique and thus makes re-identification of the same thing easier from a human perspective. A personal device like a smartphone may be very reliably identified as one's own because there is perhaps a known scratch on its outer case. Once it was dropped on ground by accident, but the scratch on the outside now makes it uniquely, easily re-identifiable.

From a technical perspective unambiguous identification of certain computer platforms and services is a challenge. For example, while it is easy to command a computer program to set up a connection to a certain server or service on the internet, it is then difficult to assess whether the job was executed correctly by the network components[2]—how can the correctness of the established connection be assessed?

[1] If users are not able to re-use and re-apply familiar concepts and have to learn everything from scratch when a new technology is released, the acceptance and adoption by users would be much more difficult.

[2] This is the optimistic view that the network is complex, causing occasional malfunction. The pessimistic view is that someone on the network is deliberately tampering with the connection, a man-in-the-middle attack.

Cryptography provides the means, algorithms and protocols to tackle this assessment problem. If it is known that a platform hosts a unique private cryptographic key, only this platform is able to perform an operation with this key. So one technical solution for unambiguous identification of a specific platform is challenging the platform to perform a certain key operation. If the operation is performed correctly, then at the other end of the connection is really the correct platform as expected[3].

Unhindered Operation

A human being most probably does not behave in a trustworthy way, if he or she is under certain influences. He or she might be suffering from a high fever, and consequently not everything said matches normal behaviour in a known, trusted, healthy state. Or as another example, if someone points a gun to a person's head, this external influence probably affects this person's further actions. External circumstances encourage modified behaviour.

Similarly, performance of a computer platform depends on unhindered operation. For example, the correct working of a standalone laptop is not easily influenced, and thus more trustworthy, compared to when there are one or more external devices attached to it. A single device or port connection is sufficient to maliciously take over complete control of a laptop (see also Section 2.6.2).

From the perspective of software operating on a platform the isolation of processes from one another is an important feature. It is undecidable whether a certain process or program operates as intended in the future or not. If, however, one process misbehaves at runtime, it immediately becomes untrusted[4]. It is important to ensure that all other running processes continue unhindered in their operation so that they and their data may remain trusted.

References

Social trust grows with the accumulation of positive experiences and references. A stranger may over time become a personal friend because of his history of consistent, good, cooperative behaviour in social situations. In the absence of personal experience it is natural to rely on recommendations and opinions from friends we trust, who can give references to good shops, quality craftsmen and quality services. If a good friend trusts someone, then it is reasonable to follow his perception of trust.

A similar accumulation of references applies in computing. For example, a user buys a laptop and installs his preferred software environment. From this point on forward the laptop is used every single day, and to the satisfaction of

[3]We admit that the practical implementation of this problem—see, e.g., the TLS protocol [29] and its infrastructure—is more complex than it initially appears, but this is not our focus here. We want to show the "unique cryptographic key for unambiguous identification" is one possible technical solution.

[4]For example, a process tries to access memory with data which is off-limits for it. Consequently the underlying operating system or hypervisor detects the memory segment access violation.

the user it performs as expected every day. Consequently, the user trusts his laptop more and more, it is a daily companion that so far has never let him down. If personal experience is lacking, for example with a new software program[5], the user depends on authorities he trusts, for example the local system administration staff, to help him choose and decide. Local system administration can help users with reference recommendations on device brands, software companies, software applications, network services, etc., which are trustworthy in their experience.

Technical Implementation of Trust

The previous three sections illustrate the connection between the social/human side of trust with the technical challenges of trust. Naturally, a program code does not care about social concepts and human reality when a trust decision is quite fuzzy and grows slowly over time. With networked computers a program needs a clear approach for a decision to trust[6].

Ideally, a trust decision can be reduced to a simple Boolean decision, or a numerical figure and a value range which allows a clear decision whether a value falls on either the *trusted* or *untrusted* side. If the result is "trusted", the program continues, if not, the program stops with an error and forwards the decision to a user.

The large problem domain of integrating components of trust into modern computers is extensively considered in [83]. In the following section we focus on one specific approach to provide base technology for the prototypes in this thesis, namely the implementation of "Trusted Computing" on industry standard PC platforms.

2.2 TCG's Trusted Computing

2.2.1 Introduction

The creation of platforms with enhanced security features is a continual challenge in computer science—[118] provides a historical overview and background. With the rise of industry standard PC platforms the problem of providing robust security functions and an increased resistance against certain attacks became the focus of PC hardware and software vendors. In the search for a solution an industry consortium formed to tackle this problem in PC mass-markets. The *Trusted Computing Platform Alliance* (TCPA) was founded in 1999 by leading industry vendors:

[5]It is impossible to possess personal in-depth knowledge about all programs. Even with open-source programs, whose code can be inspected in detail by everyone for correct implementation, this is impossible due to the effort needed.

[6]For example, a question to be decided is "whether the network connection was established to the correct computer running the correct software on the other side of the network."

> "[...] to encourage industry participation in the development and adoption of an open specification for an improved computing platform. The goal of this effort is to build a solid foundation for improved trust in the PC over time." [121]

The collaborative work of an increasing number of platform, software and technology vendors resulted in a trusted platform concept, the main component being the *Trusted Platform Module* (TPM) first published in 2000 [10]. Following these early efforts the TCPA was disbanded, and in 2003 it reformed into the *Trusted Computing Group* (TCG) initiative [128]. The TCG continues the work of the TCPA and still exists today as the core entity for specification and development of their Trusted Computing concept.

The TCG architecture specification defines "trust" as

> "...the expectation that a device will behave in a particular manner for a specific purpose. A trusted platform should provide at least three basic features: protected capabilities, integrity measurement and integrity reporting." [132]

From this definition it follows that trust is consequently not necessarily some kind of "good" behaviour, but rather trust is *expected* behaviour. For a platform which implements the three basic features mentioned in the above quote it becomes possible to assess whether the platform is in a certain state. If the platform is known to be in a certain state, the Trusted Computing concept assumes the platform will behave in the expected way.

The core building block in TCG's Trusted Computing architecture is the Trusted Platform Module (TPM). Integrated into a platform it provides the three basic capabilities demanded in the TCG architecture specification (see above), which are then the foundation on which further high-level features build.

2.2.2 Trusted Platform Module

History

Specification and development of the Trusted Platform Module (TPM) has been running for more than a decade. First major specification milestone was *TCPA Main Specification* Version 1.1a[7]. Soon afterwards, the consortium was reorganised into the Trusted Computing Group (TCG) of today and released Version 1.1b [140] in February 2002. This v1.1b release of TPM chips attained mainstream distribution and TPMs v1.1 started to appear on mass-market desktop PCs and laptops.

Based on experience with version 1.1b, the TCG continued to improve the TPM as the core building block of Trusted Computing platforms, leading to the release of version 1.2 revision 62 of the TPM specification in October 2003 [140]. This TPM generation has been continuously maintained over the years up until the current revision 116, published March 2011. The TPM v1.2 is widely

[7] Which is no longer available for public download from TCG.

distributed on a variety of platforms: the TCG estimated in 2011 [135] that worldwide more than 500 million TPMs have been shipped[8].

Parallel with maintainance of the v1.2 TPM generation work on a "next-generation" TPM progressed. The TCG is a member-only consortium requiring members to sign a non-disclosure agreement. The development of the TPM v2 was a multi-year process taking place within the TCG working groups, and the first publicly visible results in the form of a public draft specification were eventually published in October 2012 [142]. The copyright notice on the published documents suggests the TPM v2 effort had been running internally since at least since 2006. At time of this writing the current public TPM v2 revision is numbered revision 107 dated March 2014, and is still labelled as a "draft". First batches of TPM v2 chips were released from chip vendors into the market in 2014.

Features

A TPM is a stand-alone chip on a platform mainboard. It is a passive component, meaning it only answer to commands received from the platform and never initiates any action itself. It features cryptographic primitives similar to those of a smartcard, but in contrast thereto it is typically[9] physically bound to its host platform. The tamper-resilient chip hosts functional engines for asymmetric-key cryptography (RSA [101]), cryptographic hashing (SHA1 [77]), true hardware random-number generation (which feeds key generation functions), a keyed-hash message authentication code engine (HMAC), some volatile memory (working memory), a very limited amount of non-volatile memory and other features.

With these dedicated hardware cryptography support functions the TPM provides security functions for the platform and the software running on it (see Section 2.2.3). With the TPM chip operating independently from other platform components, the TPM embodies a hardware-based trust concept. More specifically the TPM was developed to support and provide three different roots of trust:

Root of Trust for Measurement (RTM)

A core feature of a trusted platform is measurement, logging and reporting of platform state. Upon platform hardware reset a special set of so-called platform configuration registers (PCRs) in the TPM are reset to a well-defined initial value. There are 24 PCR in a TPM v1.2, and all are of the size 160 bits[10]. PCRs cannot be directly written to. Rather, a PCR with index i, where $i \geq 0$ in state t is *extended* with input x (representing a single measurement) by setting $PCR_i^{t+1} = \text{SHA-1}(PCR_i^t || x)$. This enables the construction of a *chain of trust.*

[8]In private conversations with TCG members the estimate is now that over a billion TPMs have been shipped.

[9]In specific environments such as business server scenarios TPMs are also provided as plug-in modules due to backup and recovery requirements in case of mainboard failure.

[10]Equal to the size of a SHA-1 hash.

A platform's execution trail is documented in a chain of standalone measurements, which are continuously aggregated into the PCRs—for more details on this chain of trust concept see Section 2.3.2.

Root of Trust for Reporting (RTR)

With the RTR feature the TPM represents an entity trusted to report information in an unforgeable way. It can sign the current values in its PCRs with a cryptographic RSA key. This produces a cryptographically unforgeable report on current PCRs content, which can then be forwarded to an external entity as proof. This operation is called a TPM *Quote* operation. An enquiry from a (remote) third party into platform state and subsequent production of a signed PCRs report by the TPM is called a *(Remote) Attestation* process.

Root of Trust for Storage (RTS)

Naturally, the cryptographic keys used for quotes and attestation (see above) deserve special protection. To this end, the TPM provides *non-migratable* keys, which means these keys are bound to a specific TPM and that they can only be used within this TPM, and it is impossible to export them in the clear. If an application encrypts data to the public part of a non-migrateable key, this is called *binding*, as such data can only be decrypted with a corresponding private TPM key on a specific platform. An extension of this is *sealing*, where additionally a key may only be used by an application when the current PCRs are in a specific state. Consequently, decryption of sealed data is restricted to a specific platform state—which is expected to happen at some future time.

2.2.3 TCG Software Stack

Use of the TPM hardware chip security functions requires a support package of software and application libraries. The main component in the trusted software platform design is the TCG Software Stack (TSS). The initial v1.1 TSS specification was released in August 2003, with a v1.2 update following in January 2006. The current revision is "Errata A" dated March 2007 [131].

The TSS comprises a stack of layers with clearly defined interfaces in-between. The raw TPM chip data stream is handled by the operating system's kernel-level device driver. This raw device is managed by the Trusted Device Driver Layer Library (TDDL), representing to user-level code the singleton interface to exchange command byte streams with the TPM.

The two major components of the TSS design are the Trusted Core Services (TCS) and Trusted Service Provider (TSP). The low-level TCS is the system-wide singleton daemon service, which manages the TPM resources for all TPM clients. The TCS manages the TPM key slots, performs key handle and password management, PCR extend operations, event logging and enforcement of mutual exclusion for applications wanting TPM access. The higher-level layer TSP is the library component for applications, and typically a shared library applications

link to. The TSP API is specified in the programming language C. Normative C language `.h` header files are provided by the TCG. This ensures compatibility between TSS implementations from different vendors. Consequently, applications using the standard API, as defined by the TCG, should be able to execute with any TSS.

All modern operating systems recognize when a TPM is a component in the PC platform, and automatically load the kernel support driver and export a TPM device. A TSS's TCS daemon then claims exclusive access to the TPM device provided by the kernel.

The TSS specification was implemented by two major open-source[11] packages: First the C language-based *TrouSerS* implementation [155], the first source code revisions of which date back as far as 2004. Furthermore there is *jTSS* [127], a TSS implemented fully in the Java language. jTSS started as a TrouSerS wrapper in 2006 and soon evolved in 2007 to a 100% stand-alone Java library and service daemon, with the ability to directly access an operating system kernel TPM bytestream driver. Both TSS packages are still actively maintained.

2.3 Platform State

2.3.1 Introduction

The core challenges for assessment and robust reporting (attestation) of platform state are

1. the components contributing to or forming part of a platform's state

2. measuring and reporting these in an efficient way.

The first is actually the problem of separation of what *must* be measured, what is *not* to be measured and keeping them separate. A Trusted Computing Base (TCB) [25] comprises all components of a platform critical for security, integrity and reliability—whether hardware, firmware or software. Consequently, in order to assess platform state, the state of the TCB components state is crucial. Unfortunately, what components are part of a TCB depends also on the specific scenario.

The second problem is not only the measurement operation itself, but also the mapping of meaning for concrete measurement values, management of the measurement values database and drawing a conclusion from a measurement report.

In this section (Section 2.3) we discuss the above challenges from the perspective of current mass-market available PC platforms with integrated TPMs (Section 2.2.2).

[11] The specific history of commercial TSS implementations lies beyond the scope of this work.

2.3.2 The PC Platform

A platform power-on process (or hardware reboot) puts the platform into a well-defined initial state. This state is defined by the platform hardware manufacturer and could, for example, mean all memory is initialised to zero. Furthermore the low-level component initialisation code always sets platform components into an initial state. The specific steps performed should be described in the platform's documentation. The important insight is that power-up or hardware reboot should always put the platform into the same "fresh" initialised state.

When platform initialisation is finished, software executes, changing platform state. Consequently, in order to retrace platform execution history, the best way is to start from the defined initial state and retrace the intermediate steps that happened on the platform up to the current state.

On the common standard PC platform[12] the first code to execute is the BIOS[13] initialisation code, which brings the platform to a working state, then it subsequently initiates the hardware devices. When all hardware components are ready, the boot sector of the first device in the boot order is loaded, and the execution chain continues, first to operating system bootloader, then operating system, etc.

2.3.3 Chain of Trust

On a PC the integration of a TPM provides the functions for implementation of a so-called *chain of trust* rooted in the defined initial platform state. The BIOS is the first entity to execute, and is the anchor all following components implicitly trust, thus, it forms the *Core Root of Trust for Measurement* (CRTM). It is trusted to properly measure into TPM PCRs the first software or firmware executing[14]. For each subsequent block of code after the BIOS a so-called *transitive* chain of trust is constructed, meaning that in order to pass execution to the next block of code it is required that

1. the next block of code is measured and the measurement is extended into the TPM PCRs
2. execution control is transferred to the measured block of code.

Figure 2.1 provides a graphical overview of the PC chain of trust. If the chain of measurements is properly implemented, current values in the TPM PCRs are the

[12] With the meaning of a "Microsoft Windows"-compatible x86 platform based on Intel or AMD CPUs. The Mac platform is also based on x86 platform components and technology, however sometimes tends to implement things in its own "special ways".

[13] On recent PC platforms the traditional BIOS platform boot code is gradually being replaced by the more modern UEFI approach. However, the basic boot-up concept remains unchanged, so we do not differentiate BIOS vs. UEFI in this thesis. Also, the platform boot-up phase is no longer important when Intel TXT is used (Section 2.4.1), as we do in the prototypes developed in Chapters 3 and 4.

[14] The initial BIOS component measurements are defined in the respective PC platform specification [141].

result of all individual measurement events in the chain from platform reboot, up to the current platform state.

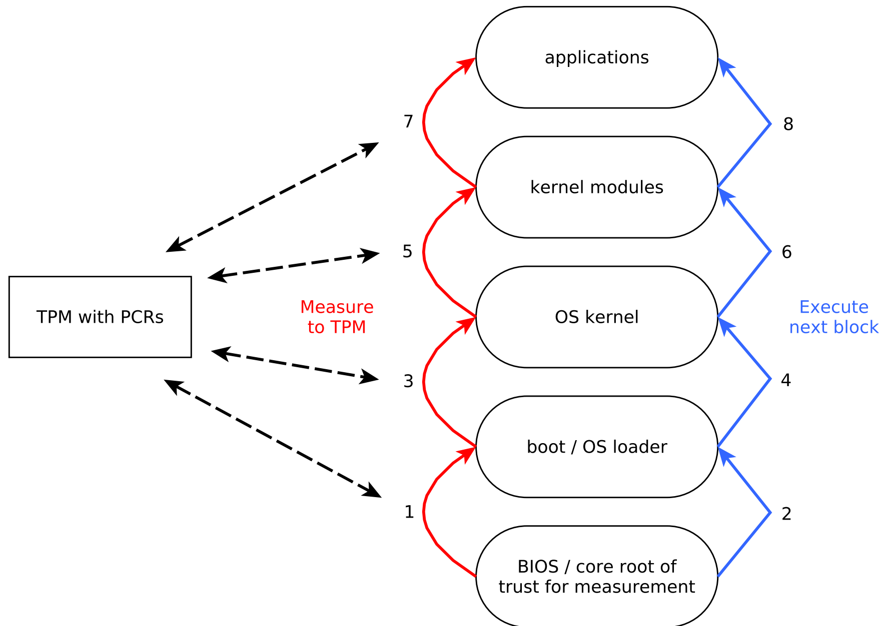


Figure 2.1: Overview of approximate chain of trust as implemented on the PC platform. Numbers indicate the order of steps. Each block is first measured (red) and the result extended into a PCR in the TPM, then execution control passes to the next block (blue).

2.3.4 Measurements and Complexity

As illustrated in Figure 2.1, the chain of trust is anchored in the BIOS and continues via a measurement-enabled boot loader[15] into the operating system kernel and the kernel drivers, and then further into high-level applications. However, though the early boot phase is usually deterministic and linear the modular design of the PC architecture and environment means the execution path is not always identical. With a general-purpose operating system implementing a chain of trust, on Windows [33] or Linux (see Figure 2.2), hundreds of measurements are performed in the chain of trust, in a non-deterministic order[16] until a simple basic desktop appears. Thus, though it is technically possible to implement a chain of trust, several practical challenges must be considered:

[15] For example, a modified version of GRUB [66].

[16] For example, at one point the DHCP client obtains a network address. However, a response from the network requires a random amount of time, consequently all further network initialisation happens a little earlier or later during boot-up.

```
     Measurement Value (fingerprint == SHA1) Measurement Hook       File Name
#000 9797EDF8D0EED36B1CF92547816051C8AF4E45EE     ima-init     boot-aggregate          Aggregate (
#001 F7A0BF5A67CE98BC06316F77CA1F404A2D447534    mmap-file     init                    Executable
#002 38C5D31E5DAD3F1B012FDD35B4E011E783CE6FD8    mmap-file     ld-2.3.2.so             Library
#003 42F79603219922016713BB8AAFC9E37F6936B226    mmap-file     libc-2.3.2.so           Library
#004 A4DC5EDF06698646CD76916F16E95C37E55DC12B    mmap-file     bash                    Executable
#005 F4F6CB0ACC2F1BEE13D60330011DF926D24E5688    mmap-file     libtermcap.so.2.0.8     Library
#006 AE1BC1746AFD2AC1ECD1D9EEEAEBD125A6A9EB8D    mmap-file     libdl-2.3.2.so          Library
#007 CFBC7EC3302145AB78A307C0D41DBB9A4251377B    mmap-file     libnss_files-2.3.2.so   Library
#008 805572455CF5BF50A7EE42E3CC6B0EDA65AF17A4    mmap-file     initlog                 Executable
#009 C95CBC5625719649103E0D1C3595967474842F7B    mmap-file     hostname                Executable
#010 0CAA342424F420FF29B7FB2FCF278F973600681B    mmap-file     mount                   Executable
#011 5E45D898530F31BADEF5E247EBCF4AB57A795366   bash-source    functions               Bash Sourc
#012 A253AF3AB981711A13AE45D6B46462386E628076    mmap-file     consoletype             Executable
#013 2E37B839BC4EC1B6BE1BDF5BACD1E7B56567D8D9   bash-source    i18n                    Bash Sourc
#014 C9D1B3E2CD0995E16AE6DD98B388FD873324740D   bash-source    init                    Bash Sourc
#015 590F75EE97E0FC560F07FCB07A8646FADEC88C2A    mmap-file     uname                   Executable
#016 5E851EFA4601B3AFCA9EAE75ED53688606630BFA    mmap-file     grep                    Executable
#017 32798F58C4F1B4CD017B09BCAAF2A22D345E7E4F    mmap-file     sed                     Executable
#018 CE516DE1DF0CD230F4A1D34EFC89491CAF3D50E4    mmap-file     libpcre.so.0.0.1        Library
#019 22EAF1B6009B23150367F465694AC63314866558   bash-script    setsysfont              Bash Comma
#020 8B15F3556E892176B03D775E590F8ADF9DA727C5   bash-script    unicode_start           Bash Comma
#021 A4C5F9D457DA16E47768423A68F135259F7180D7    mmap-file     kbd_mode                Executable
#022 497ED7F80C33AF25307DFC80970571C51006CE6A    mmap-file     dumpkeys                Executable
#023 04A0599405EBD306CEF2447679C8F4B5159A55C7    mmap-file     loadkeys                Executable
#024 AE327AD27D02BF2DE96557A1B4053D02129B1394    mmap-file     setfont                 Executable
#025 7334B75FDF47213FF94708D2862978D0FF36D682    mmap-file     gzip                    Executable
#026 93D65AB85CF5EE1ACD9E6BE5057D622D80AB5E10    mmap-file     dmesg                   Executable
#027 B6E90C3A25B69C3B1D3B643DB7D9504FBC36C1D1    mmap-file     minilogd                Executable
```

Figure 2.2: Example of a stored measurement log for the early boot-up phase of a Linux platform as produced by the Integrity Measurement Architecture (IMA) [106] feature of the Linux kernel.

- As TPM PCR extend operations are not commutative (i.e. measuring A then B does not result in the same PCR value as measuring B then A), the final PCR value is different after every boot-up, if measurements are not performed in a deterministic order. Consequently, the TPM operation of sealing data to a specific PCR state (see Section 2.2.2) has only limited applications in well-defined scenarios[17].

- In order to verify the current values in the PCRs, a stored measurement log (SML) must be maintained—for an implemented example see Figure 2.2. A SML logs every measurement, in order, with a description what was measured. A measurement by measurement replay or recalculation of the individual hashing operations can then be compared to the expected final PCRs state.

- A chain is only as strong as its weakest link. Consequently *all* components measured must be assessed for trustworthiness. An unknown measurement is not trustworthy. If a measurement is known, then it is "whitelisted" (known as trustworthy) or "blacklisted" (known as untrustworthy). As argued by [46], the approach of trustworthiness for all components as default and the creation of blacklists over time, as components are found to be untrustworthy, is futile. Instead, a better approach is to define a platform as trustworthy if all components in the chain are whitelisted.

  If one measurement or link in the chain of trust is not whitelisted, this means an untrusted component has been executed[18]. Consequently, after

[17] This property is a core challenge for the use of sealing in the prototype platforms presented in Chapters 3 and 4.

[18] In emphasis, a chain of trust documents platform *execution state* with the idea that the

this the platform is to be considered compromised, and all measurements accrued subsequently may have been tampered with, meaning the final PCR values cannot be trusted either.

- A database for collecting measurement fingerprints may become huge. For assessment of a platform's trustworthiness every measurement must be known. There are *a lot of programs* in the world and furthermore a certain version of a given program may appear as multiple measurements[19]. Consequently, the binary measurement matching approach leads to a measurements database complexity problem. This approach is practically feasible only when scenario complexity and involved components are strictly limited[20].

  There have been several efforts to move from binary matching to a more abstract level for measuring certain features[21], but so far no alternative proposal has replaced the binary fingerprint comparison of programs approach which is still the core principle in the new TPM v2 generation.

2.3.5 Virtualization for Measurements Reduction

The previous sections introduced the chain of trust and the component by component measurement process. The TPM PCR concept by the TCG provides a robust measurement and aggregation solution, but still the complexity problem of the potentially large number of measurements persists. Intuitively, it seems the larger a block of program code is, the harder it is to understand, review, and assess interaction with other programs for security, reliability and integrity.

In order to lower overall complexity, one approach is to measure fewer components, to break a large problem into smaller elements and to identify and isolate critical program pieces. To this end, a domain separation mechanism is the core component [103] partitioning the platform into isolated domains and ensuring there are only well-defined data exchange interfaces between them.

Isolation by Hardware Virtualization

The industry standard PC architecture has seen a constant evolution of features. In 2005 the two dominant CPU providers in the PC market, Intel and AMD, improved instruction sets and capabilities of their CPUs with support for full hardware virtualization[22], meaning any operating system could be run in a virtual machine without binary modification.

---

assessment of the individual components enables assessment of the platform *security state* [98].

[19]For example, every Linux distribution usually is compiled from scratch. Thus, while popular distributions may carry the identical program, due to slight differences in compiler or library versions program measurement fingerprints differ from one distribution to another.

[20]See, e.g., [67] and our platforms in the following chapters.

[21]For example, property-based attestation [104] [97] and semantic program features [49].

[22]See [1] for an overview of x86 architecture virtualization approaches.

The core management component with hardware virtualization is a small, single instance of a *hypervisor*. A hypervisor takes exclusive control of platform hardware and provides functions for creation, execution and hibernation of isolated virtualization *partitions*, each of which is capable of being host to a guest operating system. The hypervisor runs on the processor with special elevated "Ring -1" privileges, which isolates it from the standard operating system and applications code running with the traditional "Ring 0 to 3" CPU privileges.

With a scenario of hardware isolation the hypervisor is part of the chain of trust, as it is a lowest-level component. Depending on the scenario, multiple virtual machines are then started to run on top of the hypervisor. Some perform sensitive calculations, while others do not. Consequently, it is the job of the hypervisor to measure sensitive virtual machines before they start, so they become part of the chain of trust. However, measurement of the other virtual machines may be omitted (depending on the scenario), thus there is an overall reduction in measurements.

2.4 Dynamic Root of Trust

Section 2.3.2 introduced the root of trust for measurements, which on the PC platform sees the BIOS performing initial measurements after platform reset. The *static* root of trust for measurements (SRTM) is the chain of platform state measurements that begin at platform reboot [141]. State-of-the-art platforms from AMD [2] and Intel [46] improve upon this static approach, enabling a *dynamic* switch to a well-defined, measured system state at any point of platform execution, even long after a platform has been (re-)booted. This is called a *dynamic root of trust for measurement* (DRTM) [141].

As a consequence of this late initialisation this capability reduces the number of measurements necessary to document platform state changes because of the exclusion of early BIOS, firmware and boot measurements. Example open-source OS bootloaders, which implement this dynamic switch after the BIOS but before OS start, are available for AMD [61] and Intel [54] platforms.

2.4.1 Intel's Trusted Execution Technology

This section outlines Intel's specific implementation[23] of DRTM capability in their so-called Intel Trusted Execution Technology (TXT)-enabled platforms.

The so-called *late-launch* TXT process is initiated by execution of special Intel CPU instruction `GETSEC[SENTER]`. With this opcode the CPU and chipset stop all CPU cores except one. The platform chipset protects all memory from outside modification by DMA capable devices. Furthermore, TPM PCRs 17 to 22 reset from their default `0xff` content to `0x00`. CPU, chipset and TPM

[23] A comparable set of capabilities is available on AMD platforms, but discussion and feature comparison of the two platforms is beyond the scope of this thesis as we use solely Intel TXT in the prototypes of Chapters 3 and 4.

are hardwired to cooperate in this process, only a DRTM startup sequence is capable of (re-)initialising these PCR registers.

The first code to execute is a special Intel-provided and cryptographically signed *Authenticated Code Module* (ACM) [57]. The ACM code authenticity and integrity are guaranteed by a private Intel key, the public part of which is hardwired into the chipset. An ACM must be signed by Intel to allow execution. The ACM initialises the platform into a defined state and is anchor of a fresh chain of trust, this process providing a DRTM.

Subsequently execution continues with the *Measured Launch Environment* (MLE) [56]. Naturally, this code is first measured by the ACM and then executed. The MLE unlocks and re-awakens platform resources component by component and thus step by step it restores normal platform operation. The platform chipset remembers a TXT start-up process was performed. In order to protect potential memory secrets, it automatically enforces memory content erasure upon next system (re)boot, if a special TXT mode shutdown sequence is not executed first.

The ACM code is capable of enforcing a Launch Control Policy (LCP). The TPM administrator may store a LCP in the TPM non-volatile memory. A LCP specifies which MLE code binaries are permitted to execute after a successful TXT launch. This capability enables the system administrator to specify which code is allowed to run, as the ACM measures the MLE and ensures that the LCP value stored in non-volatile memory matches the expected MLE. If an unknown software configuration is detected, the ACM reboots the platform.

By default, all TXT platforms are shipped with a preloaded `ANY` policy loaded into the TPM and there is no restriction on which code can be executed after DRTM initialisation[24] is finished, because the ACM does not perform a check whether the code for next execution matches an LCP, indeed as the name suggests, `ANY` code is allowed.

2.4.2 Intel's tboot

The description of the execution flow with Intel's TXT in the previous section is rather abstract. In order to encourage TXT adoption, Intel provides a reference software implementation of an MLE called tboot [54], which implements the TXT late-launch in combination with an ACM, and is designed to be executed after early platform boot is complete but before an operating system starts. When tboot is used to implement a measured boot into Linux, a chain of trust where every component is measured, the execution flow looks like Figure 2.3.

After power-on or hardware reset the platform performs a standard start-up with the BIOS. Then the BIOS starts the OS boot loader, which in the case of Linux is usually GRUB [45]. In a normal Linux boot sequence GRUB loads the Linux kernel and its supporting initramfs into memory, and execution is transferred to the kernel. The boot process with tboot is different: GRUB loads

[24] Depending on the platform vendor, sometimes TXT support also has to be specifically enabled first in the BIOS.

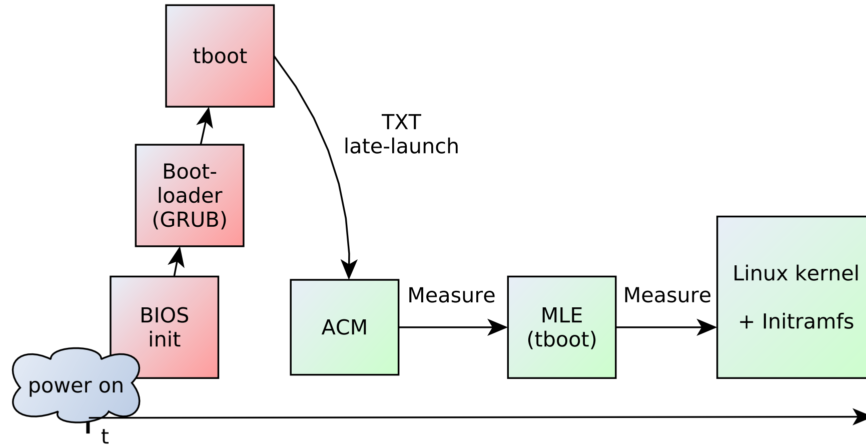


Figure 2.3: Overview of execution flow with Intel's tboot. Unmeasured components are red and measured components are shaded green.

the following files into memory: 1) tboot, 2) Linux kernel, 3) initramfs, 4) ACM, 5) tboot policy file.

Tboot is started by GRUB as the first component and performs two functions: First, the required set-up of memory layout and platform configuration before the TXT launch process is initiated, and second, after the TXT launch process is complete, a part of tboot is the MLE.

So first, tboot prepares the platform for the TXT start-up process. Secondly tboot executes the CPU opcode for the TXT late-launch. The hardware initialises itself and execution continues with the ACM module provided by Intel. The ACM configures the system further (see Section 2.4.1) and measures the MLE which is a part of tboot. If the TPM non-volatile memory contains a LCP, the ACM checks whether the tboot MLE is allowed to execute, and if not, the platform reboots. The ACM executes tboot as MLE, and tboot again checks with its own policy for restrictions on what code allowed for execution. If the loaded Linux kernel and initramfs are allowed as described in the tboot policy file, both are measured and executed. From this point on a normal Linux kernel boot process executes. The chain of trust is rooted in the TXT late-launch and continues into the Linux kernel.

To summarize, we see that the precise, desired software configuration can be specified by the platform administrator in the form of two separate policies: The Launch Control Policy (LCP) is evaluated by the ACM and specifies which MLE is allowed for execution. The MLE in the form of tboot interprets an additional policy called Verified Launch Policy (VLP). This two stage policy process is motivated by the practical restriction of non-volatile memory available in the TPM and deployment flexibility. The LCP is rather simple, as it essentially needs to contain a list of hashes of allowed MLE binaries. By contrast, tboot aims to be more flexible, and the VLP allows specification of multiple kernels, initramfs,

additional modules, module parameters, etc., in multiple configurations[25]. Consequently, tboot depends also on an externally supplied policy file, which would not fit into the TPM's very small non-volatile memory.

2.5 A Support PKI for Trusted Computing

The previous sections provided an introduction to the chain of trust (Section 2.3.2), where from a defined state a series of component measurements is aggregated into TPM PCRs (Section 2.2.2). Furthermore, the TPM can sign the current PCRs content with a cryptographic key. Thus, this signature marks an unforgeable cryptographic report of a platform's state (=the current PCRs content), which can then be presented to third parties as proof of platform state.

A verification process of a TPM signed report of a platform state requires a supporting infrastructure. There must be verifiable evidence that some data presented as TPM output was really produced from a genuine hardware TPM, not a TPM software emulator[26]. Consequently, this problem requires a TPM certification concept, and the binding of a platform report to a platform's TPM. Also, there is a potential privacy problem, as there is the possibility for unique identification of a platform in TPM signed state reports and during report verification.

The following sections outline the TCG-specified components and procedures involved in the creation of a support public key infrastructure (PKI) service for Trusted Computing-enabled platforms. Such an infrastructure comprises specific TPM keys, certificates, data formats and exchange protocols. This presentation is not exhaustive, rather it focuses on core elements and processes playing a role in remote attestation of platform state.

2.5.1 Credentials

Security credentials can be represented in varying formats. The TCG Credential Profiles Specification [130] specifies credentials in concrete instantiation of either X.509 certificates [50] and attribute certificates [34]. Additionally, some TPM functions produce binary blocks of data, the internal structures of which do not conform to any standard certificate field format. Consequently, the standard certificates were amended by the TCG with custom extension fields which can host this Trusted Computing specific data.

Endorsement Key and EK Certificate

Every TPM hosts a unique *Endorsement Key* (EK). This RSA key pair is stored in non-volatile memory inside the TPM. It is impossible to retrieve the private

[25] One might think of this as similar to a standard GRUB boot menu which also allows multiple configuration entries to start a Linux distribution in different configurations.

[26] For example [119] as a pure software solution has different security properties compared with a hardware TPM.

part of the key from the TPM—the specification of the TPM explicitly disallows extraction. A corresponding TPM Endorsement certificate hosts a copy of the public part of the key pair. This certificate represents the assertion that a given TPM identifiable by the unique public key embedded in the certificate conforms to TCG specifications and that the private Endorsement Key is protected by the TPM. The Endorsement certificate is signed by the entity that created and inserted the EK into the TPM[27].

As the Endorsement Key uniquely identifies a TPM and hence a specific platform (TPM), the privacy of a platform and its users is at risk if the EK is used to sign platform state reports, as all reports would be signed by the same key. As a consequence, the TCG rigorously restricted the operations that can be performed with the EK[28]. The EK cannot be used to sign the current state of the PCRs or arbitrary data.

Other Certificates

A platform manufacturer vouches for the platform components by means of a Platform Endorsement (PE) credential. This asserts the specific platform incorporates a properly certified TPM and the platform architecture conforms to TCG specifications. A reference to the specific TPM model and version integrated on the platform is included, which allows a PE certificate evaluator to form an opinion on trustworthiness of the platform.

A platform Conformance Credential (CC) attests that overall platform design satisfies TCG specifications. It asserts proper design of the Trusted Computing subsystem and its correct integration into the platform. The CC is described in [145], but has not been updated in newer credential profile specifications [130].

To the best of our knowledge neither platform or conformance certificates have been officially provided by platform vendors so far. With broader market deployment of the new TPM v2 the TCG will also update the certificate profiles and probably introduce new certificate types and profiles.

2.5.2 Attestation Identity Keys (AIK) and PrivacyCAs

The TCG remote attestation concept, which involves signature of TPM PCRs content with a TPM hosted key, implicitly produces a very detailed platform state description. This raises the question of privacy protection, as a platform report is evidence of what software has been executed since chain of trust initialisation. However, in some scenarios the exact declaration of software used should perhaps not be made available.

For one thing this information might be used to facilitate targeted attacks if, for example, it is known that a specific software version is vulnerable to a

[27]For example, TPM vendor Infineon provides an EK certificate with all of its TPMs. Verification of an Infineon hardware TPM thus requires verification of the certificates chain up to the Infineon TPM root certificate. Most TPM vendors do not provide TPM certificates due to cost and missing market demand (so far).

[28]There are only three: 1) For initial take ownership of the TPM, 2) for AIK certification with a PrivacyCA and 3) for Direct Anonymous Attestation (DAA).

specific attack. Another problem is that reusing the same key for all reporting operations allows tracing of a TPM (and thus the platform and the activities of its users) when reports are shared between third party entities. Although in some scenarios that might be a desirable feature, privacy protection should always be considered a primary feature[29] and may be relaxed at a later time by deliberate user choice.

Attestation Identity Keys

To address concerns of privacy the TCG specifications define Attestation Identity Keys (AIKs) and accompanying AIK certificates issued by a trusted PrivacyCA [129] service. An AIK is a TPM-created RSA key-pair which is *non-migratable* (see also RTS in Section 2.2.2) and which has the capability of producing a signature on the PCR state reports in the remote attestation process (TPM Quote operation). AIK certificates do not contain any information linking the certificates to the specific AIK hosting platform TPM. However, AIK certificates provide sufficient proof that the identity keys originate from a specific TPM series, giving vendor name and model. By creating and using more than one AIK per TPM, it becomes possible to mask the individual user/platform identity.

AIK Certification

Figure 2.4 gives a general overview of the AIK life-cycle and its practical application in a remote attestation scenario.

1. The first step is generation of a new RSA key within the protection of the TPM. In order to obtain an AIK certificate, the TPM-enabled platform and the PrivacyCA execute a cryptographically secured protocol.

2. The platform sends a request package containing the public AIK and the EK certificate of the TPM to the PrivacyCA[30].

3. The PrivacyCA checks the information received to see if the presented EK certificate (=TPM) is acceptable under the PrivacyCA certification policy.

4. On successful check, the PrivacyCA issues an AIK certificate and returns it. The returned data package is encrypted, so that only the TPM indicated in the request can decrypt it. Now the AIK key can be used for platform state attestation.

[29] See section on data protection in [35], especially: "If anonymous identities (AIK) are used which are made available by the security module (TPM), weakening of such anonymity by indirect linking of data from the TPM must be prevented (for example, by linking the AIKs via the TPM endorsement credential in a CA). Suitable technical or organizational measures must be implemented in order to ensure this."

[30] The request package contains additional information, such as additional platform certificates (Chapter 2.5.1)—the full details are omitted here as we focus on the EK and AIK properties. See TCG specifications for more details.

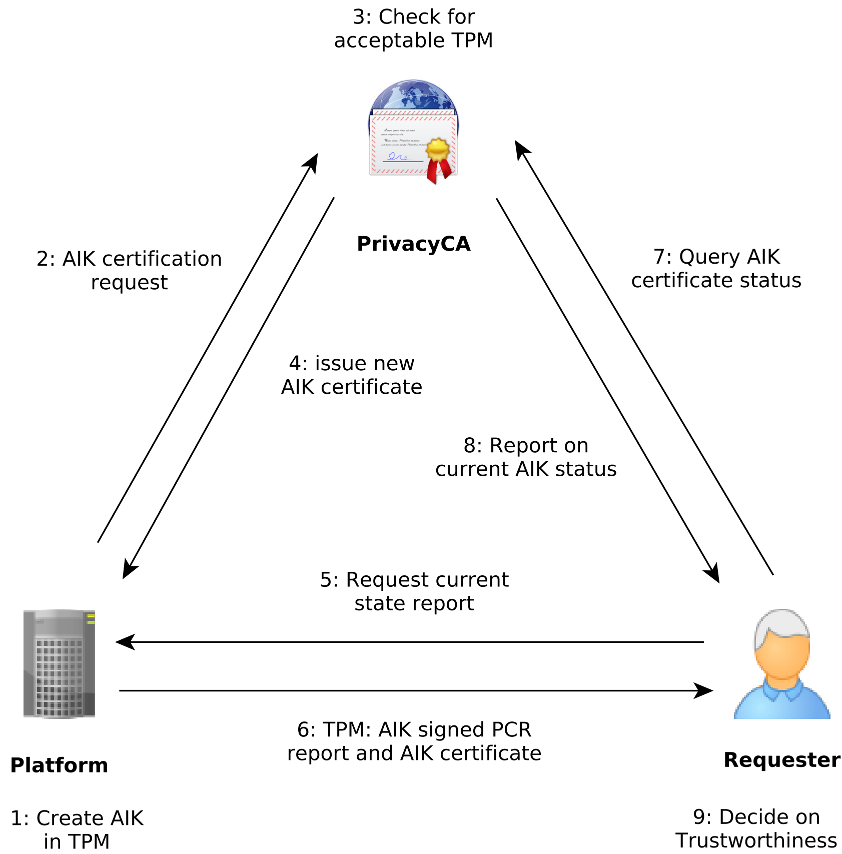


Figure 2.4: AIK creation and interaction with a PrivacyCA service provider for AIK certificate creation (steps 1 to 4). Requester queries platform state and checks back with PrivacyCA for AIK certificate validation (steps 5 to 9).

5. Later, an external third party known as requester interested in a platform's state queries the current state.
6. The platform software uses the TPM to sign the current PCRs content with the private part of the AIK and returns this report to the requester along with the AIK certificate.
7. The requester checks with the PrivacyCA the validity of the AIK certificate presented.
8. The PrivacyCA reports the AIK certificate status.
9. Finally, the requester checks correctness of the AIK signature on the PCR report. If everything is deemed in order, it is then up to the requester to decide whether the PCRs are in an acceptable state (e.g., a given software

version is running) or platform state is unacceptable (not trustworthy) to him.

PrivacyCAs and their Policy of Operation

A PrivacyCA assumes the role of a trusted third party and must be trusted by both parties in a remote attestation process. The platform (owner/user) must be assured that his privacy is protected in accordance with a specific policy, whereas an attestation requester must be assured that a remote attestation report is indeed signed by a proper TPM.

The mode of operation of a specific PrivacyCA service is documented by its certification and operation policy, which should be publicly available. The policy defines two important properties: First, who is allowed to acquire an AIK certificate. Secondly, whether any information on the requests received and certificates issued is retained, and for how long by the PrivacyCA.

In a restricted deployment scenario, for example, the PrivacyCA issues and validates AIK certificates only for well-known clients, which requires an initial registration step. In this case it might make sense to store all information acquired during an AIK cycle, but in more open scenarios, where the PrivacyCA issues certificates to a large set of customers that are not necessarily known beforehand, a more liberal policy might be appropriate. The policy options for a PrivacyCA range from *record nothing*, over *know enough for the specific operation, forget details after completion* to *store and log everything*.

The choice of PrivacyCA and privacy policy, and consequently the intended level of privacy, should be a free choice left to the end-user. If there are multiple PrivacyCA services with different policies to choose from, a user should be able to choose which PrivacyCA he trusts.

2.6 Aspects of Trusted Platforms

The construction of a platform to integrate Trusted Computing technology involves hardware and software components, each with their specific security issues and considerations. When several components are integrated and interact, further problems arise. Furthermore, Trusted Platforms need practical usability and easy deployability for adoption by users.

The following sections enumerate aspects for consideration in construction of a Trusted Computing platform, however not all aspects need to be considered for every platform, since this depends on the specific scenario.

2.6.1 Security of Trusted Computing Technology

Trust in something is established, when one can inspect it and attain an informed opinion of its inner working. Unfortunately, it is impossible to make a personal inspection of everything. Consequently, it is often necessary that one trusts the opinion, certification or competence of some third party (see Section 2.1). With

Trusted Computing one has to trust the TPM engineers and the engineers of Intel TXT as follows:

TPM

The only proof that a TPM is a real hardware TPM is embodied in the EK certificate (Section 2.5.1) for the TPM. Only very few TPM vendors (e.g., Infineon) include an EK certificate with their TPM chip. An end-user has to trust that during TPM manufacture the EK (certificate) creation process is secure and no-one has the opportunity of copying the private part of the EK, when it is injected into the TPM.

On the PC platform the TPM v1.2 has constituted the Trusted Computing basic building block for more than a decade. The v1.2 generation of TPM chips has been demonstrated to be compromisable through brute-force chip opening [120] or attacks on the local bus connecting the TPM and platform [161].

Intel TXT

The Intel TXT initialisation of a PC platform into a well-defined state relies on a binary platform initialisation code blob called authenticated code module (Chapter 2.4.1) supplied by Intel. While it is possible to disassemble and reverse-engineer its function, the code license forbids this. Some people nevertheless explore Intel TXT in detail and identify problems [164, 166, 165]. Furthermore, obviously only Intel engineers know in detail how their chipsets implement the TXT startup process internally, and how robust it really is.

Attack Resistance

The TPM chip is a mass-market device intended for integration into common PC platforms. The cost of adding an additional chip to the PC platform is significant[31]. Consequently, the TPM is by design a cheap, low-performance chip, but naturally such a chip is not a high-security module.

The TPM aims to increase trust in a platform by being a separate component. The TPM aims to resists software attacks (such as password guessing attacks). A TPM provides only very limited protection against hardware attacks, once a person has physical access to the platform (see above). The goal of Intel TXT is:

> ”...to force attackers to use hardware mechanisms to attack the platform. The Intel TXT design protects from attacks accessing memory, changing drivers, or changing the application. The attacker needs to use hardware access to gain access to the protected memory.“ [46]

For a book-length discussion of the security aspects of Intel TXT platforms see [46].

[31] The retail cost of a PC mainboard is as low as about 40 EUR. Private conversations with manufacturers give an estimate of TPM chip cost at about 1 EUR. Naturally, given these costs nobody integrates a TPM into PC mainboards, unless there is a good reason to do so.

2.6.2 Input/Output as Attack Surface

On the assumption the physical platform is safe, this leaves software attacks as primary attack surface. Runtime operating system bugs probably exist, since no software is completely bug-free. If, for instance, the TCP/IP stack of the Linux kernel contains a serious bug, it may as a result be possible to obtain platform control via remote access.

Furthermore, the primary platform exposure to the internet is the network card. Modern server network interface cards are no longer just "dumb" network packet transfer devices. Instead, they contain a small processor with firmware, which allows to perform remote management functions. For these functions to work, the network card needs main memory access through DMA and other powerful platform primitives. Consequently, if card firmware can be exploited, then platform security may be remotely compromised. A proof-of-concept work on this problem was demonstrated in [31], and the problem of attesting a network card firmware discussed in [32]. Thus, a network card exposed to the internet should be as dumb as possible to provide a minimal attack surface.

The same security argument applies to all other "intelligent" devices and interfaces on a computing platform, for example USB connections [112].

2.6.3 Practical Chain of Trust

In order to document platform configuration, the platform must be able to report its current state and provide a log of individual steps leading to that state (Section 2.3.3).

A platform's TPM provides a set of PCRs to guarantee the integrity and authenticity of individual measurements, but special care is needed to achieve expressive aggregated PCR values: Considering the complexity of a measurement chain, less is more. A shorter length of the chain of trust and fewer measurements facilitate subsequent verification. Mapping the intricacies of platform configuration into just a few PCRs is also a difficult task, since there are only 24 PCR on a TPM v1.2. Some of them are already reserved for the BIOS and some for a dynamic root of trust.

In order to be able to calculate expected PCR values *deductively* and seal data and code to a (future) platform configuration, measurements must be stable, individual measurement steps must be known, and their order needs to be constant. Only then can the benefit of sealing data blocks to the TPM PCRs be realised.

2.6.4 Integrity of Code and Data

The storage of sensitive data in unencrypted form is naturally a security risk, as this practice leaves the data ("data at rest") vulnerable to off-line manipulation attacks, so one solution is to encrypt the data. A more generic solution is to encrypt a whole file system on a platform, which provides a protection to all

applications and data on the file system. This poses the problem of where to store and how to protect the key to decrypt the data.

With a Trusted Computing integrated platform the obvious approach is to seal (Section 2.2.2) the decryption key to the TPM. Only when the platform attains a trustworthy runtime configuration defined by the system administrator do access and file system modifications become possible. Thus, this feature prevents off-line attacks and limits data modification to software currently running.

2.6.5 Open Source

A Trusted Computing platform that implements a chain of trust can attest the specific software that has been executed on it, but the questions is, who testifies the software is of good quality?

With proprietary software the vendor vouches for his software with his good name. With open-source software Linus's Law applies: "Given enough eyeballs, all bugs are shallow" [100]. This quote suggests that with open-source and a large enough spread of users and developers all bugs can/will be identified and fixed. Consequently, if the software of a platform is assembled from well-known, public sources, for example a major Linux distribution, this a) ensures the possibility of thorough inspection and b) raises confidence that the software is not maliciously manipulated, because it is hard to hide when a lot of eyeballs are looking.

Similarly, obscuring implementation details does not aid in the evaluation of security concepts. If platform blueprints, trade-offs and design considerations are available to interested parties, there is the opportunity to evaluate and eventually agree to the advertised security features of an architecture.

2.6.6 Boundary of Security Breaches

It is impossible to prove mathematically that a specific section of code does not contain fatal programming errors[32]. The pessimistic conclusion is that there are bugs in every code and sooner or later a security breach happens.

The common time-of-check-time-of-use [16] problem is a follows: Even when a chain of trust documents that a specific "known good" software version was executed earlier, and the platform state attested to an external party, after some time the platform may still be unsafe due to a later runtime security breach. There is always a gap between time of attestation and the present.

In order to assess the potential damage caused by a runtime security problem, it should be clear from the platform architecture which additional components are to be considered tainted, if a security breach in one component occurs. For example, a file system is only decrypt- and mountable in a specific platform state (Section 2.6.4). If a security incident happens, all currently mounted file systems are at risk of modification. If there are other file systems on the storage media that are only mountable in other platform states, these are safe from

[32] See also the "halting problem" in computability theory.

modification. The delimiting of the effect of security breaches is the "which filesystem is currently mounted" boundary.

2.6.7 Software Management Usability

A platform can be considered trusted when the software running on it can be attested to be trustworthy, meaning one can discover what is running on the platform and form an opinion on whether or not to trust it. This leads to the question of appropriate usability for software management in a trusted platform.

For wide platform acceptance it should not matter what kind of software is to be executed on the platform. The platform should allow installation of any common application and empower an administrator to define it as trusted. It is important that the overheads to maintain the components of a trusted platform and to perform updates of software packages should be reasonable, indeed not much more involved than on a platform without integrated Trusted Computing. Appropriate usability is needed for practical deployment, but trained system administrators are entrusted with sensitive platform maintenance operations.

2.7 Linux Platform Technology

We use the following Linux technologies in our prototypes:

2.7.1 Logical Volume Manager

Under Unix-like operating systems storage space can be abstracted as a collection of data blocks of fixed size sequentially numbered from 0 to the last block. For example, a typical harddisk has a block size of 512 bytes or 4096 bytes, and the sectors on the disk are numbered from 0 to the last one. Partitioning of a large storage device, e.g., a hard disk `/dev/sda`, results in several smaller partitions `/dev/sda1`, `/dev/sda2`, `/dev/sda3`, etc. Applications that work with storage do not normally care where the storage actually comes from. The kernel is responsible for mapping access to "block 0" of any storage space to the "real" physical storage subsystem block number.

LVM is a Logical Volume Manager [151] for Linux. Basically, LVM takes as input a large chunk of storage and divides it into smaller pieces called logical volumes (LV). The advantage of LVM is that it facilitates dynamic management of logical volumes. They can be created, resized, combined, deleted, etc. at will during runtime. Each logical volume is identified by a unique name tag, which applications use to access specific logical volumes. Consequently, it is transparent to an application where the actual storage originates, the LVM layer maps in the background all access to actual storage spaces. A common application of LVM is in management of a single large harddisk partition in smaller logical volumes.

2.7.2 Linux Unified Key Setup Encryption

Linux Unified Key Setup (LUKS) [39] defines a platform-independent standard disk format for use with disk encryption. Any block device in Linux may be transparently encrypted[33] with Linux kernel's `dm-crypt` subsystem. `dm-crypt` encrypts block devices with symmetric AES keys, called masterkeys. A LUKS header is responsible for management of keys used to encrypt a device. LUKS provides 8 slots with keys that encrypt one master key with different access secrets (e.g., a password). If a client knows one correct secret for one of the 8 key slots, he can decrypt the partition's master key and can then access the data. This LUKS key management and encryption abstraction delivers several advantages:

- The AES master key for disk encryption can be generated with high entropy independent of user input or password memorization.
- When one key slot secret changes, this does not necessitate the re-encryption of the whole storage.
- Key slots may also be assigned to maintenance processes such as backup. In the event of hardware platform failure this allows recovery of an encrypted storage.

2.8 ARM Platforms

New devices fulfil user needs and gain market share, as a PC is no longer the only option (see Section 1.1) and today small devices like smartphones, tablets and similar devices are popular. These devices are almost always powered by processors of the ARM architecture, not like PCs with processors from the x86 family. The ARM architecture offers a good performance vs. power consumption profile for mobile and embedded scenarios.

The TCG approach (Section 2.2) of Trusted Computing, implemented via a separate TPM hardware chip, has become standard in mass-market PC desktops and laptops. In order to enable the same concepts of Trusted Computing for these new classes of small devices, the TCG specified a TPM variant, a Mobile Trusted Module (MTM) [134]. However, this design never achieved mass-market deployment. The significant constraints on mobile devices, for example available PCB area, cost, power consumption and heat dissipation, do not favour a stand-alone security chip type approach or MTM-style deployment [48] [110].

The ARM platform allows for a security conscious environment: First, TrustZone as a hardware technology to support isolation of certain critical components from the rest of the platform. Secondly, the Trusted Execution Environment (TEE) standard defines how applets can be managed and executed in a TrustZone-like isolated environment.

[33] An application accessing data on such a device does not know whether data blocks are encrypted or not.

TrustZone

The ARM architecture follows a building block approach, where the main CPU core and instructions are centrally developed and licensed by one company[34]. Also, additional specialized extension blocks are possible. Individual CPU vendors elect and integrate the main ARM core and custom extension blocks as needed, so final chip capabilities fit their designated implementation scenario(s) and market very well.

One ARM CPU extension is *TrustZone*[7], which is an instruction-set extension that supports security critical scenarios. With these instructions the CPU can partition all platform software and hardware resources, thereby creating two virtual domains, namely a so-called *secure world* (SW) and *normal world* (NW) [6]. The idea that all resources exist in one of the two worlds, the security subsystem in the secure world, and everything else in the normal world, is an improvement on the basic concept of privileged/unprivileged mode-split which is found in many conventional architectures, including earlier ARM cores.

The TrustZone domain separation mechanisms prohibit normal world applications direct access to the secure world. Instead, the data flow between the worlds is controlled by a *secure monitor*, which is under exclusive control of the secure world. The total memory available for software in TrustZone is vendor-dependent, ranging from 64 kB to 256 kB on typical systems. This size enables running of a small, hopefully evaluated and certified, core in the secure world along with trusted executables.

Trusted Execution Environment

The TrustZone capability of ARM CPUs enables the construction of a *Trusted Execution Environment* (TEE) in the secure world. The purpose of a TEE is that it guarantees code and data loaded inside the TEE is protected with respect to isolated execution, confidentiality and integrity. By comparison, on a PC platform there is the solution of a chain of trust, which measures code integrity, and a hypervisor, which partitions the platform into virtual machines (see Section 2.3.5).

As there many ARM platform vendors[35] there is no standardized way of reporting the authenticity of the TrustZone implementation with a certain vendor and to attest the TEE running in it. A certification-based approach as in the TPM concept (see Section 2.5) does not exist (yet). However, typically TrustZone implementers provide platform capabilities for device identification and bootloader binary enforcement. These features facilitate implementation of capabilities similar to a measured boot chain of trust and signed reporting of the measurements. Unfortunately, the actual capabilities and details of these

[34] ARM Limited, UK

[35] The x86 platform is dominated by Intel CPUs and Intel chipsets, thus it is to easy to promote a standard like Intel TXT, as there are few compatibility problems between vendors and the core components are always from Intel. CPU, chipset and TPM are guaranteed to cooperate in initialisation of a dynamic root of trust (DRTM).

custom security features vary between platforms from different vendors and the manufacturers consider them confidential information.

The GlobalPlatform consortium drives the standardization effort for TEE implementations for TrustZone-enabled platforms and similar environments. TEE application programming interfaces (API) specify the communication interfaces between untrusted and trusted zones, as well as design and access of trusted applications in a secure execution environment [43, 44].

2.9 An Anonymous Accounting Scheme

At the core of the scenario presented in Chapter 6 is an "anonymous yet authorized and bounded cloud resource scheme". This scheme was developed by Slamanig [117]. In Chapter 6 we employ the scheme to implement a privacy-preserving method for cloud resource accounting. In the following we give an overview of the operations in the scheme, while for the mathematical details we refer to the original work in [117].

The main idea behind the scheme is that clients (C) can purchase from a cloud provider (CP) a contingent of cloud credits (CC) for resources, such as virtual machine instance hours. While clients spend their CC on resources from the CP, the latter does not learn anything about the clients resource consumption behaviour. More precisely, clients can consume an arbitrary number of their CC, as long as there are still enough credits available from their purchased amount. In any interaction with a client the CP is sure the client is allowed to consume resources, but cannot *identify* the specific client nor *link* any of the client's individual actions.

A token t is of the form $t = (CO(id), CO(s), L)$, where $CO(x)$ denotes an unconditionally hiding commitment[36] to value x. The value id represents a unique identifier of the token (chosen by the client at random), s represents the number of CC consumed up to now (essentially a counter) and L represents the credit limit.

The scheme of [117] is based on a pairing-based Camenisch-Lysyanskaya (CALY) [21] signature scheme. The CALY signature scheme supports not only the signature on data tuples (vectors), but also allows that individual data elements either to be signed in plain or commitments made to them. In our above example of token t there are 3 elements, where the first two elements are commitments and the third element is in plain.

We focus on three operations:

- `ProviderSetup`. The cloud provider generates a key-pair (sk, pk) for the CALY signature scheme. The CP publishes the public key pk and initializes an empty blacklist BL. This blacklist is later required for double-spending detection, it keeps a record of ids of tokens already shown to the CP.

[36] A cryptographic commitment to some value x is $CO(x)$. The value x remains secret (hidden) until opening information is provided by the committer. Then it can be verified that $CO(x)$ contains the value x.

- `ObtainCredits`. In this operation a client C wants to obtain a token t for L credits (the credit limit) from the CP. The client obtains a CALY signature σ_t for a token $t = (CO(id), CO(s), L)$ from the CP. id is a random token identifier generated by C. id and s are signed by the CP in the form of commitments for the unlinkability property in the following `Consume` step(s). However, the client C proves to the CP in zero-knowledge[37] that the commitment to s contains the correct default starting value, e.g. 0. Initially no credits s have been spent.

- `Consume`. The client holds a token $t = (CO(id), CO(s), L)$ and corresponding signature σ_t for it from the CP. The client wants to consume n CCs from his token, so the client a) randomizes the signature σ_t to σ'_t (σ_t and σ'_t are then unlinkable by the CP). The client b) proves in zero-knowledge to the CP that σ'_t is a valid signature for id and L, this includes showing the values id and L to the CP. Also, c) the client proves that the token t includes an unknown value s, which satisfies $(s + n) \in [0, L]$. This proof convinces the CP that at least n CCs are available in this token. The CP checks whether he has been offered this token id before (id is not contained in BL), and if all proofs succeed, then the client is eligible to consume n CCs.

  Then, in an interactive protocol between the C and the CP the signature is updated to a new signature for the new token $t' = (CO(id+id'), CO(s+n), L)$, where id' is randomly chosen by the user[38] and CP is assured that the correct value n is added to s. Also, the CP adds id to BL.

  If any of the computations or range proofs during the `Consume` operation fail, the CP rejects the client's request to consume n resources.

The drawback of this scheme variant is that the limit L is included in plain in the token and thus is always visible to the CP. In the worst case, L is issued to exactly one client and the cloud provider can subsequently link the actions of this client. However, if the set of clients associated with the same value L is reasonably large, unlinkability is no longer a problem. A practical solution would be for cloud credits to be sold only in specific well-known amounts, similar to prepaid mobile phone cards, e.g., 5$, 10$, etc.

2.10 Related Work

Trusted Platforms and Technologies

The platforms of Chapters 3 and 4 integrate a measuring boot process, the TPM, Intel TXT, whole file system measurement, KVM+QEMU as virtualization technology and Linux or Android as host OS. Some of this technology as

[37] A zero-knowledge proof is an interactive proof in which a verifier is convinced of the validity of a statement but learns nothing beyond the validity.

[38] So $id + id'$ is again a new random identifier unlinkable to id for the CP.

well as their predecessor technology have been used in earlier works to construct "trusted" platforms or features thereof.

Before the TPM, the approach of providing trust functions via a dedicated, separate secure hardware coprocessor was studied by Dyad [146]. The AEGIS [5] project modified a PC BIOS to provide a root of trust (and a chain of trust), and an integrity-enforcing boot process for a FreeBSD operating system. We do not want to modify the BIOS, and fortunately today we have a TPM.

The Enforcer platform [68] and IBM's Integrity Measurement Architecture [106] implement Linux kernel modules to perform file-level integrity measurements ("measure before first use") into a TPM PCR. For our platforms we rely on whole file system measurement to omit the complexity of per-file measurements.

Entire file system images as a solution to move applications and data are used by SoulPads [19] and Secure Virtual Disk Images in grid services [41]. They propose the use of encrypted images of virtual machines and the use of TPMs to enforce their integrity and boot-up only on "trusted" platforms. Our platforms assume the concept of encrypted/sealed virtual machine images.

Early demonstration systems making use of virtualization for component isolation are PERSEUS [84] where a Fiasco kernel is the hypervisor and L4-Linux a guest, and Terra [40]'s Trusted Virtual Machine Monitor, which is based on VMware. The Nizza virtualization architecture [115] uses a L4 microkernel to isolate security critical modules with a small TCB. We use KVM on Linux as the low-level hardware management and isolation hypervisor instead.

The Open_TC [78] project demonstrates a platform based on a static chain of trust from BIOS to bootloader via Trusted Grub [66] to Xen [11] or L4 as hypervisors and into application partitions measured and loaded from CD images. Open_TC with Trusted Grub depends on BIOS measurements, so we use Intel TXT to avoid this. Also, Open_TC does not provide any advanced at-runtime platform rebuilding features like our TVAM component in Chapter 3 does.

The challenges for a x86 hypervisor are discussed by Vasudevan et al. [158]. As a result, we chose KVM as hypervisor, as KVM uses hardware features ("Intel-VT"). Furthermore, as KVM is part of the Linux kernel KVM is actively (security) maintained.

There are several approaches to executing small pieces of code in an isolated context: BIND [114] uses AMD's Secure Virtual Machine (SVM) [2] protection features to collect fine-grained measurements on both input and the code modules that operate on it, so that computation results can be attested. Flicker [70] isolates sensitive code by halting the main OS, switching into AMD SVM and executing with a minimal TCB small, short-lived pieces of application logic (PALs). PALs may use the TPM to document their execution and handle results. TrustVisor [69] is a small hypervisor initiated via the Intel TXT DRTM process. It assumes full control and allows to manage, run and attest multiple PALs in its protection mode, however without the repeated DRTM mode switch costs incurred by the Flicker approach. We do not aim to run only short pieces of code. Also, our platforms aim to boot only once via a DRTM for establishment of a chain of trust and then continue into a normal (Linux) operating system.

OSLO [61] is an OS loader module which implements a dynamic switch to a measured state in the OS bootchain on AMD SVM systems. OSLO is a proof of concept and does not boot policies. We use Intel platforms and tboot [54] instead.

Trusted Platforms and the Cloud

The TPM promises the possibility of strongly identifing a single platform in the cloud, measuring and reporting the software configuration and protecting the integrity of data and code stored in the cloud. In our platform presented in Chapter 4 we explicitly take advantage of the TPM to construct a chain of trust. Furthermore, we use the up-to-date Android platform as base. We also prototype a cloud join protocol which is based on remote attestation. However, we do not move beyond these relatively low-level constructs.

There are several other efforts that aim for trusted, attestable cloud nodes or services running on them. By contrast, none of these use Android as a base as we do. Furthermore, they study a higher-level management of trust in the cloud while we focus on the low-level platform boot and integration, or they do not integrate the TPM and TXT as we do.

Santos et al. [107] propose a trusted cloud computing platform (TCCP) approach to verify trust in virtual machines rented from a cloud provider. No practical implementation is reported.

Also Krautheim et al. propose in [64] the Trusted Virtual Environment Module (TVEM), a software module that serves as virtual security module for IaaS cloud applications on virtualization platforms. The TVEM aims to offer features beyond the TPM, allowing platform owner and cloud user to share responsibility and control over data in the cloud.

Brown and Chase [18] propose using remote attestation, so that users can gain insight and trust into service applications by leveraging trust in a neutral third party. They assume cloud platform and provider to be trustworthy, without actually relying on hardware security mechanisms.

SICE [9] is a framework providing hardware-level isolation and protection for sensitive workloads running on x86 platforms in compute clouds. It is not based on a traditional hypervisor, but utilizes the System Management Mode (SMM) of x86 CPUs to isolate CPU cores. The presented prototype therefore requires customized platform firmware and currently does not integrate further trust mechanisms such as the TPM.

The IBM Trusted Virtual Data centre (TVDc) [15] is designed to offer security guarantees in hosted data centres. It provides containment and trust guarantees based on virtualization. Isolation and TPM-based integrity are managed. It builds upon a hypervisor derived from Xen and performs TPM-based software measurements.

The myTrustedCloud [160] project studies integration of an existing IaaS cloud platform with KVM-based virtualization and hypervisor trust mechanisms built upon IBM IMA. Different levels of attestation are provided for the different layers in the software architecture.

In [109] Schiffman et al. propose a centralised cloud verifier (CV) service which aids customers in verifying the integrity of a cloud platform running customer VMs in IaaS clouds.

Podesser and Toegl study the use of the platform presented in Section 3 in a cloud scenario [96]. They demonstrate seamless integration of remote attestation in a SaaS cloud for Java applications. Their architecture enables developers to annotate code with security requirements to be automatically enforced throughout the attested cloud.

PrivacyCAs and Trusted Third Parties

In Chapter 5 we develop a PrivacyCA service to issue and manage certificates for Trusted Computing platforms. To the best of our knowledge our PricacyCA service responder was the first of its kind to provide a public service.

We launched the first XML based prototype into operation in 2007 at `http://opentc.iaik.tugraz.at/`, as this work was sponsored by the OpenTC project [78]. The second iteration of our PrivacyCA followed a year later at `http://privacyca.iaik.tugraz.at/`. Over the years it was only rarely used by external parties. However, it was part of the local annual university "Trusted Computing" lecture in the practical exercises. The website and certificates of our service have now expired and remain unmaintained[39], consequently all certificate validations fail.

The website `http://privacyca.com/` was started in 2008 by a private individual. Its function was to issue only AIK certificates with a basic HTTP-based interface. This service did not develop any further and is now off-line, as the private operator died in 2014.

The 2012 announced Open Attestation Project [157] is an Intel-initiated open-source project to develop infrastructure for host integrity verification using TCG-defined remote attestation. The project aims at cloud and enterprise deployment. Naturally, it includes a PrivacyCA component.

As an alternative to the trusted third party concept of a PrivacyCA service, Brickel et al. propose Direct Anonymous Attestation (DAA) [17]. The RSA variant of DAA has been implemented in the TPM v1.2 generation, while TPM v2.0 provides the commands for a ECC variant of DAA [167]. However, the general-use software and service infrastructure required for practical DAA deployment is missing up to now.

A Bare JVM And a Stripped Down OS

The PrivacyCA service developed in Chapter 5 can execute on a OpenJDK [80]-based minimized Java Virtual Machine (JVM) with a reduced Linux operating system as base. Neither of the following efforts provided a small profile and sufficient functions to be selected as a base for our PrivacyCA.

There are, e.g., a small-sized JVM [79], a JVM which can run on a small OS [153] or a JVM which also assumes the services of an OS and runs directly on the bare hardware (or in a virtualization compartment) [150].

[39] As the research projects sponsoring this effort have also ended.

The latest official release of Java from Oracle is Java 8, which was released for mass-market adoption in March 2014 [81]. Project Jigsaw [82] aims to implement a module system for the Java runtime environment. A modular runtime provides the possibility of excluding unnecessary runtime parts, which consequently leads to a small base footprint of the Java runtime environment. However, the Jigsaw effort has been pushed back for integration into the future Java 9 release.

Privacy-Preserving Accounting and Payment

In Chapter 6 we develop a scenario where a resource constraint client takes advantage of remote cloud services. The cloud services must be paid for, however the client wants to make payment in a privacy-preserving way. To the best of our knowledge an integrated approach, privacy-preserving algorithm, a cloud scenario and security-enhanced hardware platforms as presented in Chapter 6 has not been considered before. There are three lines of related work whose combination leads to our work presented in Chapter 6.

Privacy and trustworthy mobile platforms: The work of [159] implements a variant of direct anonymous attestation (DAA) [17] for mobile devices and extend TLS [29] with it. They split the computation into a TrustZone isolated normal world and secure world part to prevent copying and sharing of private data/credentials. They also provide anonymity and unlinkability for the mobile client against a verifier. Similar to this idea in Chapter 6 we also split our computations and aim for anonymity and unlinkability of client actions. However, we focus on the problem domain of anonymous resource accounting.

Privacy and cloud computing: The following works recognize the problem of hiding client identity and actions from other clients and the cloud provider. The problem of anonymous cloud storage access and manipulation is in addition to the payment problem we focus on in Chapter 6, however both problems need to be solved to protect client/user privacy.

Sharing of resources among different users may potentially lead to construction of covert or side channels facilitating inference of activity patterns of other users [23]. Consequently, user's access patterns represent privacy-sensitive information that should be hidden from the cloud.

Others works focus on storing and sharing data in the cloud. [37] present an oblivious RAM (ORAM)-based approach, which allows users to outsource a set of data items to the cloud. No other users or even the cloud provider observing access can infer which user is accessing which data items how often. Another approach based on dynamic accumulators is proposed in [116]. The cloud and other users may learn which data items are accessed, but each access is anonymous and unlinkable to another. Hence user usage patterns can also not be inferred.

Privacy and payments: Concepts for anonymous and untraceable electronic payments have been around for quite a long time [22]. While these first schemes were based on blind signatures and the cut-and-choose paradigm[40], over the

[40] For example if there are n claims and $n-1$ of them are checked to be correct, it is

years several improvements, especially for off-line e-cash, e.g., compact e-cash [20] and divisible e-cash [8], allowing to spend 2^n coins from a “single coin” accumulating all coins, have been proposed. Also, the use of anonymous payments for overlay networks like Tor [4, 24], which use lightweight payment protocols for micropayments, have been proposed.

Unlike all aforementioned schemes, which assume a bank, a set of payees and a set of payers, in our scenario in Chapter 6 we have just one bank and one payee represented as the same entity, the cloud provider. Thus, we do not need to employ offline schemes, but use a kind of online payment scheme. Since we do not need the properties of e-cash schemes which aim to replace real money and all its features, such as clients spending arbitrary amounts with arbitrary payees, which adds a non-trivial computational overhead, we can employ a scheme tailored to payment for cloud resources in our work. The reduced scheme complexity and reduced set of transactions in Chapter 6 allow us an implementation which is just fast enough to run on a very resource-limited device, namely a smartphone.

Autogeneration of a TPM and Software Bindings

In Chapter 7 we semi-automatically generate a TPM v2 simulator solely from the official PDF specification. To our knowledge a tool parsing the TPM v2 specification and generating something from it exists and is used during the development of the specification. As hinted in part 4 chapter 4 “Automation” of the specification: “The automated processor is not provided to the TCG. It was used to generate the Microsoft Visual Studio TPM simulator files.”[144]. We know the tool is not public and can only speculate that even not all TCG members have access to it. More information on this is probably available under a non-disclosure agreement from the Trusted Computing Group.

A somewhat similar effort at auto-generation, but in the reverse direction, was implemented by [60]. In this study a TPM v1.2, its data structures and commands is specified in a custom domain specific language. This specification is then the input for an automation toolchain which generates a) a human readable specification in PDF and b) an implementation skeleton of a TPM in C. As during this work the TPM v2 was published, this work on the TPM v1.2 is only a proof-of-concept with a limited selection of examples.

---

reasonable for large n to believe the last one also to be correct, but without explicit checking.

3

A Platform with Enforcement of Application Integrity

3.1 Introduction

Assessment of remote platform state is a problem which Trusted Computing solves through its chain of trust approach (Section 2.3.2). Intel TXT builds on the foundation provided by the TPM and explicitly gives a platform administrator the power to restrict which code is allowed for execution via TXT (Section 2.4). The ability of ensuring that only specific, administrator-blessed software can TXT boot on a platform supports additional data protection: a block of data is sealed (Section 2.2.2) to the TPM, so that it can only be decrypted in a certain platform state as defined by the administrator. While above features might appear to be simple, practical implementation and integration into a full-size operating system are not.

In this chapter we explore considerations and trade-offs in construction of a platform prototype to integrate Intel TXT for code and data protection. More precisely, the platform constructed in this chapter aims to combine the following properties into an integrated platform prototype:

- Use of up-to-date standard Intel TXT enabled PC hardware that is mass-market available as a platform base. Selection of mass-market hardware lowers costs and enhances availability and adoption.
- To take advantage of TXT hardware platform security features with a dynamic root of trust (Section 2.4) for measurements based on Intel TXT along with a TPM as central component. Hardware virtualization features

of TXT platforms (CPU and chipset) provide for separation of software blocks into virtual partitions. In addition, TXT features also enable an administrator to restrict execution of software partitions to trustworthy, integrity-enforced configurations. Furthermore, these technologies enable platform state reporting to other hosts. Also, data sealing and remote attestation become practical due to deterministic measurement of individual code block/partitions into the chain of trust. The use of sealing protects platform data at rest.

- A set of administrative procedures and mechanisms to help retain administrator ease of use and administrative flexibility in the face of configuration changes, guaranteeing defined transition from one trusted state to another.
- Almost all platform building blocks originate in open-source components, and this supports individual inspection of each component and fosters open discussion about platform security properties.

The resulting prototype implementation described in this chapter was published under the name *acTvSM*[1] platform and is available for public download and experimentation at [92]. Furthermore, the interim results of this effort have been previously reported in several publications—see Appendix A for a detailed listing.

The text of this chapter quotes from these publications, verbatim if appropriate.

3.2 Architecture

This section presents our practical design for an architecture designed to be run on mass-market Intel TXT-enabled PC class hardware. The major components can be broken down into the following major building blocks:

- Secure Boot:
  This building block implements the platform boot process into a state defined by the platform administrator. The platform boots via Intel TXT, and if the resulting state is good, the TPM releases the keys for decryption of the main Base System.
- Base System:
  The Base System comprises the basic components an operating system needs to run and manage hardware platform resources, and naturally, it comprises also all support components for Trusted Computing technology. Furthermore, the Base System contains all components for management of

[1] acTvSM is the abbreviation for "Advanced Cryptographic Trusted Virtual Security Module". See also Appendix A for the relationship of the platform described in this chapter to the TvSM demonstration application partition.

user applications. A platform administrator may install any software package and integrate it into the Base System. The Base System is encrypted on platform storage media.

- Trust Management:
  From a file system point of view the Trust Management block is part of the Base System. Logically, the Base System contains "normal" operating system components, while the Trust Management components represent the trusted "extras" added. The Trusted Management block comprises all scripts and libraries required to manage platform Trusted Computing features and operations. Also, in evidence are the scripts to (re-)build the platform to a new trusted state (as defined by the platform administrator).

- Virtualization Partitions:
  The Base System can instantiate virtualization partitions for user software to execute. This is for two reasons: a) Isolation of user (application) code from the Base System. b) Measurement of a user application as one (or more) virtualization blocks, thus resulting in a short chain of trust length.

Figure 3.1 gives an overview of these major blocks and the execution flow between blocks. The following sections describe each block in greater detail. Subsequently, Section 3.3 focuses on platform operational and management aspects. The practical prototype implementation details follow in Section 3.4.

3.2.1 Secure Boot

The Secure Boot block is responsible for platform initialisation into a defined state and the deterministic platform early boot process, which are accomplished with Intel TXT features. From now on we assume knowledge, vocabulary and boot process of Intel TXT to be known—see Section 2.4.1.

The platform administrator defines a certain platform configuration state as trusted. Only when the platform boots component by component as expected does the Base System become accessible. The practical steps are as follows: TXT establishes a chain of trust (Section 2.3.3) and all components up to the Linux kernel and its accompanying initramfs[2] become part thereof. Then, the temporary in-memory file system initramfs contains additional code to unseal the key to decrypt the next block, which is the Base System: If the chain of trust up to this point is as expected, the TPM can unseal the key for decryption of the Base System. The Base System is measured into the chain of trust and mounted—and a normal Linux boot process continues. Otherwise, execution stops, as there is no alternative way to obtain the key to decrypt the Base System. For more implementation details see Section 3.4.

[2]The initramfs normally contains all code necessary for early boot of Linux. Device driver modules, file system mounting tools, etc.

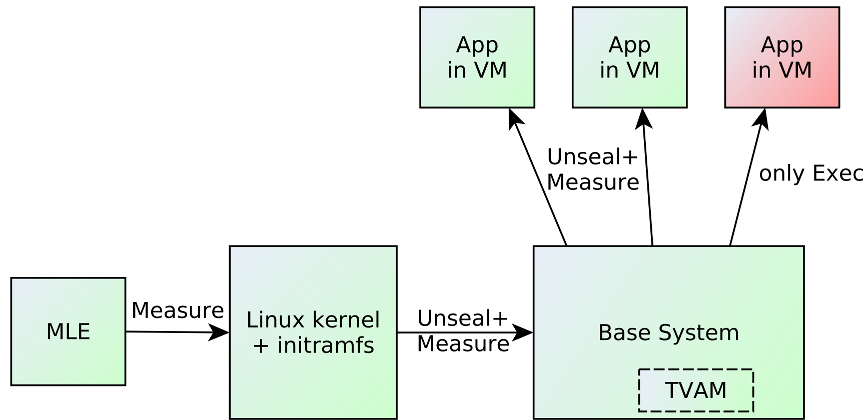


Figure 3.1: Overview of the execution flow of main platform blocks. Blocks earlier than the MLE of TXT are omitted—see Figure 2.3 for TXT details. Components which are measured before execution are shaded green, unmeasured are red.

3.2.2 Base System

The Base System includes the standard operating system basic components. Furthermore, it assumes the role of a hypervisor (Section 2.3.5) and manager of the platform's resources. It manages the hardware devices at the lowest-level and provides virtualization support services to run user applications in virtual partitions (Section 3.2.4). As a base component of the platform software in the Base System must be mature and actively maintained with regular security updates.

In order to enable a deterministic Base System image measurement into the chain of trust (so the encryption key for the Base System can be sealed to a certain platform state) the Base System is a read-only file system image. This conflicts with the requirement of a read-write file system for operation of a Linux-based platform. As a solution, an ephemeral file system merges at runtime with the read-only base file system (Section 3.4.2). Consequently, changes to the Base System do not survive platform reboot, except by explicit patching or system update (Section 3.3). This approach ensures the resistance of the Base System image to (malicious) modification.

3.2.3 Trusted Virtual Application Manager

The platform can run custom applications provided by users, which run in virtualization partitions (Section 2.3.5), so as not to interfere with other platform components. The management of the user-provided custom virtual applications is performed by a component called TVAM, which stands for *Trusted Virtual*

Application Manager. TVAM is the central component to manage user applications import, initialization, start-up, clean-up, etc., of virtual applications images. TVAM is a script that runs as part of the Base System and provides also a command line interface to the platform administrator.

Similar to execution transition from initramfs to the Base System, application images only exist in encrypted form on storage media. The process of running a virtual application is the same: If the platform is in a suitable state for the application, TVAM can unseal the key required to decrypt, measure, mount and run a virtual application.

Another core task of TVAM is the platform update process: If the platform administrator changes the Base System configuration and initiates platform rebuild, TVAM calculates the new future trusted state, prepares necessary boot policies and re-seals the Base System, applications, their data and associated image encryption key to the new state (for details see Section 3.3).

3.2.4 Virtualization Partitions

User applications which are not part of the basic operating system run in isolated partitions. The Base System (Section 3.2.2) operating system manages platform resources and provides a hypervisor (Section 2.3.5) to instantiate these partitions. TVAM (Section 3.4.1) manages the file system image and application start and stop.

An application image is essentially a full virtual machine, which may be an unmodified Linux or Windows or a special-purpose system. TVAM can start multiple applications in parallel, and it can extend, if configured to do so, the chain of trust to an application by the measurement of a virtual partition image, before it is started. Naturally, if stable PCR measurements are required, the image must be read-only.

3.3 Operating the Platform

This section now identifies and describes basic platform operations implementing platform installation, configuration and long-term maintenance.

3.3.1 Installation

The installation process constitutes initial transfer from installation media to platform storage media. Already at first boot of a freshly-installed platform the full chain of trust is operational and the platform Base System is only accessible if the boot process runs exactly as set up by the installation routine, which means no (malicious) platform modification has taken place between installation and first boot.

In order to start from an initial trusted state, the software needs to be distributed on a trusted medium. A root of trust for installation can be ensured by read-only media such as a CD-ROM. Once booted from the installation media,

the installation routine first wipes platform storage media and then installs a default basic platform configuration. Immediately, the measurement values for the MLE, the platform Base System image file, kernel and initramfs image are calculated by the installation routine, and required TXT boot process policies with these expected values are generated and stored in the TPM. After installation is complete the platform is ready to perform first reboot straight into a defined, trusted state.

3.3.2 Update and Application Mode

After the platform boots successfully into the Base System, it runs in *Update Mode*, where it waits for platform administrator maintenance commands. The administrator may connect from the network via secure shell (SSH). The update of policies in TPM non-volatile memory requires the TPM owner password. Consequently, before providing this TPM password, an administrator must confirm a) connection to the right platform and b) that the platform is in the correct Update Mode configuration. The first constraint demands that the SSH client verifies that the platform always presents the same SSH public key. The second is satisfied when the SSH server daemon private key is sealed to Update Mode.

If no external log-in attempt is effected within a defined timeout period, the platform automatically switches to *Application Mode*. TVAM performs the following: The SSH daemon is stopped and its private key is removed from memory. A PCR is extended to document the state transition and consequently prevent further access to the TPM sealed SSH key. Finally, in Application Mode installed applications are started by TVAM as configured.

3.3.3 Base System Rebuild and Resealing

At first installation the platform is in default configuration. A new configuration takes as basis the current one plus whatever updates an administrator performs during platform Update Mode.

After all updates or modifications are undertaken by the administrator, he initiates the platform re-build and re-sealing process. TVAM assembles a new Base System image and creates a new entry in the bootloader menu. If a component in the boot process has been modified, the VLP (kernel or initramfs change) and LCP (MLE change) are also updated. As all data blobs holding decryption keys for application partitions are sealed to the current (old) platform state, TVAM must re-seal all of them to the new one.

3.3.4 Small Updates

The full update procedure outlined above may be cumbersome for minor configuration changes such as a change of a static IP address in some configuration file. One possible solution is a "patch" facility, which allows the platform administrator to provide one or more patch file(s) at a specific file system location.

Upon boot they are automatically applied by the Base System. The authenticity of a patch is guaranteed by a cryptographic signature by the platform administrator—naturally, the validation key must be part of the Base System. Upon next full system update (and re-sealing) patches are automatically integrated into the system. This approach allows for small bug fixes and easy distribution of pre-configured system platform images, for instance in homogeneous data-centres, where machines only vary in small configuration details[3].

3.3.5 Application Management

TVAM provides a comprehensive set of commands for application image management. These enable a platform administrator to import, backup and delete virtual application images. The simplest way to install an application image is to import a raw disk image. Upon import TVAM creates an encrypted storage space and copies the raw data to it. The encryption key is sealed by TVAM to the Base System Application Mode state.

3.3.6 External Reboot

A method of forcing a reboot of the platform is an important feature of administration. Intel TXT capable platforms are usually equipped with a set of features called Intel Active Management Technology (AMT) [53], which amongst others allows externally triggered reboot as well as serial console redirection via a network connection. An externally-triggered reboot thus causes the platform to reboot into a state trusted by the administrator (Update Mode), regardless of the runtime state it was in before. Consequently, a reboot forcibly stops all user applications and provides the administrator platform access.

3.4 Implementation

As proof of concept of the architecture outlined in the previous sections we assembled a platform prototype and required management operations. The following subsections give an in-depth description of prototype implementation details and implemented storage management concept.

3.4.1 Components

In theory, any Linux distribution can be customized for our platform architecture, but in practice, a substantial number of patches and changes are needed, to multiple software packages, in order to assemble a proof of concept prototype.

[3] For example, identical hardware, though every machine gets a different static IP

Platform Boot Process

Secure boot is accomplished with standard bootloader GRUB [45] along with `SINIT` and `tboot` [54]. `SINIT` is Intel's implementation of an ACM, while `tboot` is Intel's public [57] open-source prototype implementation of an MLE. For a description of the tboot boot process before the Linux kernel assumes control see Section 2.4.2.

The Linux kernel passes execution to the `initramfs` code. We added to the `initramfs` a customized 64-bit port of the tools from IBM's *TPM-utils* [105, 52]. These tools provide the PCR extend and unsealing capabilities in the initial ramdisk (`initramfs`) environment.

The Base System software packages are from the x86_64 *Debian Linux* lenny release [27]. This is a pragmatic choice for a robust base system, as Debian is known for its emphasis on stable code base, frequent security releases and conservative approach to adding immature features. To support Trusted Computing and virtualization hardware we complemented Lenny with selected packages from the newer Debian testing tree. For example, only Linux kernels 2.6.32 or newer integrate Intel TXT support and drivers for chipsets that implement it. We customized installation, initial ramdisk management and rebuilding of the Base System image from Debian to our needs. The system bootstrap scripts to create distributable and bootable initial installation CDs originate from GRML Linux [47].

Runtime Architecture

For an overview of the platform runtime architecture see Figure 3.2. A customized Linux kernel with the *Kernel-based Virtual Machine* (KVM) [63] module enabled provides the lowest-layer hypervisor functions. KVM isolates multiple virtual machines on x86 CPUs equipped with hardware virtualization support. The Base System with TVAM is a minimal Debian Linux. While KVM manages the low-level hardware, input/output devices for the virtual machines are from QEMU [12]. QEMU can emulate all standard hardware devices of a x86 platform for a virtual machine (=a chunk of memory isolated by KVM). QEMU itself runs in the Base System. The Base Systen can host/support multiple virtual machines with their own operating system and applications. QEMU may provide a TPM device to one virtual machine and forward the commands to the real hardware TPM.

Trusted Virtual Application Manager

The implementation of TVAM in the Base System is scripted in the Ruby language [152]. TVAM implements the operations to create new trusted platform states and re-seal the Base System and applications to them. Also, TVAM adds and removes platform states from the list of trusted states in TXT boot process launch policies. Furthermore, TVAM enables the platform administrator to import an application image into the platform, start, stop, list and remove virtual applications. See Figure 3.3 for a list of commands in TVAM v0.3. For

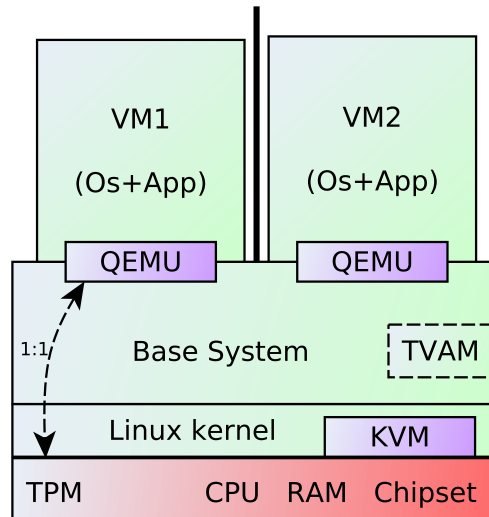


Figure 3.2: Platform prototype runtime architecture. KVM manages low-level resources for the kernel and provides virtualization and isolation. QEMU emulates devices for VMs, while running in the Base System. TVAM is the manager for base system configuration, start-up and tear-down of VMs, and manager of hard disk space. The hardware TPM can be forwarded to only one VM by QEMU.

more complex operations, such as trusted platform state calculation, sealing, unsealing, policy creation and storage in TPM non-volatile ram TVAM depends on sub-calls to *jTpmTools* and *jTSS* [92].

After boot, PCR 17 holds the LCP and details on the MLE [56]. PCR 18 contains the measurements (hashes) of the MLE, the kernel module and the respective command lines. PCR 19-22 contain the elements described in the VLP, the modules defined in the bootloader configuration (kernel, `SINIT` module, `tboot` binary and `initramfs`). The read-only image of the Base System is extended during boot to PCR 14. PCR 13 is used to measure virtual applications by TVAM. PCR 15 indicates the transition from Update to Application Mode. All these PCR states are used to seal data, especially file system encryption keys, to the expected platform states in the different platform execution phases.

3.4.2 Disk Storage Management

For a practical chain of trust the number of measurements should be small and the measured components few. Thus, for our prototype we use a sophisticated disk layout with different types of file systems to achieve a practical implementation.

```
# tvam

    TVAM - Trusted Virtual Application Manager
      import        - import an application
      remove        - remove an application
      remove_state  - remove a system state
      resetvlp      - reset VLP in TPM nvram (DANGEROUS!)
      seal          - seal basesystem to a new state
      show          - show informations about installed applications
      start         - start applications
      stop          - stop a running application
      tmp_create    - create a crypted temporary storage
      tmp_remove    - remove a crypted temporary storage
      version       - show version information
```

Figure 3.3: All Trusted Virtual Application Manager (TVAM) commands.

The on-disk structures are created and maintained by TVAM. On assumption of hard disk installation Figure 3.4 depicts an overview of hard disk partitions, logical volumes, file systems used and their encryption status. Since this figure may at first appear complex, we now describe each of these pieces in detail:

Partitions

System hard disk `/dev/sda` contains two partitions:

- The first partition (`/dev/sda1`) is a standard Linux read-write `ext3` file system. On this small partition are all components necessary for platform boot. These are: GRUB bootloader, Intel's `tboot`, Intel's ACM (SINIT) and the Linux kernel image plus `initramfs` image. This partition is mounted as `/boot`.
- The second partition is `/dev/sda2`. All available space is dynamically managed by the Logical Volume Manager (LVM) [151] of Linux as a number of logical volumes (LVs).

Logical Volumes

Storage space of the second partition `/dev/sda2` is managed with logical volumes (LVs) (Section 2.7.1). Some LVs are in plaintext and some are encrypted via LUKS (Section 2.7.2). Our prototyp uses the following volumes:

- Storage logical volume – STORELV:
  Logical volume STORELV contains TPM sealed data blobs constituting the access secrets for encrypted logical volumes. Furthermore, STORELV contains the files for patching (Section 3.3.4). STORELV is mounted as `/store` on the platform. This logical volume is unencrypted.

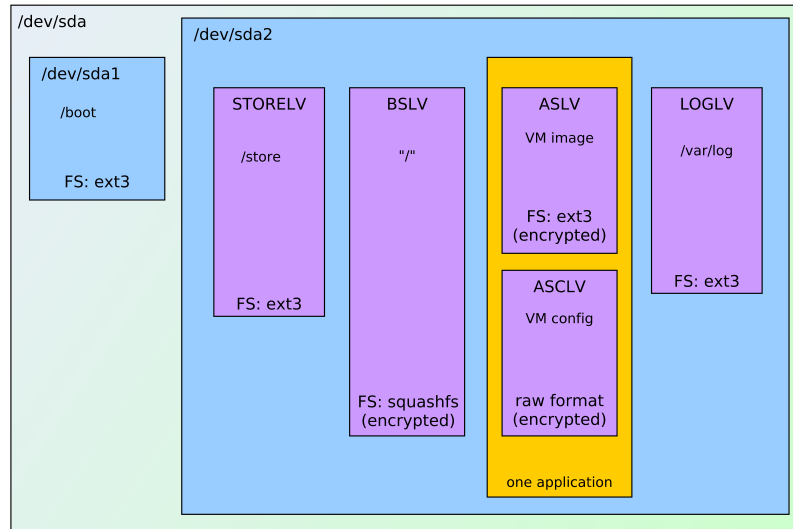


Figure 3.4: Detailed platform disk layout. The two hard disk partitions are blue, while logical volumes are violet. One application (orange) typically consists of two logical volumes. Logical volumes are host to different file systems (FS) and may be encrypted.

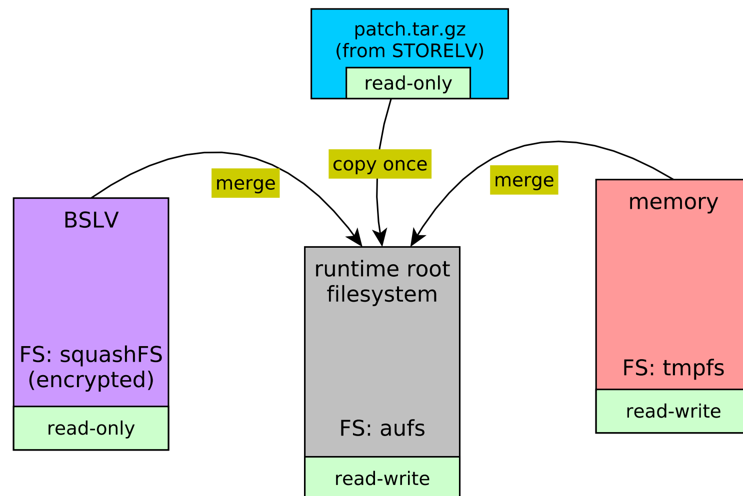


Figure 3.5: The platform (middle) writeable main root file system is assembled at boot time from read-only Base System image (left) and an in-memory temporary file system (right). A patch (top) may also be integrated.

- Base System logical volume – BSLV:
  BSLV contains the compressed Base System read-only image of the root ("/") file system (binaries and configuration files). The file system is squashfs, which is a file system specially optimized for such read-only applications. In the platform boot process this is the Base System image measured by the `initramfs`.

- Application Specific logical volumes – AS(C)LV:
  One application virtualization image is stored in one Application Specific Logical Volume ASLV. The associated configuration is stored in an Application Specific Configuration Logical Volume ASCLV. Both are managed by TVAM and are required by TVAM to run one application.

  In the scenario of a service that encompasses multiple applications running in parallel, e.g. a webservice front-end, a database partition and a firewall virtual partition, configuration files for such an application group may be linked, so TVAM starts them together.

- Logging logical volume – LOGLV:
  LOGLV stores platform activity logs. Naturally, as this logical volume is unencrypted, an administrator must consider what logging activities should be enabled. During platform development and debugging it is of course very useful to log as much as possible for subsequent later analysis.

Runtime Root File System

At runtime a Linux platform requires a writable file system, and root "/" Base System from the read-only BSLV is not enough. Our solution for this is that the actual runtime root file system of the platform is assembled from three sources:

- Base is the read-only root file system as available from the Base System logical volume.

- Patches found on the StorageLV (STORELV) are overlaid on the Base System once during platform boot.

- Finally, a read-write temporary file system (tmpfs) is instantiated in system memory. Modern PCs have sufficient main memory for an in-memory temporary file system.

The three layers are merged at platform boot with help of the advanced multi-layered unification file system (aufs) [59]. Figure 3.5 illustrates this process. All writes go to the in-memory file system and are lost upon next reboot, unless the platform administrator initiates a re-build and re-sealing of the current platform state (see Section 3.3.3)

3.4.3 Performance

Prototype implementation and testing were done on a HP dc7900 machine with an Intel Core 2 Duo E8400 CPU clocked at 3GHz, 4GB RAM, Intel Q45 Express Chipset and Infineon TPM 1.2 as reference platform.

Prototype Base System installation image implementation size (`.iso` file) is approximately 400 MB. Installation and setup of the encrypted file systems on such a machine complete in just under 15 minutes. Prototype platform measurement and integrity-enforcing mechanisms are only performed upon boot, which takes 57 seconds from power-on to display of the login-prompt. The delay added by the TXT late-launch at boot time is about 5 seconds.

Other than that, Base System runtime performance does not vary from that expected from a normal Linux platform with similar configuration.

3.4.4 Shared TPM Access

The Base System needs the hardware TPM only at start-up. Afterwards it can be assigned to one VM application partition via QEMU. This required a modification[4] of QEMU to allow TPM command byte stream forwarding from/to the `/dev/tpm` device in the Base System in a non-blocking manner. Consequently, one virtualized partition can run Trusted Computing applications at a time.

More general approaches have been proposed to provide access to the hardware TPM device to multiple virtual machines at the same time via multiplexing, for example virtualized TPMs [14, 108, 136].

3.5 Platform Security

The previous sections gave an overview of platform working. In this section we reflect on platform security features and limitations.

3.5.1 Attacker Model

As our platform is based on TCG's TPM and Intel's TXT technology, the familiar limits of their security, benefits and possible attacks apply (see Section 2.6.1). In the case of our specific platform attackers may attempt

1. To manipulate the boot process, so the platform does not reach the intended administrator-defined state

2. To manipulate applications at runtime to obtain elevated runtime privileges

3. To manipulate the Base System and from there applications

[4] Which meanwhile has also been integrated as a standard selectable feature in QEMU, and it is now available in the official stable QEMU version.

...in order to gain access to applications and their data (=assets) executing on top of the base platform. We place full trust in the system administrator to be careful with secrets, backup secrets and key material of the platform and of applications, thus he is not the weak link compromising sensitive data.

3.5.2 Discussion

- Attack 1: Manipulation of the TXT-based chain of trust requires malicious manipulation of the ACM, tboot as MLE or Linux kernel and initramfs. Only if measurements of all these components are correct, does the TPM unseal the Base System key. Thus, a maliciously-modified boot process fails at the unseal operating via the TPM. Therefore, our platform architecture ensures that a defined Base System is reached after boot, as defined by the platform administrator.

- Attack 2: In each virtualization partition a normal operating system typically runs, with a kernel and user application permission model. Security history suggests that any application can be exploited and controlled. Malicious elevation of application privileges to system privileges (root) may follow, since a manipulated network connection may obtain root privileges in a virtual machine. Horizontally in the software stack virtual partitions are contained by the Intel hardware virtualization features that isolate memory areas. Vertically QEMU provides device emulations for input/output data exchange. An attack may use a bug in a QEMU device emulation to break out from a virtual machine into the Base System, where QEMU runs as normal user application. Naturally, the more devices are emulated by QEMU, the higher is the possibility of a bug.

- Attack 3: If an attacker succeeds in obtaining Base System root privileges, access and modification of the encrypted application (data) partitions currently mounted at that time are possible, as the required key material is available in kernel memory. However, such a security breach cannot permanently modify the Base System. The update process of the Base System to a new configuration requires a rewrite of the policies in the TPM, which can only be performed with TPM owner's permission. However, the TPM owner password is not stored on the system. The next reboot discards all Base System changes and restores a trusted state (or fails, if the boot process was modified).

3.6 Summary

The prototype platform described in this chapter demonstrates a practical architecture suitable for mass-market off-the-shelf PC hardware. We have used TPM and TXT features to provide a general-purpose platform that offers integrity guarantees for the base operating system layer and applications and services

running on top. More precisely, our platform has achieved the following properties:

- The use of Intel TXT and its TPM stored policies enables a platform administrator to enforce what configuration is allowed to boot via TXT.
- The change of platform configuration requires changing the policies stored in the TPM. Only the platform (=TPM) administrator can change these and the TPM provides protection against brute-force password guessing.
- The Base System and user data and applications running on top are protected via encryption at rest. Only if the platform attains the platform administrator defined state, do they become accessible.
- Furthermore, Intel TXT shortens the chain of trust by omitting early BIOS and boot process measurements. There are only a few components in the boot process up to the Base System, thus the chain of trust is so short that it can be practically audited in a remote attestation scenario.
- The chain of trust for our platform is so short that TVAM can precalculate the new future platform state at next reboot. This capability of TVAM allows our platform to transition from the current state to the new one. TPM sealed data blocks are resealed by TVAM to a correctly predicted future state.
- The choice of Linux as basis for our prototype essentially allows installation of any Linux application. We show a common general purpose operating system can be adapted for Trusted Computing.
- The choice of KVM+QEMU as virtualization technology allows a free choice of guest operating systems and at the same time isolates memory areas of virtual partitions by hardware mechanisms.
- The central administrative tool TVAM empowers platform administrators to manage user applications (images) easily.
- From a software perspective, all components (except the ACM provided by Intel) are open-source. This ensures accessibility to security inspection and the ability to customize the platform according to specific needs.
- Necessary components for implementation of this architecture are available, and this includes mainboards capable of Intel TXT and Intel AMT, as well as processors capable of hardware virtualization. These solutions are off-the-shelf hardware not carrying a price premium.

4

Lightweight Distributed Attested Clouds

4.1 Introduction

Chapter 3 explored modification of a general-purpose off-the-shelf Linux operating system in order to achieve multiple security and usability goals. The implementation prototype achieved these goals, however at significant cost. Implementation was long and tedious[1], and overall the platform came to involve a significant degree of complexity: The goals were, amongst others, enforcement of a platform boot process into an administrator-defined platform state, creating a practical short chain of trust ensuring platform user data is only accessible in defined states, and administrative processes to manage platform transition from one trusted state to another.

By contrast, this chapter eschews the scenario of a full-size Linux system on which any user program may be deployed, but we still strive to achieve a security-enhanced, attestable platform. We explore a more focused scenario of lightweight attestable networked "cloud" processing nodes:

- First, the basic scenario assumption is a generic network-connected architecture. The processing nodes are distributed widely, possibly covering diverse physical sites, operators and hardware platforms. The idea is that "anyone" should be able to become part of the processing network, the cloud.
- Secondly, individual cloud nodes actively use Trusted Computing technology in the cloud formation phase. This provides remote assessment and

[1] Which one may also deduce from the number of publications reporting interim results of this effort.

reporting support for platform state. Platform state is interesting for security reasons: The ability to determine platform state is a requirement for newly-connecting nodes to proof that they are in an acceptable state and for existing nodes for later verification. The cloud aims to force all participating nodes to be in a given acceptable platform configuration.

- Thirdly, while commercial off-the-shelf x86-based platforms with Trusted Computing enhancements are mass-market available nowadays, in the near future there may be other options. One alternative is ARM architecture-based server designs, prototypes of which are already being evaluated by early adopters and friendly purchasers[2]. Consequently, consideration of an implementation prototype covering future ARM platforms as processing nodes provides feedback on practicability of the approach taken.

The prototype described in this chapter has never been released to the public, however architecture, protocol, experience and lessons learned have been previously published. See Appendix A for detailed references.

The text of this chapter quotes from these publications, verbatim if appropriate.

4.2 Scenario

In this section, we first enumerate core assumptions and properties to be pursued, then we describe interacting entities and their role.

4.2.1 Cloud Node Properties

In our architecture design and implementation prototype we aim for the following core properties:

Distributed Individual cloud nodes may be distributed geographically and organizationally. Nodes may be placed in different countries and continents, and may be owned and operated by a diverse sets of operators.

Attested It should not be possible to join the cloud of processing nodes at whim. The cloud network should be able to assess a new applicant node through the use of Trusted Computing. Upon joining the network every node has to proof what state the platform is in through remote attestation. The result of this attestation process is the decision whether a node is in a trusted state and therefore allowed to become part of the cloud.

[2]The motivation is that the architectural differences between ARM vs. x86 promise to allow more CPU/space density and efficient power use for the ARM platforms.

Protection of data at rest When a cloud node uses a local data cache in order to perform tasks, data in this cache should be protected at rest. The data is encrypted with a key only available in the identical runtime state the platform was last in when it was processing data.

Lightweight In order to enable a diverse set of stakeholders to participate in the cloud network, setup, joining and maintenance processes for individual nodes should be simple and not involve significant organisational overheads.

Heterogeneous Many nodes should be able to join the cloud. Consequently, the cloud node software should build on a base and primitives easily portable to different platforms. The larger the cloud network, the more resources may be shared among participants. Consequently, this allows processing of larger tasks and improves economics of scale, which reduces costs.

4.2.2 Entities

A cloud infrastructure connects individual computing nodes, providing coordination among nodes. We propose a central controlling entity for commissioning nodes, to perform accounting and oversight tasks. We identify two different kinds of entities in our scenario:

Cloud Control

Cloud Control (CC) is the central entity responsible. It provides information services on available cloud nodes and service capacity. Furthermore, it manages cloud client profiles and is responsible for resource consumption accounting, authentication and authorization. As Cloud Control is central to the operation of the cloud, core services/servers may probably be best run in a professionally managed datacentre available 24x7.

Cloud Nodes

Our scenario is not restricted to the few processing nodes operated locally by the Cloud Control provider. Instead, we expect the vast majority of nodes to be distributed remotely, with their operators not necessarily under direct supervision.

The assumption is that essentially anyone can offer nodes to join the cloud and thus provide computing capacity[3]. However, to do so they must first evidence a certain level of security in their cloud-joining platform. Trust in a platform is not established by reputation or contracts, but instead cloud nodes must apply and run platform system software that is well known to and trusted by central Cloud Control. This hurdle protects against fraud and abuse, and the use of Trusted Computing security features enforces this requirement.

[3] And then may be compensated for doing so under a service contract.

In practice, this means that a cloud node software image must be downloaded from central Cloud Control and then booted at a node. If a newly joining node's platform state is successfully attested to Cloud Control, the node becomes available to the cloud. If a node wants to run custom or modified platform software, it is blocked from joining by Cloud Control.

4.3 Node Operation

This section examines core node operations, states and management flows. Of special interest is the security of the cloud formation process when distributed remote nodes join the cloud network via central Cloud Control on the requirement that every node hosts a TPM as specified by the TCG (see Section 2.2.2).

4.3.1 Cloud Node Joining Cloud Control

A new cloud node wants to join the cloud network. The cloud node software image boots via a measured boot process: measurements of platform state and components are documented in the TPM PCRs, starting from a known initial platform state. Furthermore, the image contains the unique asymmetric RSA public key CC_{pub} and the network address of Cloud Control (CC). As the first operation immediately after boot, the node attempts to join the cloud.

The following steps 1 and 2 in our joining protocol are a modification of the original TCG-specified AIK-PrivacyCA exchange[4]. For space reasons we cannot explain every detail of the original protocol[5], so we concentrate only on the data fields we use here. After steps 1 and 2 conclude, we add steps 3 and 4 for exchange of platform state.

Figure 4.1 provides an overview of the joining protocol. Four messages are exchanged between joining node and central Cloud Control:

1. The first objective is to establish a secure connection to Cloud Control. The first data blob sent to CC is symmetrically encrypted with a fresh random symmetric key K, while K itself is encrypted asymmetrically with CC_{pub}. The request data blob contains the EK certificate (EK_{cert}) of the TPM, along with the public key AIK_{pub} of a newly created AIK keypair in the client's platform TPM. We also sent a data field "platinfo"[6], whose purpose is to provide supplemental information about the platform to Cloud Control, if needed. This information may be whether the platform is x86- or ARM-based, or authentication information, but this is scenario-dependent.

[4]See Section 2.5.2 for a description of the original TCG-specified AIK creation and deployment flow.

[5]Which is rather complex and requires understanding of TPM commands, data structures and TSS API.

[6]In practice, we (ab)use the data field that normally carries the name for the AIK certificate to be issued for the PrivacyCA.

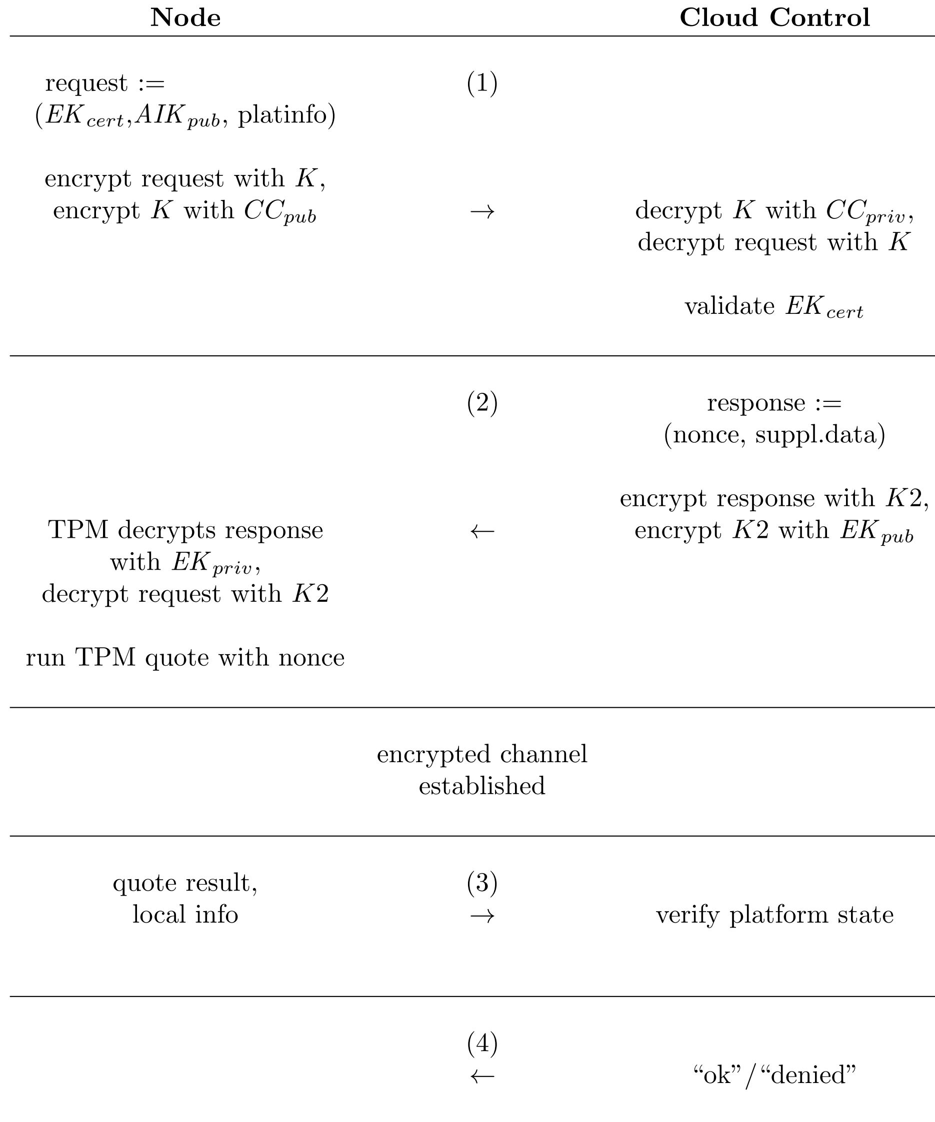


Figure 4.1: Node-to-Cloud join protocol
(notation simplified for clarity)

Cloud Control receives the blob and decrypts the payload with its secret CC_{priv}, thus obtains K and can then decrypt the request. A certificate validation process runs for the received EK_{cert} to determine whether the TPM is a valid hardware TPM.

2. On successful validation, Cloud Control generates a fresh nonce and further supplemental data necessary for requesting a quote of the remote node platform state. For example, the bitmask of PCRs to quote depends on the hardware platform reported in "platinfo" of the request in step 1. Again, this is scenario-dependent. The response data blob is encrypted with a new random symmetric key K2 and K2 is asymmetrically encrypted with EK_{pub} (contained in the certificate received in step 1).

   When the client platform receives the response, it uses the TPM to decrypt the response to obtain K2 and then with K2 to decrypt the payload. The client runs a quote of the platform's state with the fresh nonce received, and the TPM uses the previously generated AIK to sign the current platform state.

From now on we assume the symmetric key K2 created by Cloud Control is used for the encryption of all further exchanges with Cloud Control[7].

3. The TPM signed platform state along with additional node or platform specifications (e.g., available storage or processing power) is sent back to Cloud Control.

   Cloud Control now validates the presented TPM signed platform state. The quote must be signed with the AIK presented in step 1, and the nonce must be identical to the one sent back in step 2. The node platform state reported must be well-known to Cloud Control, i.e., a trusted cloud node software image must have been booted on the platform.

4. Cloud Control now either welcomes the node to the cloud or denies access.

Discussion

The first two steps in this protocol follow the standard exchange for AIK certification with a PrivacyCA to establish a secure point-to-point connection from Cloud Control to the platform (hardware TPM) without a man-in-the-middle. Cloud Control does the same as a PrivacyCA would, it checks whether the TPM is really a hardware TPM, according to the EK certificate presented. Using the public EK to encrypt the response ensures that only a specific TPM can decrypt the response, consequently there is no man-in-the-middle. Furthermore, Cloud Control remembers the non-migratable AIK presented as the runtime identity of the platform.

The following steps 3 and 4 run remote platform attestation. The fresh nonce of Cloud Control ensures that the response is up-to-date and the AIK on the

[7] Or, alternatively, TLS [29] is another option.

platform quote must be identical. Consequently, to Cloud Control this is proof the remote platform is in a trusted state and is welcome to join the cloud.

After this initial joining protocol any higher-level cloud software platform can run, and jobs are assigned to this processing node. Depending on the actual scenario and applications Cloud Control may enforce additional restrictions. For example, security sensitive applications may require node operators and their platforms to register as a first step and to perform an additional authentication step when a platform joins the cloud.

4.3.2 Node Update / Cloud Rejoin

The joining process for a new node represents the basic construction of a cloud network, from individual nodes as outlined in the previous section. Yet, there are two more basic operations to consider:

Update

Eventually, the cloud's requirements for the node software image may change. Consequently, Cloud Control refuses an obsolete version, now untrusted, in step 4 of the join protocol. The cloud node operator must obtain an updated software image and retry.

Rejoin

If a cloud node reboots (for whatever reason) and wants to *rejoin* the cloud network with the identical cloud node software image, this raises the question of persistent node data. If the working dataset is very large, perhaps gigabytes of data, then it is probably inefficient to re-synchronize all data with the cloud over the internet again. Rejoin is faster, if there is a node local storage for the working data persisting over a platform reboot.

4.3.3 Node Local Storage

Temporary storage on node-local mass media (e.g., on a hard disk) must be encrypted to protect data at rest (Section 4.2.1). One solution for fast bulk encryption of data is a symmetric approach, e.g., encryption based on AES. In order to re-access the local storage after interruption (e.g., platform reboot) the key for local storage must be accessible only to the identical cloud node software image running at the identical cloud node. This problem can be solved by encrypting the symmetric storage key with an asymmetric TPM key sealed to a specific cloud node software image state[8].

We identify three data items enabling an automatically (re-)bootable, rejoining cloud node with persistent, secure, local cloud node data storage:

Storage—image file or partition
The local encrypted storage on mass media.

[8]This is basically the same approach as the file system protection in Chapter 3.

Bulk storage encryption/key

If a cloud node boots for the first time local storage does not yet exist. Upon successful completion of the cloud join protocol, local encrypted storage is created and initialised. A new, random encryption key protects the data, and the key is TPM-sealed to current platform state.

If the identical cloud node software image boots on the identical platform, in step 3 of the join process the node looks for local storage. If a storage is found, the access key can be unsealed, as the platform is in the same state again. Local information discovered is reported as supplemental "local info" to Cloud Control (in step 3) and is proof of the previous state of work performed on this node. Consequently, a local storage can significantly speed up rejoining of the cloud.

TPM ownership password

In order to boot a new software image version on a cloud node, the TPM ownership password must be entered at least once. This requirement stems from creation and activation of the AIK keypair in steps 1 and 2 of the cloud join protocol requiring consent of the TPM owner[9]. It is a sensible policy that the owner should consent to software to be run on his machine.

Consequently, in order to automate the reboot/rejoining of a platform to the cloud, one solution is that during first boot the ownership password is hashed and also TPM-sealed to the specific software image booted. The sealed blob with the ownership password is then only accessible to the same (trusted) software image during joining when it is needed. Storage of the ownership password on the platform goes against the idea of Trusted Computing that a platform owner is thought to explicity consent every time to certain sensitive TPM operations, however, it enables the automation of platform operation, after the owner has once consented to a new platform software image version.

In our proposed solution for local storage three pieces of data together enable a local, persistent, encrypted data storage over (automated) node reboots. As the three pieces of data belong together, we propose to integrate them into one image (file or partition) for easy maintenance.

4.4 Implementation

4.4.1 Overview

As proof of concept of the architecture presented in previous sections, we prototyped the core chain of trust in our architecture.

[9] See, e.g., TPM v1.2 specification, Part 3, command *TPM_ActivateIdentity*: "Only the Owner of the TPM has the privilege of activating a TPM identity. The Owner is required to authorize the TPM_ActivateIdentity command"[139].

In order to provide a chain of trust and platform attestation support, every platform needs a TPM. Our prototype implementations use Infineon TPMs, which come with an on-chip EK certificate issued by the manufacturer. We chose as operating system Android, as it is by design split into a read-only base system image and separate read/write areas for runtime data. This greatly simplifies the modifications needed for adding the required measurement hooks for a chain of trust into the Android start-up process[10].

There are two test platforms, one x86-based and one ARM-based. Android runs on both with a Linux kernel, which provides a standard char device as interface to the Infineon TPM. The software environment on top is the Java programming language, a natural fit in the Android environment. jTSS [92] provides the Trusted Computing library support to access TPM functions, which is a modified version of the initial Android port performed in [163]. The prototype cloud join protocol is a modification and extension of the original AIK cycle code implemented in jTpmTools [92].

The following two sections describe implementation details and differences, x86 vs. ARM, for each cloud node platform prototype. The Cloud Control server side simulation runs on a generic PC.

4.4.2 x86 PC Platform

The x86 development platform is an HP Elitebook 8440p, which is Intel QM57 chipset-based and fully supports Intel TXT. The on-board TPM is an Infineon TPM v1.2, firmware rev 3.17.

On the PC platform the BIOS controls the master switch to enable or disable Trusted Computing features on the platform. So setup requires going into the BIOS and enabling the TPM and TXT. The default "`ANY`" TXT launch policy preloaded into every TPM by PC platform vendors suffices and does not need modification. A boot process performing the chain of trust measurements is sufficient, and the enforcement of specific boot/launch policies is not needed, as implemented with the platform presented in Chapter 3.

The platform owner needs to take TPM ownership once to create the storage root key (SRK) in the TPM. The SRK password is the common well-known secret, namely all zeros. The take-ownership function may be performed with the installed OS or can be provided as a separate utility function on the cloud node software image.

Storage and Boot Process

In terms of proof of concept prototype we use USB-connected flash memory as medium for mass storage, as it is very simple to handle and cheap and allows for experimentation with various software images and configurations at little cost. Most importantly, with this approach the main platform hard disk is

[10] Compared to the full-size Linux solution of Chapter 3 with its significantly higher implementation complexity.

not modified in any way, fulfilling our goal of lightweight deployment, so no installation is needed and there is no worry about data on the system.

The content of the drive is a single partition and the file system structure is quite simple. As bootloader we use Syslinux [154], which sits in the bootsector and requires *ldlinux.sys* and *syslinux.cfg* as support files. *mboot.c32* and *menu.c32* display a menu and perform multiboot kernel chainbooting. A TXT boot requires *tboot.gz*, the trusted boot reference implementation provided by Intel [54], and a set of SINIT ACM modules [57], the chipset specific initialisation code, also provided by Intel. The Android base OS software is a modified image of Android-x86[11] of the Android x86 porting effort [3]. The Android system consists only of the *kernel*, *initrd.img*, *ramdisk.img* and *system.sfs*.

The *syslinux.cfg* configuration file connects all pieces together. *Menu.c32* specifies as primary kernel *mboot.c32*, which runs *tboot*, which starts up TXT with the proper SINIT ACM module. Control reverts to *tboot*, which executes the Android Linux *kernel*, who with help of the *initrd* and *ramdisk* starts the full system from the *system.sfs*.

In summary, the chain of trust is: The SINIT ACM measures *tboot*, which measures *kernel*, *initrd* and *ramdisk*, while the script in the *initrd* measures the *system.sfs* image.

Android OS

Modifications to the Android base system to implement a trusted boot process on a x86 PC are as follows: We replace the default kernel with a more generic kernel to support all common x86 hardware, including the TPM and TXT drivers. This increases the *systems.sfs* from 79 MB to 97 MB. Furthermore, we modify the *init* script of the *initrd* to load the TPM *tpm_tis* kernel driver and measure the *system.sfs* Android base system into a PCR. This measurement operation needs to load the full system image once, however, due to small size (see above) and the fact that modern USB flash drives perform at read speeds of more than 20 MB/s[12], and this delays the boot process by only a few seconds. Then, initrd mounts the system partition and the file system root moves to it. The *TPM_extend* utility binaries for measurements increase the *initrd* size by about 700 kb. Consequently, the impact on boot process performance and binaries sizes is negligible.

Application

Android package *cloud_node.apk* contains our test code. Immediately after Android boot-up finishes, we install this package with the help of the Android Debug Bridge (adb)[13] and execute our test code. The Java-based test client

[11] Release 2.3 RC1 eeepc (Test build 20110828)

[12] Newer platforms have USB 3.0 capable ports and flash drives already perform at over 100 MB/s read speed, so we expect the delay in reading the full file image into main memory to become less.

[13] Android Debug Bridge (adb) is a standard tool in the Android SDK.

accesses the TPM under Android and runs the cloud join protocol with our Cloud Control server simulation—client and server are connected by a standard ethernet network.

4.4.3 ARM Platform

Primary hardware reference platform for ARM implementation of our proof of concept prototyping is a Freescale iMX51 evaluation kit [38]. Figure 4.2 is a photo of our setup.

The Freescale MX515D application processor found on this board is based on ARM's Cortex-A8 core and would offer advanced security features, which include ARM's TrustZone security extensions and a set of secure boot facilities. Unfortunately, most parts of the documentation describing the processor are only available under NDA from Freescale. For this reason, we do not and cannot use these advanced hardware security features in our prototype implementation, because focus is to implement a TPM-based chain of trust.

As operating system software we use Android, in comparable configuration to the x86 platform. We use the open-source U-Boot [28] bootloader to load and boot the Android kernel, initial ramdisk and file system images from a removable SD card. With the iMX51 board the built-in SD card boot facilities of the processor's integrated boot ROM starts the U-Boot bootloader. Consequently, this does not require a modification of any of the board's flash memory chips.

TPM Support on ARM Platforms

The ARM platform we use as a prototyping platform does not include a Trusted Platform Module. Moreover, this platform does not provide a Low-Pin-Count (LPC) bus, to connect a standard TPM intended for integration on desktop x86 platforms.

Consequently, for the purpose of this prototype a simple LPC bus adapter is needed to attach a TPM to the ARM platform. We use an adapter based on a common off-the-shelf FPGA development board. The FPGA board we use as an interface adapter contains a Cypress-FX2 microcontroller with USB interface as well as a small Xilinx Spartan 3E FPGA. The block diagram in Figure 4.3 gives a high-level overview of the hardware setup.

The FPGA can be set-up with an an LPC bus controller block and exposes a serial peripheral interface (SPI) bus interface. This allows the FPGA-based LPC bus controller, and consequently any TPM attached to it, to be connected to virtually any microcontroller or microprocessor system. The FX2 microcontroller hides the low-level details of the LPC bus protocol and offers a convenient USB interface instead.

TPM-based Chain-of-Trust on the ARM Platform

On PC platforms, Intel TXT depends on a series of modifications to platform hardware, firmware and CPU microcode to implement a fully measured, well-

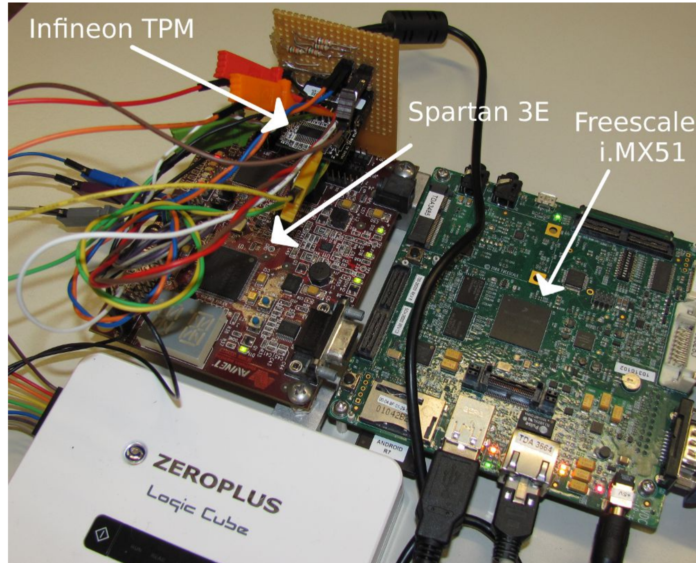


Figure 4.2: Experimental hardware setup.

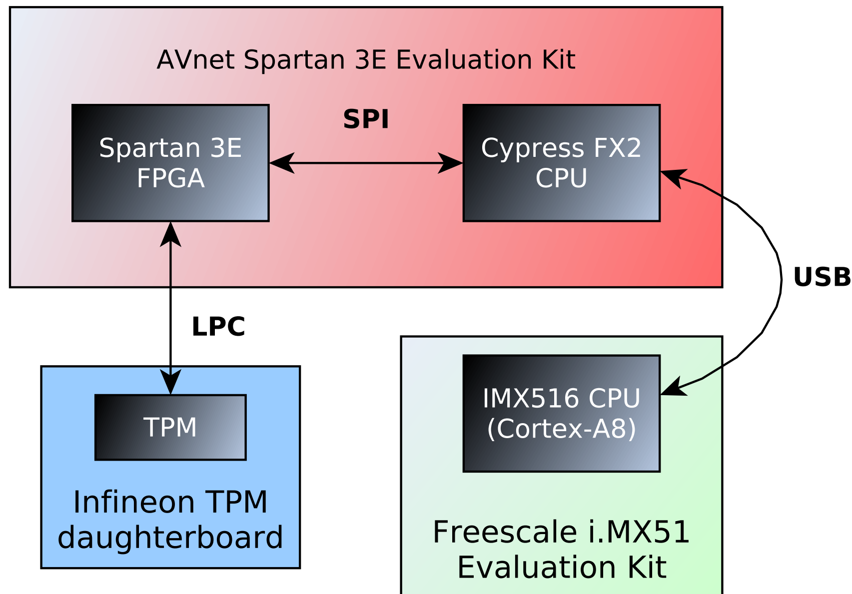


Figure 4.3: Attachment of a LPC bus-based TPM to Freescale ARM platform through an intermediary board for bus conversion.

defined system state. Common ARM platforms do not provide a direct functional equivalent to Intel's Trusted Execution Technology out of the box. However, they do provide the required building blocks to construct a system with comparable security capability.

The PC version of our node uses TXT to construct a trusted boot chain for the base system loaded from a removable USB thumb drive. The ARM implementation of the cloud node software boots from a removable SD card. This SD card contains the actual bootloader (U-Boot), the Android Linux kernel, an initial RAM-disk and the actual Android root file system, similar to the x86 platform. The mechanism used to load the boot-loader from the removable SD card depends on the hardware platform being used. Here, the iMX51 evaluation kit on-chip boot ROM performs a direct boot from the SD card. Once the U-boot boot-loader has been loaded from the SD card, the remaining platform boot process can be customized.

The modifications to realize a TXT-style measuring boot process on the ARM platform are as follows: The initial RAM-disk of the Android boot image includes a system-level service for interfacing with the USB-to-TPM adapter discussed above. This "Android TPM access service" is a native application, to allow its inclusion at a very early stage of the platform startup phase, before the standard Android runtime environment has been fully initialised. The service takes care of initialising the TPM interface hardware. Moreover, it is responsible to perform the initial PCR extend operations for the construction of a chain of trust.

The approach outlined for bootstrapping a chain of trust on an ARM platform suffers from one obvious problem: From a Trusted Computing perspective the TPM access service is the first component to communicate with the TPM, taking the role of the core root of trust for measurement (CRTM). Without additional support from underlying hardware and on-chip boot ROM there is, however, no (hardware) guarantee that initial measurements were actually performed by the intended CRTM. In order to fix the deficiencies in the bootstrapping process of the simple chain of trust some support from the boot-ROM of the platform is required.

Emulation of Trusted Execution on an ARM platform

The last section focused on construction of a chain of trust on an ARM platform in the same way as a static root of trust (SRTM) works on a PC platform. In case of the relatively simple and deterministic software configuration of the cloud node client presented here, this is sufficient for practical testing.

Application

Once the Android base system successfully starts, we install *cloud_node.apk* as on the x86 platform and it runs the same as described with the x86 based prototype. The cloud join process is successful.

4.4.4 Summary

In summary, our prototyping work focused on three achitecture challenges:

- Demonstration of a chain of trust for Android. We modify Android x86 to integrate a TXT-based chain of trust.
- Implementation of a chain of trust for an ARM platform. We show how to attach an external TPM to an ARM platform and modify the boot process as early as possible to implement a chain of trust similar to x86 platforms. We do not achieve a root of trust based on a hardware feature like TXT.
- Implementation of the join protocol in Java+jTSS to run on Android. We implement our protocol in Java and use jTSS to communicate to the TPM (device) on each platform, x86 and ARM. The high-level Java code is identical on each client platform and it matters little on which platform it runs.

We did not try advanced features such as persistent storage and the automatic start of a cloud middle-ware software package after our protocol completes. Which specific package is best suited to run a cloud infrastructure on our Android cloud nodes needs to be evaluated and selected separately, in accordance with requirements of a specific scenario. The selection of the cloud software provides the necessary feedback to flesh out the "platform information" data fields in our protocol in more detail, depending on the practical needs of the scenario and hardware platforms.

4.5 Aspects of Platform Security

We discussed the security properties of the cloud join protocol immediately following the protocol's presentation in Section 4.3.1. Naturally, for our prototype platforms the general considerations for the security of (Trusted Computing) platforms also apply—see Section 2.6. Now we discuss additional aspects specific to the scenario of this chapter.

Distributed Nodes

The distribution of cloud nodes to geographically distributed operators provides two potential benefits:

First, distribution of responsibility over additional operators. If all nodes were centralized, a single operator might be able to tamper with all of them. If nodes are widely distributed, the chance that one node operator turns malicious may be considered higher, however, the potential damage is reduced, as he has access to fewer nodes.

Secondly, the increased effort required to gain physical access to all nodes. The designers of Trusted Computing aim to raise the barrier for manipulation, so that physical access to a platform is required for malicious manipulation (see

Section 2.6.1). Consequently the effort to visit all physical sites and obtain physical access to all nodes is higher, the more geographically distributed they are.

Obviously, a distributed data processing cloud with a potential share of compromised nodes is not the best solution for all data processing scenarios. Some sensitive computations do not tolerate a loss of nodes containing sensitive code and data at all.

Node Diversity

The distribution of cloud nodes to a diverse range of parties assists the heterogeneity of the cloud node hardware population. A monoculture of cloud node hardware platforms makes maintenance easy, however, if a serious issue is found, all machines and the whole cloud would be affected. Instead, a diverse set of platform vendors encourages resilience and raises robustness of the cloud.

Future Trusted Platforms

Our ARM prototype effort attaches a dedicated hardware TPM to the platform. However, any additional component increases the cost of a platform. An alternative approach would be to take advantage of ARM TrustZone [7] technology. The idea is to place a software TPM emulation into the secure world domain of TrustZone, which is strictly isolated from the rest of the system running in the normal world. The TPM functions are available to applications via a standard `/dev/tpm0` device and the applications would not notice a difference to a hardware TPM. A prototype of such an approach was demonstrated in [163]. This alternative approach provides the functions of a TPM, but the security implications, advantages and disadvantages need to be studied carefully for each scenario, as the platform boot process is obviously different.

4.6 Summary

The prototype presented in this chapter shows an approach to assemble cloud nodes in a specific, known state into a cloud computing network. We have provided a protocol that uses Trusted Computing features for the platform state attestation of nodes wanting to join the cloud. The implementation prototype targets a lightweight setup and update procedure, which is easily deployable. Overcoming the limits of available hardware, our ARM based prototype of a potential near-future TPM enhanced ARM platform suggests that security qualities similar to x86-based systems are possible, and this supports the vision of future heterogeneous networked cloud nodes. The use of Android as an open-source base platform provides the common software ground between the x86 and ARM platform and maintains the link to the dynamic developments in this area.

5

A PrivacyCA for Anonymity and Trust

5.1 Introduction

The platforms presented in Chapters 3 and 4 take advantage of Trusted Computing technology and protocols. Platform state attestation requires cryptographic keys that a) uniquely identify a platform's hardware TPM and b) to sign platform state reports. Naturally, this poses the problem of how to determine and trust the correct keys are used by the correct entities. What is the certification process for these Trusted Computing keys and what service authority implements this process?

In this chapter we explore considerations and design trade-offs in construction of a prototype *PrivacyCA* (Section 2.5.2) service responder. While the primary focus of the service functions is on AIK certificates, our prototype also provides basic functions for EK certificate management, as not all TPMs come with an EK certificate included by default (Section 2.5.1).

Our implementation prototype strives for the following properties:

- A compact, minimalistic and attestable service.

  This requirement stems from the experience of chapters 3 and 4, whereby if only a small number of components is involved, this produces a short, practical chain of trust[1]. A short chain of trust enables a) attestation, which is a desired property for a certification service. Furthermore, a small number of components encourages b) inspection and discussion of security and robustness of components.

[1] We note that a service with low (runtime library) dependency on other components naturally also needs only a small system to run on.

- A compact, auto-generated network protocol.

  A network protocol open to any client connecting from the Internet represents a constant path of attack. A compact network protocol in combination with an auto-generated input parser reduces the possibility of human implementation (security) issues.

The PrivacyCA prototype implementation described in this chapter (the responder itself and the minimalistic virtual compartment) was publicly released for download and experimentation at [92]. During actual development an experimental public responder was operational for use at [85]. Furthermore, interim results of this effort were reported in several publications—see Appendix A for a detailed listing.

The text of this chapter quotes from these publications, verbatim if appropriate.

5.2 Building a Trustworthy Service

In the Trusted Computing scenario as envisioned by the TCG [129] services like a PrivacyCA play the role of a trusted third party. Ideally, such a service should be demonstrably secure. Of course, trustworthiness does not depend solely on design, but also on practical implementation. This includes trustworthiness of the platform it is executed on.

Regrettably, software engineering techniques so far do not allow formal verification of practical large programs or services. Still, a certification service is a security critical service and should not be implemented with simply successful achievement of service functions in mind. Rather, several qualities should be considered in order to trust an implementation.

Restriction of Code Base

Today's platforms are versatile, but from the viewpoint of assessing platform trustworthiness this is a drawback. Every service with an external interface provides a potential path of attack. It is therefore sensible to deactivate every service and component not strictly necessary for service execution. Indeed, an even stricter approach is to remove every superfluous component. The rationale is that every component can even passively represent a path of attack[2] and certainly makes security analysis and inspection of the overall system more complex.

[2] For example return-oriented programming attacks [102], where reachable code fragments may be chained together by an attacker to act as a new program.

Formal Service Interface

Formal verification of complete platform security is desirable, however, the level of complexity of today's software packages is formidable. Yet, the main path of attack for a service is its network protocol interface, which limits scope. Consequently, a formal notation and automated mechanisms for interface specification and implementation code generation are goals to strive for, and this is a good basis that may later be used in formal verification [72] efforts.

A Safe Programming Language

The programming language in which a service is implemented has an influence on service security properties. A language allowing direct memory access through the use of pointers, or which does not automatically perform range checks is inherently more unsafe than a language that does [36]. While it is possible to create correct programs in a wide variety of languages, it is wise to choose a language that implicitly makes it hard for programmers to generate common security issues[3].

Open-Source Components

Kerckhoff stated [62] that the security of a cryptographic system should solely rely on key secrecy, while the rest of the system should be open for inspection. A similar principle can be applied to components selection. Publicly available source code not only increases the trust in a service, but in addition adds to the development all the other advantages of public review and criticism. For the same reason a trusted service should be deployed on a platform composed of other mature open source components.

Service Attestation

Ensuring trustworthiness is a two-way process [113]. A client should be able to trust a service and vice versa. Trusted Computing techniques help to achieve this goal by measuring service state and reporting it through remote attestation. A trustworthy service should use tools and TPM-backed capabilities available on modern, trusted platforms.

5.3 First Iteration and Building Blocks

This section describes the initial prototyping effort for our PrivacyCA service. The service facilitates creation and validation of EK and AIK certificates (Section 2.5), and also provides a simple, yet sufficient API for certificate retrieval and revocation. See discussion in the second iteration (Section 5.4) for a full command list.

[3]For example, in some languages the compiler automatically generates checks to detect array buffer overflows.

TCG Specific Certificates – TCcert

As mentioned in Section 2.5, the TCG specification of the EK, AIK, and other Trusted Computing certificates requires new data structures in the extensions for X.509 type certificates [50] and attribute certificates [34].

General purpose PKI tools do not support these extensions, and during development of our prototype there was also no publicly available library that did. Consequently, the first step towards a PrivacyCA service is implementation of a Java library (now publicly released under the name TCcert [92]), which enables creation and parsing of these new certificates and field structures. This library is the core building block required for a Trusted Computing-supporting PKI service.

The following credentials are implemented by TCcert for certificate creation and validation:

- TPM Endorsement Key (EK)
- Platform Endorsement (PE)
- Attestation Identity Key (AIK)

A PrivacyCA Service Responder Interface in XML

The interaction of distinct entities connected by a network requires a common protocol understood by all participants. Several protocols exist that are already employed in PKI and for credential management. For Trusted Computing, a protocol should be able to support common PKI services as well as specific Trusted Computing attributes, queries and data structures.

The TCG considered this infrastructure problem in [129]. The two candidates mentioned are the 1) CMC protocol [74] for X.509 certificates and 2) XKMS protocol [75] for XML-based credentials. In a first prototype iteration an approach building on the XKMS option appeared more attractive because of its ability to wrap legacy CA services designed for X.509 certificates and express certificate management in XML (with XML promising readability for human consumption).

Consequently, we developed a first PrivacyCA prototype implementation to build on XKMS as protocol. This first implementation is proof it is possible to use an unmodified XKMS schema to encode and transport the messages required by a Trusted Computing-enabled PKI. For example, Figure 5.1 shows the query[4] for a specific AIK certificate identified by the AIK certificate label name. This operation maps well to XKMS and the binary certificate blob is returned, if found.

However, our implementation experience was also that not all Trusted Computing certificate operations map to XKMS operations in a straightforward way. For example, the AIK certificate creation uses pure binary data blobs in request

[4] "Locate" a piece of data in XKMS terminology.

```
<?xml version="1.0" encoding="UTF-8"?>
<LocateRequest xmlns="http://www.w3.org/2002/03/xkms#" [...]
   Id="_OP2MHIMWWG2FOABW7N769419BACO58T"
   Service="http://opentc.iaik.tugraz.at/xkms/aik">
  <RespondWith>http://www.w3.org/2002/03/xkms#X509Cert</RespondWith>
  <QueryKeyBinding>
    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <KeyName>someLabel</KeyName>
    </KeyInfo>
  </QueryKeyBinding>
</LocateRequest>

<?xml version="1.0" encoding="UTF-8"?>
<LocateResult xmlns="http://www.w3.org/2002/03/xkms#" [...]
  Id="MYF11V848QZ5UMM9BBCKFQROE384D263"
  RequestId="_OP2MHIMWWG2FOABW7N769419BACO58T"
  ResultMajor="http://www.w3.org/2002/03/xkms#Success"
  Service="http://opentc.iaik.tugraz.at/xkms/aik">
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#"> [...] </Signature>
  <UnverifiedKeyBinding>
    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <X509Data>
        <X509Certificate> [...] </X509Certificate>
      </X509Data>
    </KeyInfo>
  </UnverifiedKeyBinding>
</LocateResult>
```

Figure 5.1: Query on a specific AIK certificate mapped to the “Locate” request and response of the XML-based XKMS protocol. The important request and response parameter data fields are marked with purple circles.

and response messages, as they directly relate to low-level TPM/TSS data structures. This property conflicts with the intention of an XML-based protocol to encode as much as possible with plain text XML structures. Furthermore, the proof-of-possession signatures on data expected for certain PKI commands is not always feasible with TC keys. For example, TPM data policy does not allow TPM identity keys to sign arbitrary externally-supplied data, as this would permit fabrication of fake statements[5]. Finally, the use of an XKMS/XML-based solution is in conflict with our intended deployment scenario, where we prefer a compact, minimalistic approach for easier security assessment and attestation. XML introduces an implementation overhead and several external library dependencies that significantly increase service binary size.

Consequently, upon reflection of these issues and moving forward, we abandoned the XML based approach.

[5] See, e.g., TPM v1.2 specification, Part 3, command *TPM_Sign*: “The TPM does not allow TPM_Sign with a TPM_KEY_IDENTITY (AIK) because TPM_Sign can sign arbitrary data and could be used to fake a quote.”[139].

5.4 Second Iteration: A Compact PrivacyCA Protocol and Minimalistic Service

A PrivacyCA that offers a high level of trustworthiness requires a communication protocol that offers a set of PKI operations, at the same time allowing a small implementation. Keeping the guidelines enumerated in Section 5.2 in mind, the goal of the second iteration was to design and build such a PrivacyCA prototype.

This second effort builds on well-maintained off-the-shelf operating system components and mature libraries. The implementation is performed in Java running on Linux. This is a pragmatic approach that offers a good balance of prototyping speed, maturity, features, invested effort and security. However, this approach is versatile and sufficiently trustworthy for common usage.

Following up are implementation choices made for each of the components and how they fit together in detail. Certificates support library TCcert was adopted from the first prototype.

5.4.1 Communication Protocol

A simple ASCII-text-based solution is sufficient for a compact, robust communication protocol. In the following protocol the basic structure of most commands is simply a command identifier followed by data. Data items are line-based, each line terminating with a new line character. Data type identifier and actual data are separated by a colon followed by a space. Binary data payload is transmitted as Base64-encoded strings.

This is a very basic approach and thus not prone to implementation errors. We use the Ragel state machine compiler [122] to auto-generate large parts of the server-side parser code. Ragel generates executable finite state machines from a regular-expression like, formal description of the expected valid input data stream. Furthermore, Ragel allows generation of code for multiple target languages, not only for Java, and thus encourages protocol implementation in other programming languages, which means additional clients for the service.

The short example in Figure 5.2 illustrates the formal specification of a request and response in the `aik_create` command. This specification input fed to Ragel generates a code skeleton of the implementation.

Command Set

The small implemented set of commands is sufficient as a foundation for a Trusted Computing PKI managing AIK certificates. This includes commands for creation and validation of EK certificates, which are currently missing for the majority of shipping TPMs. Also, commands facilitate basic tasks such as creation and revocation of AIK certificates. Figure 5.3 gives an overview list of all commands, as output by our prototype client implementation.

The prototype implementation supports the following operations to enable the AIK certification and validation flow (Section 2.5.2):

```
 create_aik_request =
    "CREATE_AIK_REQUEST" "\n"
    "Blob: " base64 >clsbuf %save_blob "\n"
    ".\n"
    @do_create_aik_request;

create_aik_response =
    "CREATE_AIK_RESPONSE" "\n"
    "Blob1: " base64 >clsbuf %save_blob1 "\n"
    "Blob2: " base64 >clsbuf %save_blob2 "\n"
    ".\n"
    @do_create_aik_response;
```

Figure 5.2: A request/response command specification example, which is the input for parser code generation tool Ragel. The characters ">" "%" and "@" mark custom subroutines to be called at that specific input processing stage.

ekcert_create This command creates a TPM endorsement key certificate for a given public key. Only few vendors supply an EK *and* an EK certificate for their TPMs. In order to support TPMs from all other vendors it is necessary to equip these TPMs with EK certificates.

ekcert_validate This function validates a TPM endorsement certificate. The prototype recognizes certificates issued by Infineon and the `ekcert_create` command (see above).

aik_create implements the AIK certificate creation cycle as specified by the TCG (see Section 2.5.2). By default all AIK certificates issued are saved in local storage by our PrivacyCA.

aik_validate provides a function for any entity to validate an AIK certificate later. The certificate to be checked is submitted by the client and the PrivacyCA returns its assessment whether the certificate presented is valid.

aik_locate offers a search function for retrieval of a specific AIK certificate previously issued. The AIK certificate label serves as search key.

aik_revoke facilitates revocation of individual certificates. The copy in local PrivacyCA storage is removed. Thus, the certificate is no longer available for `aik_locate`.

tcb_quote asks the PrivacyCA to quote itself. The PrivacyCA uses the TPM to perform a quote of platform state and returns it to the requester.

Furthermore, a trusted third-party service like the PrivacyCA presented here should use an end-to-end secure connection for communication with clients. The

```
*** TC apki commandline demonstration client ***

available sub-commands:
  version         ... query library builds
  read_pubek      ... read public EK of TPM
  ekcert_create   ... read public EK and create EK certificate
  ekcert_validate ... validate EK certificate
  local_aik       ... simulate local PCA cycle
  aik_create      ... create AIK certificate
  aik_validate    ... validate AIK certificate
  aik_locate      ... locate AIK certificate
  aik_revoke      ... revoke AIK certificate
  tcb_quote       ... request quote of server state
```

Figure 5.3: Experimental PrivacyCA commandline client commands

prototype implementation uses an unprotected communication channel, namely a simple TCP connection. However, such a connection can be upgraded with, for instance, Transport Layer Security (TLS) [29]. To establish an encrypted TLS-session, one approach may be to derive a channel key from the known public PrivacyCA key, which is already available on the client side[6].

5.4.2 Reduced Software Image

To facilitate functional assessment and security analysis of the service, the code base should be as small as possible. Therefore, it should only include components crucial for its operation. This extends to even removing unnecessary parts of said components. Intuitively, fewer lines of code generally provide less chance for defects to hide. However, the number of defects is not strictly linear to code size (but monotonic). Still, overall reduction of code base complexity aids the goal of understanding and uncovering security issues and furthers the goal of a trustworthy service, and it is thus worthy of exploration.

In the following we enumerate how to remove unused components from our PrivacyCA service prototype implementation.

Software Layers – Overview

The code in the PrivacyCA prototype service, which we implemented in Java, can be roughly grouped into the following layers: PrivacyCA service, JVM/JRE, OS runtime support, OS kernel and the TPM—Figure 5.4 gives an overview of these layers.

The top layer is the Java PrivacyCA implementation itself. In order for the Java bytecode to execute, a Java Runtime Environment (JRE) with a Java Virtual Machine (JVM) is the next lower layer. The JVM makes use of the

[6]For the encryption of the request to the PrivacyCA public key during `aik_create`.

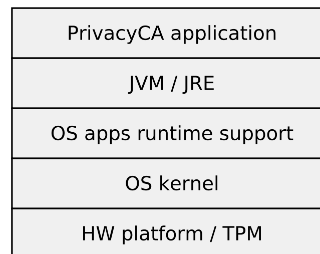


Figure 5.4: The PrivacyCA service implementation consists of several software layers.

native environment to use system specific functions. The native environment is a set of high-level application libraries, the C/C++ standard libraries and operating system functions. The operating system kernel implements low-level services.

The complete layered software stack can run directly on hardware as well as in a virtualization compartment, unaware of the surrounding virtualization layer. In order to achive a compact, minimalistic software image, the initial prototype uses a default configuration of every component. Then, from every layer we remove unused functions, which reduces each layer to the absolute minimum. This process is possible due to the use of only open source components, which can be easily customized and reassembled as required.

Java Environment

To provide the best possible Java compatibility and allow reuse of existing code, we choose IcedTea [149], which is based on Sun's official OpenJDK [80]. The actual subset of the Java environment necessary for the PrivacyCA is small. We use the class-loading profiling feature of Java to identify all classes in the Java runtime environment the PrivacyCA depends on. Additional monitoring of the system dynamic linker/loader `ld.so` produces a list of the required native libraries. For error handling we manually add the required sets of `Exception`s and `Error`s.

Our dynamic identification and removal of unused components reduces the Java runtime for our specific PrivacyCA application to a size in the range of 10 to 20 MB. Naturally, this approach requires manual support, and a sufficient identification of the runtime components required is only possible for small applications where every function must be executed at least once, in order to identify required dependencies.

Cryptographic functionality depends on IAIK JCE [51] and TPM support comes from IAIK jTSS, a pure Java TSS [127].

A Small OS Base and Kernel

The base layer OS needs to be only so powerful as to provide all necessary functions for the stripped down JVM to run as sole application. GNU/Linux is our choice, as it is widely used and actively maintained by a large global community. It is a suitable environment to host the IcedTea JRE. In addition, the build system allows a fine-grained selection of only those capabilities required by the PrivacyCA service. The Linux kernel configuration enables only essential kernel functionalities and a small set of drivers to enable execution directly on hardware or in a virtualized compartment environment.

A Minimal Runtime

The standard Linux `glibc` system library uses about 20 to 25 MB disk space on a typical installation. Additional system and shell tools required for the boot process are about 3 MB.

The compact Busybox [147] toolkit provides a minimal userland program environment for a lean boot process. A minimal set of configuration files for startup and running a GNU/Linux system comes from the `baselayout-lite` package, made available by the Embedded Gentoo project [148]. Furthermore, uClibc [156] replaces glibc as an alternative C library with drastically reduced footprint.

Reductions Result

The PrivacyCA service assembled from components and layers outlined in previous sections can be bundled into a single compartment image. The PrivacyCA operational mode configuration (see Section 2.5.2) is an intrinsic part of the image file. Thus, if the image is measured with trusted computing means at compartment startup, the configuration is also implicitly measured.

Table 5.1 gives an overview of the sizes of reduced components in our prototype implementation snapshot. The complete image has a total size of some 17 MB.

5.5 Use Cases

A small but functional PrivacyCA compartment may be deployed in various scenarios. In this section we outline scenarios where we feel an implementation approach like our prototype can provide added value.

Scalability for PrivacyCAs

When in the future internet platforms integrate a TPM and want to use Trusted Computing, a PrivacyCA potentially has to serve millions of users. Furthermore, natural pressure to create and employ numerous certificates exists as the use of more AIKs per user increases user anonymity to service providers. This

Layer	Component	Size [kB]
OS Kernel	Linux Kernel	900
OS Runtime	BusyBox	750
	Baselayout-lite	61
C Libraries	libstdc++	3545
	uClibc	1103
	GCC Runtime	42
Java Core	Stripped Icedtea JRE	8792
Java Application	PrivacyCA Server Core	200
	lib IAIK JCE	818
	lib IAIK jTSS	312
	lib TCcert	49

Table 5.1: Total component sizes in the PrivacyCA prototype

demand necessitates a powerful, scalable certification authority infrastructure that is capable of issuing and validating a great number of certificates. One way to achieve scalability of a service is through hosting many small PrivacyCA compartments in parallel. This may be done through hardware virtualization technology on a pool of server machines.

We believe our approach is well-suited to be the network facing component as a) it is easy to replicate b) our auto-generated protocol parser (from a formal specification) robustly distinguishes valid requests from invalid ones c) provides attestation of the service itself[7].

Naturally, our current prototype does not scale as the current implementation does not use a real database—certificates are stored simply as files in a directory. Obviously, the next challenge is a certificate storage and synchronisation protocol among individual PrivacyCA instances.

Compact PrivacyCA Service for Restricted or Mobile Environments

The PrivacyCA prototype presented is compact and self-contained: once evaluated it is easy to show through Trusted Computing attestation mechanisms that a specific version started is the evaluated one. Also, self-contained services are easy to deploy, especially for an in-house PrivacyCA service scenario for specific organizations. Local deployments of PrivacyCAs will most likely happen in

[7] We note, if every client demands a fresh TPM quote of service state, this may cause a denial of service, as a TPM can only run a limited number of quote operations per second, and requests from clients may come faster. One solution is a virtual software TPM which somehow references attestation back to the hardware TPM.

advance of Internet-wide services.

The compactness of our service in protocol and implementation encourages its usage on resource restricted systems, such as trusted mobile phones or restricted trusted execution environments.

5.6 Summary

In this chapter we have reported on development challenges and design trade-offs of a PrivacyCA service. Our implementation prototype has achieved the following service properties:

- Implementation of a certification service for Trusted Computing certificates. Our service issues and manages AIK and EK certificates. These are the core certificates in Trusted Computing and used in the remote attestation process of Trusted Computing platforms.
- A robust network interface/protocol. Development of our own custom protocol provides for a) compact implementation and b) auto-generation of protocol parsing code. This reduction in complexity reduces the chance of human implementation mistakes and aids understanding and security analysis.
- A compact execution image. Our software image removes components not strictly needed by the service. Fewer components means less chance of problems but simplifies security inspections.
- An attestable service. Our protocol has one command for remote service attestation. This feature accords clients the possibility of assessing the service before they use it.

6

Privacy-Preserving Payment for Constrained Clients

6.1 Introduction

In the world of new digital devices (see Section 1.1) data processing is no longer limited to one device. A small, lightweight task can be readily accomplished on a smartphone, however, even this is implicitly limited in resources. One solution is to outsource complex tasks to a remote cloud data processing service. In the future it may be possible that a local mobile device user dynamically decides to take advantage of external resources supplied by cloud providers.

Naturally, cloud providers do not and cannot provide resources free of charge. They inevitably demand some form of payment in return for resources consumed by a client. It seems a straightforward implementation would be for every client[1] to open an account with a cloud provider first. Every account must be associated with a payment system identity (e.g., credit card number). Thus, every time a client consumes resources, he must identify his account first, with the result that every resource consumption is billed to that account.

While an accounting/payment system like the simple one just described is sufficient for many cloud services, some clients may deem it too intrusive. As argued in [23], not only code and data need protection, but activity patterns are also worthy of protection. Depending on the service, some clients may actually *demand* that a cloud provider is unable to build detailed dossiers of client activities.

[1] We use "client" in a loose context here, as we focus on the accounting and payment conundrum and not on whether the real-world client is actually a person or a company.

Challenges

In this chapter we focus on privacy in the accounting process for cloud resource consumption. More precisely, our scenario sees a smartphone as a client wanting to consume resources from a specific cloud provider anonymously. We aim to achieve privacy in two properties: a) anonymity: the client can consume resources from a cloud provider without being forced to reveal his identity and b) unlinkability: no separate client action can be linked to his previous or future actions or any action by other clients of the cloud provider. Furthermore, the cloud provider receives payment for the resources consumed by clients, while above privacy properties are intact.

We observe the following challenges in our scenario:

- An accounting scheme that fulfills our two privacy properties. The scheme must run fast enough for practical use on a smartphone.
- Privacy in the payment process itself is not enough. There should be a method of distributing payment credits to clients in a privacy-preserving way, so the cloud provider cannot know which client owns how many credits.
- The safety of credit storage on a client smartphone.

Contribution

In order for our scenario to become feasible, we develop a solution for each challenge:

- We adopt as accounting scheme that from [117], since it fits our scenario and ensures the two privacy properties we seek. However, the reported implementation was done on a PC. Thus, we port the scheme's implementation to smartphone class hardware to find out whether it performs sufficiently well for practical application.
- We develop our scenario into one with three active entities. We identify core operations and how credits flow between them. Also, we show how to preserve privacy in each operation.
- For safety of the client credits store we study use of an isolated, secure processing area and benefits this approach may provide.

Outline

Section 6.2 introduces our scenario and active entities in more detail. Then, Section 6.3 maps individual operations in our scenario to practical accounting scheme details. As a follow-up, Section 6.4 reports on implementation results of porting of the scheme of [117] on a set of hardware platforms. Section 6.5 provides supplementary discussions on certain properties of our approach.

The privacy-preserving cloud resource accounting scenario described in this chapter was developed in theory, however, in practical implementation we focus only on assessing the core question of practical usability of the scheme's execution performance on various smartphone-class platforms. The scenario and proposed solution were presented at an international privacy and security conference, see Appendix A for detailed publication references.

The text of this chapter quotes from these publications, verbatim if appropriate.

6.2 Scenario and High Level Description

In the following we present the scenario for deployment of our privacy-preserving accounting scheme. First, we identify core interacting participants and then we describe the scenario. As a follow-up, we provide a high level description of operations between the entities. Finally, we also discuss privacy perspectives and considerations for all entities.

6.2.1 Entities

Figure 6.1 gives an overview of the three entities in our scenario. We assume a small, resource-limited *client* (C) wishing to outsource computations to a powerful cloud datacentre.

The *cloud provider* (CP) professionally runs a large datacentre. He rents computing power to anyone who can pay for resources consumed. We make no assumption of resources available, but we define them to be accounted for in discrete units of *cloud credits* (CC). Naturally, the client and the cloud provider must consent to a protocol, so that the client can prove to the CP that he has sufficient credits to pay for consumption of a certain amount of resources. Obviously, client and cloud provider must interact directly when the client uses a CP service. However, before this step the client has to obtain a certain amount of cloud credits.

As an additional feature we introduce a third entity, the *credit reseller* (CR). He obtains cloud credit units from the cloud provider in bulk and subsequently distributes and resells them to clients, possibly also through local outlets. The addition of a reseller provides a variety of distribution options. Also, the time between a reseller receiving credits and then distributing them is unknown. Consequently, the cloud provider cannot infer any time-based correlation between issuance of new credits and clients using his service.

6.2.2 Scenario

We assume the client to be a state-of-the-art smartphone with a small isolated execution environment for performing sensitive data processing. In this environ-

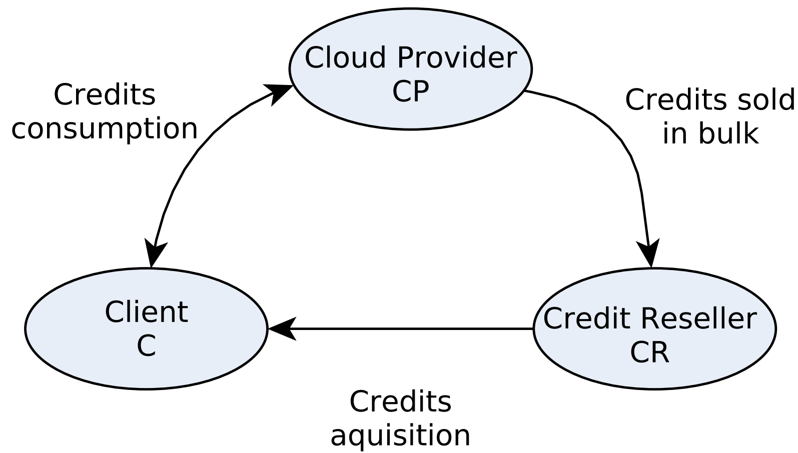


Figure 6.1: Cloud credits accounting flow overview

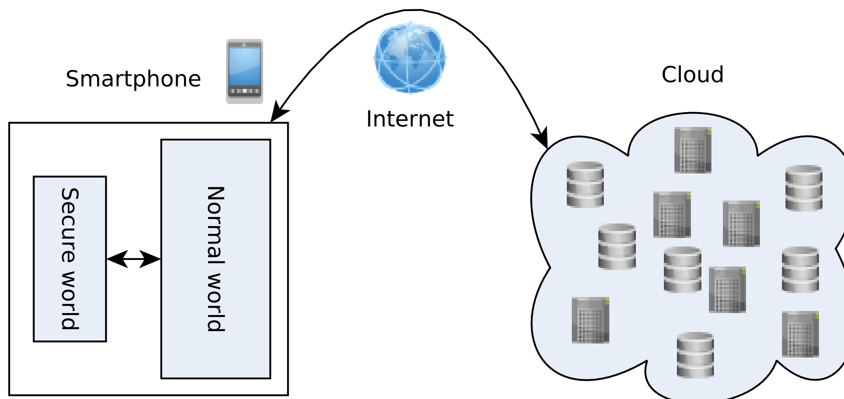


Figure 6.2: Smartphone client connects via the internet to remote cloud provider datacentre.

ment cloud credits are stored and processed in isolation and protected from the main operating system and common applications.

The smartphone is connected to the internet, and thereby to the cloud provider servers. Figure 6.2 gives an overview of this scenario, and subsequently we describe each major block.

Cloud provider

We assume server computing resources offered by the cloud provider are Intel TXT (Section 2.4.1) enabled. The TXT features permit a client to perform a remote attestation protocol on a server/service, in order to obtain an assurance that he will actually receive the service expected.

Network

In our scenario the client is a smartphone on a mobile/wireless network. The network forwards network connections via internet to the cloud provider. There are two potential privacy issues here: a) The wireless network provider knows who the client is and where he is connecting. Furthermore, b) the cloud provider sees the connecting client IP.

In order to mask network connection source and destination, we assume the client uses an anonymization network such as Tor [30]. With network connection anonymization technology the mobile/wireless provider cannot observe where the smartphone is connecting, and the cloud provider does not know which IP the client is connecting from[2].

Smartphone

The client device software environment is split into two processing worlds: a) The normal world hosts the mobile main operating system. Android is in common use today. Also, end user applications run in this environment. Furthermore, b) there is also a small isolated trusted execution environment, the secure world, for sensitive data and operations. Data exchange between the two worlds is restricted, and only a defined set of API calls are available[3].

The secure world stores the cloud credits private data structures and performs accounting operations. Once initial data structures are imported, they never leave the secure world again. One exception is a special process to implement transfer of all credits from one client to another. Applications in the normal world use two API functions: 1) query how many credits are left and 2) request n credits to be used at the specific cloud provider. We discuss the security value of this approach later in Section 6.5.

[2] If one is able to globally monitor the Tor network, of course, then even Tor cannot hide network connections. However, a discussion of the robustness of Tor is beyond our scope, so we leave these discussions and continuous resistance of Tor to attacks on the Tor community.

[3] See for example [163] for a practical ARM TrustZone-based realization of these two worlds architecture.

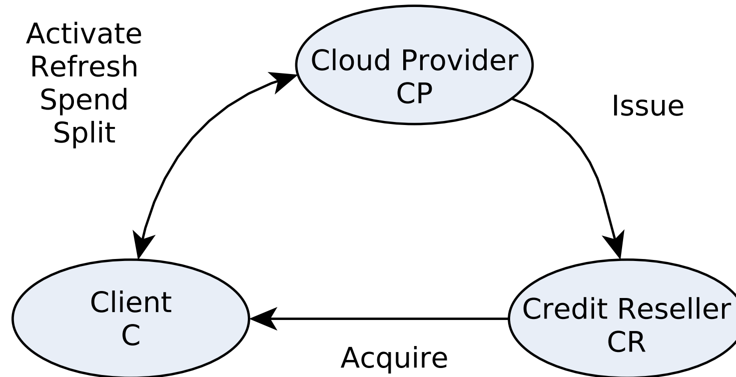


Figure 6.3: High-level operations of credit flows

6.2.3 High Level Description of Operations

We model the flow of cloud credits in our scenario as essentially a circle connecting the three entities. An initial overview of credit flows was given in Figure 6.1, and now we describe the operations in more detail. Figure 6.3 gives a graphic overview of individual high-level operations for the client, cloud provider, and credit reseller.

The cloud provider is the central entity. It provides resources and requires clients to pay for resources consumed with the proper amount of cloud credits. Consequently, clients wishing to consume resources from the cloud service need to acquire a certain number of cloud credits from a credit reseller in advance. This transaction is carried out without involvement of the cloud provider and thus can be offline. After acquisition of credits from the reseller, clients activate them at the cloud provider, perhaps at some later point in time. Clients may not only consume acquired credits themselves, but may also give or sell them to others. Furthermore, if clients overpay credits at the cloud provider for a certain task, e.g., because they do not know how in advance how many resources the task actually requires, then the cloud provider can refund the unused credits.

Subsequently we enumerate these operations in more detail:

Issue: This transaction is an initial bulk transfer of cloud credits (CC) from cloud provider (CP) to credit reseller (CR). It is a business relationship operation that happens in a secure and reliable way (see Section 6.2.4 for privacy considerations).

Acquire: This transaction is carried out between a client (C) and a CR. At the end of this transaction the C holds an amount of CCs that cannot be used before the activation procedure is run.

Activate: This transaction is carried out between the C and the CP. Essentially, the C ends up holding an amount of activated CCs, in form of one token. Furthermore, the token may possess limited validity.

Spend: This transaction is carried out between a C and the CP, and the C pays a certain number of CCs and consumes an equivalent amount of resources from the CP. The client ends up with the result of the task performed at the CP and the remaining number of activated CCs in an updated token.

SplitCredits: This is a transaction between the C and the CP, in which the C ends up with at most two valid tokens. Let us assume that the C wishes to split off n CCs from his store of m CCs (and $m > n$ holds). The one token of m CCs is split into two tokens. The first token contains $m - n$ CCs, and the second token is worth n CCs.

Refresh: This transaction is carried out between the C and the CP. It is required when CCs have limited validity. When the validity period ends, the C must perform this transaction. The current token of the C is exchanged by the CP for a new token which holds the same amount of activated credits, but the token is valid for a new period.

Transfer (not shown in figure)**:** This transaction is carried out between two clients C_1 and C_2. C_1 transfers a number of activated CCs to C_2 and C_1's CCs are deleted, i.e., we explicitly support transferability.

6.2.4 Privacy Considerations

Entities and operations having been outlined, and now we discuss privacy requirements effective in our scenario. We assume two entities can always connect with the technology of a private, secure channel[4]. This does not mean each entity knows the real-life identity of his connection partner, but every participant is sure they are connected via a secure channel to the correct entity. The practical implementation of these secure channels depends on the deployment scenario.

As already mentioned in our introduction (Section 6.1), we aim for privacy through anonymity and unlinkability of transactions conducted. On one side clients enjoy these privacy guarantees, while at the same time the other entities ensure only authorized actions are conducted. We now consider the privacy perspective for each entity in more detail.

Cloud provider

From a privacy perspective the cloud provider is in the position to be honest but also curious. If he claims he does not care about what the clients are doing and is honest, there would be no requirement for the work of this chapter.

If we assume the provider to be curious, he has the means to sniff, log and track all cloud activities and maybe subsequently be able to discern who the clients are and what they do. Consequently, implementation of a privacy-preserving accounting protocol demonstrates a certain level of privacy irrespective of whether the cloud provider is really curious or not. Without a privacy-

[4]For example with Transport Layer Security (TLS) [29] over a Tor [30] connection.

preserving approach customers may refuse to do business with the provider as they cannot be sure what is happening to them.

Credit reseller

The relationship between the cloud provider and the credit reseller is a business one. The cloud provider desires a reliable and clearly identifiable reseller for a good business reputation, and the reseller wishes to ensure he deals with the proper provider contact at the cloud provider. From this follows the requirement for secure communications and transactions between the two, when they trade credits for money. Privacy towards each other is not important in this context.

The reseller is the entity coming into real-life contact with end-users. Consequently, privacy of client identity depends on the method of payment for prepaid cloud credits sold by the reseller and how actual credit handover is carried out (for more discussion on this see also Section 6.5). Without external pressure the reseller has no motivation to invest extra effort in identifying his clients.

Client

The client is the entity motivated to ensure that privacy is enforced at all stages in the transactions. As a first step he has to obtain credits from the reseller. A straightforward solution is to pay in cash at a local reseller. Payment in cash seems privacy-preserving, as it does not leave the trace left by electronic forms of payment[5]. By definition, the client already knows which cloud provider he wants to use[6], thus he must be able to verify the credits offered by the reseller are genuine.

The accounting process for cloud credits for resources consumed represents the core privacy challenge. *Is it possible to consume credits at the cloud provider without him being able to identify and track clients?* A simple accounting process does not provide privacy. Our two privacy goals, anonymity and unlinkability (see Section 6.1), map to our operations in the following way:

Anonymity The cloud provider is unable to identify clients. On assumption there is an anonymous channel between the client and the cloud provider, none of the operations Activate, Spend, SplitCredits and Refresh must involve any information that could be used by the cloud provider to identify a client.

Unlinkability The cloud provider is unable to link individual, separate resource consumptions (=credit spending) or to discover how many credits a client possesses. As in the case of anonymity, the operations Activate, Spend, SplitCredits and Refresh must not provide any information that can be used by the cloud provider to link individual, separate client actions.

[5] Unless we assume a conspiracy of government and central bank to track every paper note by its serial number.

[6] Otherwise he would not be motivated to buy credits in the first place.

6.3 Practical Anonymous Payment

In the following we map our scenario and the operations in it to the scheme of [117]. An introduction to this scheme is found in Section 2.9, so from now on we assume knowledge of it.

Data Structures

We abstract scheme details with the following high-level parameters and state information in order:

- The CP has a set of data $\mathsf{cpparams}_{pub}$ and $\mathsf{cpparams}_{priv}$. The first are the parameters for the CALY signatures used in the scheme and are public knowledge, the second are respective private parameters only known to the CP. Also, the blacklist BL and its content are kept secret by the CP.
- The C also has two states of data: cstate_{pub} is the pair of token $t = (CO(id), CO(s), L)$ and the CP signature σ_t for it. Furthermore, cstate_{priv} contains the token values id, s, L, as well as the randomizers for the commitments and the randomization factors for the CALY signature of the client's token.

The states of the client are updated at every `Consume` operation, and only the current values are required. The data in $\mathsf{cpparams}_{pub}$ is not privacy sensitive, because without concomitant knowledge of cstate_{priv} it is impossible to convince the CP the signature σ_t is valid for token t.

Operations

The `Acquire` operation may be implemented by several means, and we now present one possible implementation.

One privacy-friendly approach is for the CP to issue CCs in the form of scratch cards to the CR. These scratch cards are of different CCs denominations. The user buys such a card in cash at a local CR outlet. The user scratches off the opaque covering, and a barcode is visible[7] containing the following information: a) denomination b) a unique serial number (so the CP can detect duplicates during Activate) c) perhaps validity period information and d) a cloud provider digital signature over all these values (so the user can check the card is genuine).

The end-user now scans the barcode with the built-in smartphone camera[8] and obtains all information necessary to conduct an Activate operation with the CP. Our scratch-card approach is advantageous from a privacy perspective, as the CR does not learn the card serial number, which is only seen by the C. Consequently, if CP and CR were motivated to collude and link the Acquire and the releveant later Activate operation, they would be unable to do so.

[7] For example, a two-dimensional QR-Code [58] would be a suitable solution.

[8] For example, use of a mobile phone camera to provide a trusted import path for cryptographic data was demonstrated in the Seeing-is-Believing effort [71].

Now we present the remaining operations in a more formal manner. For simplicity we assume the CP provides *one* type of resource and has already conducted the `ProviderSetup` procedure, and the public parameters are known to all entities.

Furthermore, we denote by NW the normal world and by SW the secure world of the client platform. The split of the client data structures is motivated by our goal that sensitive data structures are only available through a well-defined API (see Section 6.2.2).

Activate: The NW sends the unique serial number (obtained from scratch card) to the CP, who verifies whether the serial is valid. If yes, C imports $\mathsf{cpparams}_{pub}$ into the SW and C's SW runs an `ObtainCredits` protocol with CP. The client obtains a token and respective signature from the CP and stores cstate_{pub} in the NW and cstate_{priv} in the SW. If the CP does not accept the serial number, the operation terminates.

Spend: The application in the NW requests n CCs and provides cstate_{pub} to the SW. If there are enough CCs available on the token, the SW runs a `Consume` protocol with the CP. During `Consume` the CP is assured that a) cstate_{pub} is a valid token+signature pair b) there are enough CCs available in the token and c) the token id is not already contained in the blacklist BL. If any check fails, the operation terminates. If all checks succeed, the SW updates cstate_{priv} and obtains an updated token+signature pair cstate_{pub}, which is stored in the NW.

SplitCredits: The NW sends m (the amount of CCs to split off the current token) along with cstate_{pub} to the SW. Let us assume $n > m$, where n is the number of remaining CCs in cstate_{priv}. Essentially, the SplitCredits operation works identical to the Spend operation, but the m CCs are not consumed at the CP. Instead, the C receives an additional token worth m CCs (thus creating cstate'_{pub} and cstate'_{priv} at the C) from the CP.

Transfer: There are two clients, C_1 and C_2. The SW of C_1 exports cstate_{priv} and the public and private state information $\mathsf{cstate}_{pub,priv}$ are transferred to C_2 who imports cstate_{priv} into his SW and cstate_{pub} into his NW. The SW of C_1 deletes cstate_{priv}. Note that C_1 transfers all n CCs represented by cstate_{priv} to C_2.

The CP is unaware of this transaction, as he cannot distinguish individual clients during Spend. Consequently, C_1 may Acquire and Activate a token at the CP and later Transfer it to C_2. However, a privacy problem occurs, if C_1 fails to delete the token and later wants to use it: the CP detects the attempted double spending by C_1 and C_2 as he is offered the same token id.

Refresh: The C's NW sends cstate_{pub} to the SW and the SW sends cstate_{pub} along with cstate_{priv} (representing n CCs) to the CP. The CP checks whether cstate_{pub} represents a valid token+signature pair for n CCs. If

yes, the CP performs an `ObtainCredits` protocol with respect to the new parameters $\mathsf{cpparams}'_{pub,priv}$ (see below) with C and issues a new token for n CCs.

Also, we observe the following:

SplitCredits: A client may have several motives for initiating a SplitCredits operation. For instance, he may want to "split" off some CC's from his one token to obtain another token, in order to give one of the tokens to someone else, e.g., as a gift. Another scenario is that a client pays n CCs for some computation, but the computation actually only requires $m < n$ CCs. Then, after having conducted the computation, the CP issues a voucher in form of a new token to C for $n - m$ CCs.

Refresh: The Refresh operation version presented here is the simplest one. Essentially, the client is issued fresh CCs with respect to new CALY signature parameters, every validity period is represented by distinct signature parameters. Unlimited validity of tokens does not scale, as the CP needs to maintain a blacklist BL of previously seen tokens to detect double-spending.

Spend: For every Spend operation at least one CC is removed from circulation, and at least one new entry in the blacklist BL is required to prevent double-spending. Naturally, the number of CCs in circulation must be known to the cloud provider, as he must be able to manage a blacklist of at least this size. The cloud provider blacklisting capacity restricts the maximum number of credits in circulation. Consequently, the CP service policy must enforce a periodical Refresh operation for the client tokens, in effect accounting periods, which results in periodical clearing of the blacklist (for the expired period).

6.4 Implementation

For practical evaluation we prototyped the core anonymous accounting operations of our approach on multiple software and hardware platforms. We ported the original implementation from [117], freely provided by the author, and modified the code to run on our test platforms and obtain performance measurements for certain operations.

6.4.1 Setup

On the software side the scheme was implemented in the high-level language Java and uses the jPBC 1.2.0 library [26], a library for pairing-based cryptography (PBC) in Java. As an alternative to the pure Java implementation there is also a C implementation PBC [13], which can be called from Java via a Java-to-C wrapper. To achieve this wrapper, first we replaced the default JNA wrapper

for the Java-to-C bridge with our own custom JNI coded wrapper to allow the C code to be accessible on Android. Secondly, the ARM code was compiled to take advantage of ARM processor Neon SIMD and ARMv7 instructions features[9].

In the following we denote these two variants as pure Java "-J" and Java with native C core accelerated "-C" variants. Our testing platforms run either Linux (Li) or Android (An) as operating system. We use the following platforms to measure execution speed:

Pc* Laptop HP 8440p Elitebook, Intel i7M620 @2,67 Ghz, running Android for x86 2.3.5 (RC1 20110828) of the Android_x86 porting effort [3], or Ubuntu Linux 11.10 with IcedTea6 1.11pre (OpenJDK 64-Bit Server VM (build 20.0-b11)).

Ek* Freescale i.MX51 evaluation kit [38], Freescale MX515D @800Mhz, running Android 2.3.4 (build R10.3.2_3), or Ubuntu Linux 10.04 LTS with IcedTea6 1.8.10 (OpenJDK Zero VM (build 14.0-b16)).

SpGs Smartphone Google Nexus S, Samsung Exynos 1 GHz (ARM Cortex-A8), running Android 2.3.6.

SpS2 Smartphone Samsung Galaxy S2, Samsung Exynos 1.2 GHz dual-core (ARM Cortex-A9), running Android 2.3.3.

6.4.2 Results

For performance evaluation we focused on the Activate and Spend operations conducted by the client. This is due to the fact that the remaining operations only require negligible computational resources or are based on one of the two afore-mentioned operations, thus they should perform nearly identically.

Activate runs only once for token initialisation, while Spend runs every time credits from the token are spent at the CP. Table 6.1 shows the results for the Activate and Spend operations with our test implementation on the platforms presented in Section 6.4.1. The token limit is encoded as string of bits in a range from 4 to 16 bits, which results in a cloud credits limit of $2^4 = 16$ to $2^{16} = 65536$ credits. We think a limit of 16 bits is sufficient for many practical scenarios. A larger limit is always possible, if the resulting additional computation time is acceptable to the client.

Figure 6.4 provides a more detailed bit-by-bit analysis of the Spend operation execution time for tokens containing 2^x CCs. As can be seen from the figure, the time required running a Spend operation grows linearly[10] in the number of bits x of CCs in the activated token, which is due to the required zero-knowledge range proofs per bit. We do not provide separate timings for computations by

[9] GCC CFLAGS="-march=armv7-a -mfloat-abi=softfp -mfpu=neon"

[10] The non-linearities in the measurements are caused by common operating system services still running in the background. It is impossible to completely stop the operating system and dedicate the CPU solely to our test code.

	PcLi-C	PcAn-C	PcLi-J	SpS2-C	SpGs-C	EkAn-C
Activate	0.06	0.15	0.35	0.64	0.86	1.12
Spend 4 bits	0.16	0.35	0.82	1.43	1.94	2.54
Spend 16 bits	0.34	0.77	1.72	2.99	4.11	5.32

	EkLi-C	PcAn-J	SpS2-J	SpGs-J	EkAn-J	EkLi-J
Activate	1.09	1.80	6.76	11.1	15.7	19.8
Spend 4 bits	2.53	3.60	13.3	20.9	29.6	39.7
Spend 16 bits	5.42	6.87	24.4	41.7	54.7	77.3

Table 6.1: Execution time of Activate and Spend, sorted by length of Activate, for a token limit of 4 bits and 16 bits, in seconds.

the cloud provider, since in the scenario he uses state-of-the-art servers, and the computations are very efficient[11].

The results clearly show three of the *PC** versions are fastest. Also, the C accelerated variants take full advantage of the processor-close nature of C code. We interpret the difference between *PcLi-C* and *PcAn-C* due to the first being 64 bit and the second a 32 bit platform and operating system differences[12]. They are closely followed by *PcLi-J*, a pure Java version executed by a JVM optimized for the PC platform over many years. The next 4 places are claimed by the remaining C builds, as expected by their platform processor powers: 1.2GHz *SpS2-C* before 1.0GHz *SpGs-C* and 800MHz *Ek*-C*. The *EkLi-C* build runs almost identical to *EkAn-C*. The pure Java versions trail, again as expected by their processor speed. For the last platform, EkLi-J, the ARM JVM port appears to be quite un-optimised.

We should note that all Androids except *PcAn-J* and *SpS2-J* exhibit slight process memory leakage during execution. Although the Java garbage collector runs continuously during execution, the memory usage of our process grew steadily. We assume this issue adds to the runtime of the garbage collector as it must consequently scan more and more memory. If memory use remains stable, total execution time should be a little lower. We were unable to determine the cause of this problem.

The main Java classes of our basic test code consume 25 kB in size. The supporting Java libraries (*jpbc-api.jar*, *jpbc-plaf.jar*, *jpbc-pbc-jni.jar*) require 371 kB. The size of native C support libs (*libgmp.so*, *libjpbc-pbc.so*, *libpbc.so*) varies depending on the specific platform. For our platforms they are: Linux_x86_64 772 kB, Android_x86_32 688 kB, Linux_Arm 588 kB, and Android_Arm 800 kB.

[11] The most expensive operation of the CP, i.e. `Consume` for a limit of 2^{30} CCs, reported in [117], takes about 1 second.

[12] Unfortunately, a 64 bit version of Android was not available and also setting up a complete 32 bit Linux test environment was precluded by lack of time.

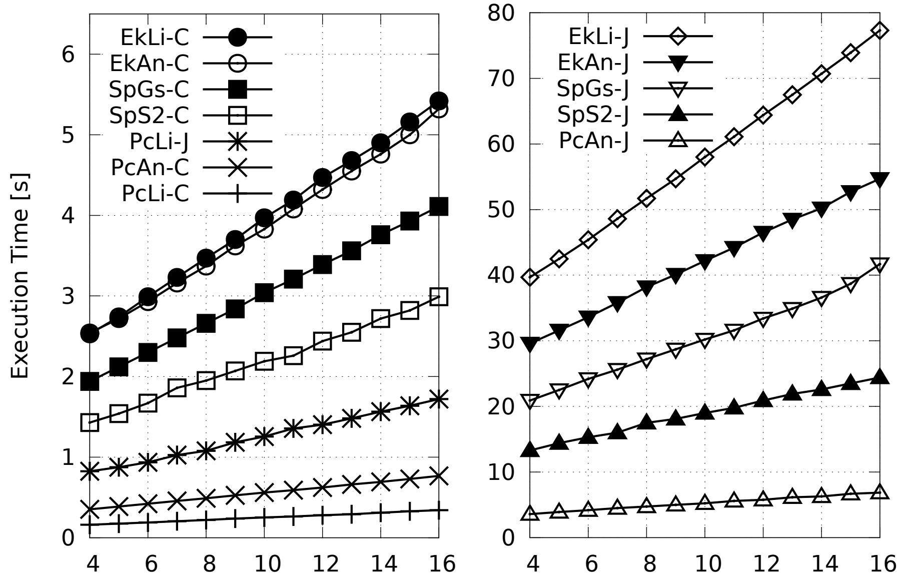


Figure 6.4: Spend execution time; x-axis shows credits token limit $L = 2^x$ [bits]

6.4.3 Implementation Discussion

Measured results of our practical prototype support the feasibility of our scheme. We think a delay of 1, 2 or 3 seconds, and this will become even faster with every new hardware generation, is acceptable, as end-users already expect short delays due to the network communication from the client device to a remote service. Our payment scheme does not incur a large delay, consequently the time for one anonymous Spend operation is acceptable in practice.

While the Java numbers may appear very high at first glance, the variants that use a C core for faster computation are many times faster. An optimised, C-core enhanced Android build runs fast enough to perform Spend operations without long delay, see variant SpS2-C.

Our prototype implementation depends for cryptography on off-the-shelf libraries, which are not optimised at all. Our primary objective was experimental evaluation of whether the performance is acceptable on ARM based smartphones, as it is hard to estimate performance of an ARM platform from PC platform results. As current unoptimised libraries perform well enough, optimised ones can only reduce delay and improve user experience.

Our current test code plus support libraries and JavaVM are way too large to fit into a TrustZone environment[13]. However, we are hopeful that a standalone implementation of the scheme, without any of the generic library overhead and with a small JavaVM, fits into a TrustZone environment.

[13]See Section 2.8, the secure world size is kilobytes in size.

6.5 Discussion

In the following we reflect on supplemental aspects not discussed during presentation of our scheme or our scenario.

Trusted Computing attestation

In Section 6.2.2 we use Trusted Execution Technology to attest cloud servers providing resources or services to the client. Intel TXT technology is already mass-market available and TXT integration has been demonstrated for Linux based servers (see Chapter 3). Thus, if the software image to be rented to the client is agreed upon, a remote client, for example a smartphone, can ask for a remote attestation proof from the server or service to confirm what specific software image has been booted on the server. Attacks on the TXT components require physical intervention (see Section 2.6.1)[14] and consequently raise the bar for manipulation.

Identification at credit purchase

As already noted in Section 6.2.4, the process of how a client obtains cloud credits from a reseller is critical to his privacy. An obvious privacy problem would be a camera at the reseller's outlet. The camera may a) collect a photo of the user's face and b) if the user decides to immediately uncover the barcode of the scratch-card just bought, may also record the barcode. Then, the reseller would be able to link a face to the unique serial number that is used later in the Activate protocol with the cloud provider.

As mitigation a user a) should never uncover the scratch-card at the reseller's outlet or any other public "monitored" place and b) there is still the possibility to Transfer credits further to trustworthy friends.

An isolated secure world

By design, computation in the SW is isolated from the rest of the platform and only reachable via a well-defined, narrow API. This protects the data in the SW from NW influences, so an attacker from the NW is restricted to the available interface[15].

Consequently, an obvious problem is to decide from inside the SW whether a request from the NW is authorized. One solution would be to require a trusted input and display path implementing a user-interaction (e.g., entry of a PIN) for explicit authorization of a sensitive operation in the SW. We assume that users are acting in their own best interests when using their own smartphones, which is reasonable. Nevertheless, this still leaves the problem of potential malware on the client platform that might be able to circumvent even this confirmation feature.

[14] Or good old runtime software bugs that allow privilege escalation.
[15] Unless, of course, the interface or isolation contains a bug.

Isolation security also impacts the privacy of our network connection, as the Tor connection in our scenario is currently anchored in the NW. This leads to the question of how the limited SW can verify whether an anonymous network connection to the CP is properly established.

In summary, our explicit consideration to split the processing into a NW and SW side makes it harder to get to secret data pieces in our scheme, which protects client privacy, compared to the situation whereby the whole scheme just executed in the NW.

Client honesty

If two clients exchange cloud credits by means of a Transfer operation, say C_1 gives n CCs to C_2, then C_2 has no means of verifying whether C_1 has properly deleted the transferred credits. Essentially, if C_1 is dishonest, the first-come-first-served principle applies, and whoever is the first one to spend credits from the token is also able to continue spending, while the other is blocked due to the duplicate detection via the blacklist at the CP. We assume that only clients who trust each other exchange CCs, in which case this is not a problem. Furthermore, from the CP perspective, even if C_1 does not delete his CCs, then this does not mean any harm to the CP, since only n CCs can be consumed in total and clients cannot create "extra" credits.

Forward secrecy

What happens when a smartphone is stolen, lost or seized? If the Spend operation is not protected by additional secret information (see example above, confirmation via PIN entry), then someone in possession of the smartphone could spend all credits left on the currently activated token. Nevertheless, even if an adversary could extract secret information from the secure world, he cannot link the previous actions of the smartphone holder, since no "history data" of the randomization processes of the underlying scheme is stored.

6.6 Summary

In this chapter we have presented a privacy-preserving resource accounting scheme for resource constrained mobile devices such as smartphones. Our solutions for the challenges (Section 6.1) in our scenario are as follows:

- We adopt a suitable accounting scheme that implements the two privacy properties of client anonymity and client actions unlinkability. Our prototype port of the core operations of the scheme for a diverse set of platforms suggests that a version with a C core is fast enough for practical deployment on smartphone platforms.
- We develop the scenario in more detail. We identify three main entities and then enumerate the operations between them. Also, for each entity we

describe privacy motivation and goals. Furthermore, we enumerate operations that affect credits and map them to the operations of the underlying accounting scheme.

- We consider how security of the client's credit store and processing can be realised by use of an isolated, secure world on the client device.

7

Semi-Automated Prototyping of a TPM v2

7.1 Introduction

The platforms and prototypes developed in chapters 3, 4 and 5 take advantage of the functions provided by a Trusted Platform Module (Section 2.2.2) for platform state attestation. The TPM v1.2 was introduced to the market about a decade ago. In order to update the TPM to current hardware platforms, software demands and advances in cryptography, the Trusted Computing Group developed and released a public draft specification for a novel "v2" generation of TPMs. This generation introduces new features and commands, however, also the specification format has changed. The TCG claims[1] that the TPM specification documents are suited for automated parsing and subsequent automated processing.

Contribution

In this chapter we follow up on the automation claim in the TPM v2 specification. We report on our effort to construct a software simulator of TPM v2 by automatically processing public TPM v2 specification documents (see Section 7.3). We discuss how to extract code fragments, command and response parameters, data structures and constants from the public specification documents. Based on these extracts we assemble an all-but complete TPM v2 software simulator.

[1] For more details see Section 7.2.

Only some manual tweaks and supplementary implementation pieces are needed to run this software simulator on standard Linux desktop platforms.

Furthermore, we show how this TPM software simulator can be hosted on an FPGA hardware platform (Section 7.4). With this setup we show the path to a hardware TPM simulation for desktop and embedded systems.

Our prototyping work shows it is possible to explore, develop and experiment with the new v2 TPM generation without further delay.

The TPM v2 specification parsing, analysis and code generation process in this chapter resulted in an open-source full TPM v2 emulator. However, at time of this writing it is not yet publicly released. Description of the specification extraction and development process were presented as a paper at an international conference. Please see Appendix A for detailed references.

This chapter text quotes from previous publications, verbatim when appropriate.

7.2 From TPM Specification to Implementation

For an introduction to TCG's Trusted Computing, the Trusted Platform Module and the TCG Software Stack we refer to Section 2.2. From now on we assume knowledge of this background.

The TPM Specification Process

Specification of a component such as the TPM is a complex, tedious and precarious process due to the complexity of the TPM itself—security functions are by their very nature a challenging domain—and the writing process covering hundreds of pages of specification text requires a certain level of coordination among contributors. The current TPM v1.2 specification documents are more than 700 pages of English text. They describe architectural design decisions [137], implementation recommendations and data structures [138] as well as command details [139]. The text portions were written and edited by multiple contributors and then merged with a text processing application by a central editor. Occasionally official PDF specification files are released to the public by the TCG. As for the TPM v1.2 specification, revisions 62, 85, 94, 103 and 116 are available for download from the TCG homepage.

Humans are the primary consumers of PDFs, reading the specification, understanding and interpreting it and then implementing software for Trusted Computing with a TPM. The result may be TPM firmware, a TPM emulator, or support software, libraries and applications wishing to use the TPM functions and features.

TCG software stacks TrouSerS [155] and jTSS [127] (see also Section 2.2.3) painstakingly interface with every TPM command and data structure. Their

publicly available source code history documents their efforts at turning a specification into a bug-free and compliant TSS implementation that interfaces to a TPM v1.2.

There exists no official TCG provided reference emulator for TPM v1.2. However, a student implemented an open-source TPM emulator as project work [119], and IBM also published an emulator [52]. In 2009 the TCG announced a certification program for TPMs [133]. In this program TPM chip vendors can voluntarily submit their implementation to a TCG-approved specification compliance test-suite. TPM implementations successfully passing this compliance test are then added to a reference list on the TCG homepage [143] as public documentation of their specification conformity.

From this history of human struggle in implementing open-source TPM emulators and open-source TCG Software Stacks for the TPM v1.2 generation we observe two properties:

- The specification should be consistent, accurate and unambiguous for easy human comprehension.
- The specification format should allow the use of automated support tooling to reduce the translation and implementation effort—and thus a natural source of implementation issues.

Also, an ongoing feedback cycle between specification writers and implementers is valuable for early detection and correction of shortcomings and defects. Without a feedback cycle it is more likely that problems pass unnoticed and end up in millions of hardware chips. Compliance tests introduced years after mass-market release cannot undo errors in chips already integrated in platforms.

The TPM v2 Specification

In comparison to version v1.2 the situation has improved significantly for TPM v2. The TPM v2 specification consists of four major parts describing high-level architecture, data-structures, TPM commands and supporting routines. All four parts together comprise almost 1400 pages. To facilitate use of automation tools with the TPM v2 specification, the TCG implemented several important improvements to TPM v1.2 specification. The v2 specification [144] prominently states[2] this intention:

> "Notation: The information in this document is formatted so that it may be converted to standard computer language formats by an automated process. The purpose of this automated process is to minimize the transcription errors that often occur during the conversion process. [...] In addition, the conventions and notations in this clause describe the representation of various data so that it is both human readable and amenable to automated processing. [...]"

[2] See Part 2, Chapter 4.1 Introduction

Furthermore, v2 specification part 4 includes support source code to document the inner workings of a TPM and its surrounding environment. This reveals for v2 a reference software simulation was developed, in parallel to the specification process. Consequently, in an ideal world, the public specification contains enough information and source code to facilitate rapid construction of a TPM v2 software simulator and supporting software libraries with the aid of automation tools.

7.3 Assembly of a TPM v2 Software Simulator

In this section we describe our process of extracting information from the TPM v2 specification and generating a TPM v2 simulator from it. As input we use the publicly-published specification, in the form of the PDF documents (revision 01.03), available on the TCG homepage [144].

Our semi-automated process consists of the following major processing steps, as explained in the following subsections:

- Input data transformation.
- Extraction of source code fragments from specification parts 3 and 4.
- Extraction of data-structures and constants from specification part 2.
- Generation of code from the parsed structures.
- Manual implementation of missing code fragments and integration into a build process.

We use a Linux environment to implement our semi-automated extraction process. It contains the `LibreOffice` suite, `Ruby` language, `Nokogiri` XML parsing library, `indent` source code formatting tool and `makeheaders` tool for C function prototype extraction[3].

7.3.1 Input Data Transformation

As a first step towards the TPM simulator extraction process we transform the PDF specification documents into a format easy to process with automation tools. PDF is a document format which is ready for printing by design. PDF files are not intended to be edited any further and do not retain the text structures (e.g., table delineation) from an original document as written with a word processor.

Our initial idea was to convert the PDF documents to simple plaintext files with a PDF-to-plaintext extractor. Unfortunately, this simple approach loses all formatting meta-information. For example, in order to detect and recover a table from the PDF the location coordinates of where text fragments are on a page are crucial.

[3] LibreOffice 4.2.5.2, Ruby 2.1.1p76 with Nokogiri 1.6.1, indent 2.2.11 and makeheaders 0_p4.

To overcome the limitations of PDF-to-plaintext converters we use the LibreOffice non-interactive PDF import function[4]. This import function converts the PDF specification documents to OpenDocument `FODG` format (for an example see Figure 7.1). The XML-based `FODG` document format preserves layout and formatting information of the original input PDF file, which renders the following extraction steps possible.

7.3.2 Information Extraction

In our process a Ruby script performs further extraction. The script loads the XML-based `FODG` format documents with the Nokogiri XML parser and extracts all text fragments along with their formatting. This process ignores any additional markup, such as the solid horizontal and vertical lines that adorn tables. Extracted text fragments are then stored in an intermediate YAML[5] format file, see example in Figure 7.2.

Identification of Information

In order to find specific fragments in this YAML dump of extracted text, we use regular expression search patterns. With a regular expression and the additional information of how a text fragment is styled[6] it is easy to find locations automatically in the extracted text, where interesting pieces of information are located. Starting from every such position a more detailed analysis proceeds. Detailed analysis of text fragments is performed by another processing step in our script. Depending on the nature of specification part being processed, our script performs different parsing strategies to extract information.

Basic Extraction and Code Generation

Part 2 of the TPM v2 specification defines data structures and constants. To process this part, the script scans for all the tables defining TPM data structures or constants. After all tables are parsed into working memory, it generates a C header file with the appropriate C language `#define` directives for constant, type and structure definitions for all TPM data types.

Part 3 specifies the 112 commands, essentially chip API, of the TPM v2. Specification of a single TPM command always consists of input and output data structures (in form of tables) and a C code fragment that illustrates the actual command actions. Extraction of the C code fragments is straightforward, so our script simply dumps the fragments into one C source file per command. The input and output parameter structures of a command supply the information needed for each command to generate a C header file with the C function signature and input/output data structures for the command.

[4] `libreoffice --headless --convert-to fodg <pdf-filename>`

[5] YAML is a human-readable data serialization format. Its easy readability and availability in Ruby motivated our choice to use it as intermediate format.

[6] For example, a section heading always starts with a number and has a distinctive larger font size than the main text.

```
<draw:frame draw:style-name="gr115" draw:layer="layout" svg:width="16.509cm"
svg:height="0.395cm" svg:x="2.539cm" svg:y="14.306cm">
  <draw:text-box>
    <text:p text:style-name="P1"><text:span text:style-name="T1">
      The information in this document is formatted so that it may be converted
      to standard computer-language
    </text:span></text:p>
  </draw:text-box>
</draw:frame>
<draw:frame draw:style-name="gr127" draw:layer="layout" svg:width="16.506cm"
svg:height="0.395cm" svg:x="2.539cm" svg:y="14.729cm">
  <draw:text-box>
    <text:p text:style-name="P1"><text:span text:style-name="T1">
      formats by an automated process. The purpose of this automated process is
      to minimize the transcription
    </text:span></text:p>
  </draw:text-box>
</draw:frame>
<draw:frame draw:style-name="gr128" draw:layer="layout" svg:width="8.426cm"
svg:height="0.395cm" svg:x="2.539cm" svg:y="15.153cm">
  <draw:text-box>
    <text:p text:style-name="P1"><text:span text:style-name="T1">
      errors that often occur during the conversion process.
    </text:span></text:p>
  </draw:text-box>
</draw:frame>
```


Figure 7.1: The fit-for-automation promise in the TPM v2 specification converted from PDF to the XML-based OpenDocument FODG format.


```
---
- - 2.539
  - 14.306
  - T1
  - 'The information in this document is formatted so that it may be converted to
    standard computer-language '
---
- - 2.539
  - 14.729
  - T1
  - 'formats by an automated process. The purpose of this automated process is to
    minimize the transcription '
---
- - 2.539
  - 15.153
  - T1
  - 'errors that often occur during the conversion process. '
---
```


Figure 7.2: The fit-for-automation promise as extracted text fragments in YAML intermediate format. For every text fragment we keep 4 pieces of information: x-coordinate and y-coordinate of text on page, text style and text string.

Finally, part 4 of the TPM v2 specification contains support algorithms and methods as C code. The code in part 4 is called by the TPM command code in part 3 and also implicitly documents a way to provide a runtime environment for a TPM software simulator. Extraction of the C code fragments in part 4 is straightforward.

In total, our script directly extracts 6792 lines of C code from part 3 and 25277 lines from part 4 of the TPM v2 specification. The extracted source code is incomplete, as most of the header files referenced by C `#include` directives in parts 3 and 4 are missing. We generate these header files with proper function prototypes with our script and/or simply use the *makeheaders* tool as a subroutine to generate them.

7.3.3 Advanced Code Generation and Additional Steps

Sections 6.1, 6.2[7] and 9.11 of part 4 of the TPM v2 specification describe code for command input/output data structure un-/marshalling and individual command dispatching within a TPM v2. However, this code is not included in the specification, as it is repetitive and would significantly increase size of the specification documents. Our script generates this missing code from information parsed in part 2 (basic data structures), part 3 (command input/output data structures) and part 4 (command dispatching). However, at the time of this writing the code is still incomplete—it runs, but still lacks sanity checks on the data. For example, checks for array buffer overflows of pointers or a value being in a certain range as specified are missing.

In summary, with direct source extraction, header files generation and repetitive code generation our script automatically generates about 360 source files (`.c` and `.h`) in 63k lines of code[8]. This result is an all but complete TPM v2 simulator.

Manual Fixes

The target runtime environment for our TPM simulator is a Linux-like system with GNU C compiler and support for the OpenSSL cryptography library. To successfully compile the extracted code on our Linux build environment a few patches and some additional code are required. In the following we enumerate a selection of these manual improvements and modifications:

- A Makefile.
- Modifications for using the Linux platform's OpenSSL library.

[7]Section 6.2 should really be called section 6.3, however, the 01.03 revision of the TPM v2 spec has a formatting problem.

[8]These are estimates, as the number of lines and files vary a little, depending on how much source code pretty printing is applied with the `indent` code formatting tool and whether certain code generation debug options are enabled or not.

- The specification code was developed on a case-insensitive filesystem. However, Linux is case-sensitive, consequently some header references need to be adjusted.
- Some code uses Microsoft C compiler/platform specific functions which have to be replaced, e.g., fopen_s.
- Due to certain defaults on Linux/GCC some standard C headers are missing, while others are superfluous.
- A new "main" where execution starts, with custom startup and initialisation code for a Linux environment.
- In revision 01.03 of the TPM v2 specification the C code from parts 3 and 4 contains stray "<K>" and "<Q>" markup markers—obviously these are not valid C and need to be removed first.
- The network RPC[9] routines in part 4 are specific to Microsoft Windows platforms due to their use of Windows-specific system header files such as `winsock.h` and `windows.h`. We discard the RPC interface originally in the specification in favour of a simpler socket-based interface: Our Linux TPM v2 software simulator exposes one TCP port accepting raw TPM command and response data blobs, without adding any additional communication protocol overhead[10].

In summary, after application of several manual patches to the code, we obtain a Linux executable of a TPM v2 software simulator. Our Linux software simulator utilizes TCP/IP sockets for communication between the simulated Trusted Platform Module v2 and its clients.

The simulator passes the "Hello world!" test of Trusted Computing: A client writes to a PCR, extends a PCR, and reads from a PCR. Unfortunately, at the current time we cannot perform a more comprehensive test, as software for the TPM v2 generation—a TSS—is under development.

7.4 Hardware TPM v2 Simulation Platform

7.4.1 Introduction

Our success in semi-automatic construction of a Linux-based TPM v2 software simulator leads to a question of the feasibility and complexity of building a TPM v2 hardware simulator platform. Ideally, a TPM v2 hardware simulator would seamlessly integrate with a desktop PC or embedded system and operate as a replacement for a hardware TPM. For current desktop PCs and notebooks this implies that a TPM v2 hardware simulator should be able to communicate via the Low-Pin-Count (LPC) [55] bus, the standard type of bus connection for

[9] Remote Procedure Call.
[10] This approach is similar to the one in IBM's TPM v1.2 software emulator [52].

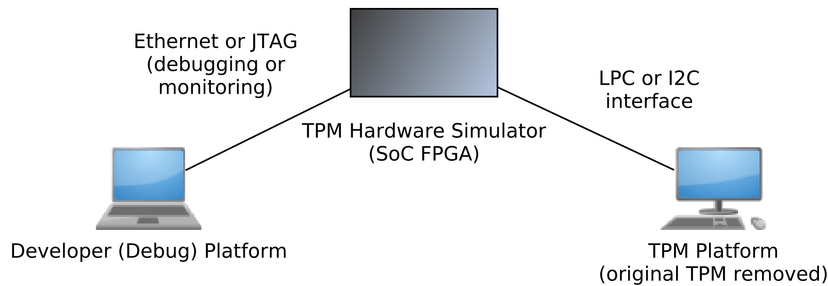


Figure 7.3: Development setup with our TPM hardware simulator (middle). Developer laptop (left) connects to the simulator and monitors the TPM simulation. TPM platform (right) does not know its TPM has been removed and the bus has been connected to the simulator.

a TPM chip on the PC platform. Alternatively, recent variants of version 1.2 TPMs use a Inter-IC (I^2C) bus, which is suitable for use on mobile and embedded devices like smartphones or even embedded microcontrollers. Consequently, a general-purpose TPM v2 hardware simulator should also be able to communicate over embedded bus systems like I^2C.

Environment

In order to achieve flexibility to provide a TPM v2 to both desktop and mobile/embedded platforms, we choose a system-on-chip (SoC) platform implemented on an FPGA evaluation board as our target to port the software simulator from Section 7.3. This board then simulates a TPM to any connecting platform via LPC or I^2C bus. This approach provides for several important properties for development and testing of TPM v2 software:

- The connecting platform “sees” a TPM on the bus, but there is no functional difference between our TPM simulator and an actual hardware TPM.
- The simulator is independent from a platform’s boot process, consequently it can be used to test and develop any kind of TPM-enabled software—this includes trusted boot firmware, trusted bootloaders and operating system kernels without major software changes.
- A software simulation of a hardware TPM simplifies debugging of TPM-enabled software, such as a TSS. At any time the internal state of the TPM simulation can be inspected. A possible development setup is shown in Figure 7.3. Configuration consists of the original TPM platform, the TPM hardware simulator and an optional developer platform that can be used to monitor the internal operation and state of the TPM hardware simulator.

7.4.2 Implementation

Hardware Platform

Base platform for our TPM hardware simulator is a "Xilinx Spartan 3AN Starter Kit" board [168]. This FPGA evaluation board contains a Xilinx FPGA, 64 MiB of DRAM, an ethernet port, two serial (RS232) ports and an on-board JTAG[11] probe with USB interface, which can be used to debug the logic design running on the FPGA.

The core of this SoC design is a synthesizable 32-bit Xilinx MicroBlaze microprocessor. The simulator platform external main system RAM (64 MiB) is connected via a DDR RAM controller. A small 16KiB on-chip RAM is used to store and execute an initial boot loader to load the firmware from non-volatile storage into main system RAM. Also, the FPGA itself integrates a non-volatile 8 Mbit on-chip memory. We use this on-chip memory to hold the FPGA logic design, initial boot loader, TPM simulator firmware, and non-volatile state of the TPM.

The integrated JTAG probe of the FPGA evaluation board can be used to (invasively) debug and single-step the TPM simulator software executing on the Xilinx MicroBlaze processor. Additionally, an UART with external RS232 level-shifters can be used to (non-invasively) trace execution of TPM commands and corresponding TPM responses.

To enable network access to the TPM hardware simulator, the SoC incorporates an Ethernet peripheral block, which interfaces with the Ethernet PHY found on the FPGA evaluation board. Furthermore, adding a I^2C peripheral block can provide a I^2C interface for embedded platform connections and a LPC peripheral block can provide a LPC bus to PC platforms.

Firmware

The firmware of our hardware TPM simulator prototype uses the Xilinx Xilkernel (5.01a) microkernel and its basic operating system services. This kernel can be configured for a small memory footprint and code size, while providing API interfaces compatible with a (subset) of the POSIX threads API and with SystemV-style semaphores. Networking support for the TPM simulator firmware is realized using the Xilinx port of the LwIP (lightweigth IP) embedded TCP/IP stack, configured to expose a BSD style sockets interface. A cross-compiled OpenSSL (1.0.1c) cryptography library completes the runtime environment, with all unused features disabled to conserve space.

The Linux version of the TPM simulator (Section 7.3) ported to the FPGA platform with almost no changes. Naturally, the main start-up code and socket interface required adaptations to suit the runtime environment.

[11] A standard mechanism for debugging embedded systems that may not have any other debug-capable communications channel.

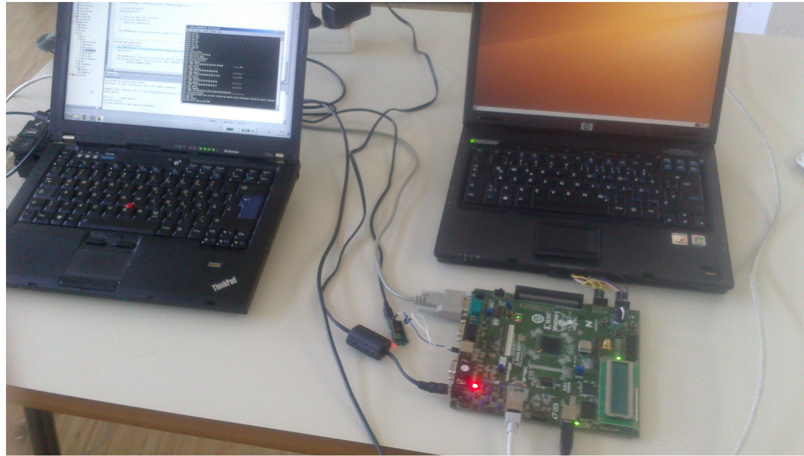


Figure 7.4: Development setup for the TPM hardware simulator. Development laptop (left) connects to the FPGA board (right front). The TPM has been desoldered from the laptop on the right and its LPC bus extended to the FPGA board.

7.4.3 Prototyping Results

The uncompressed size of firmware code for the hardware TPM simulator is currently between 950 KiB and 1 MiB. We anticipate this size can be reduced further. With basic data compression the size of the firmware binary can already be reduced to approximately 600-650 KiB. This allows it to fit the FPGA bitstream, a small bootloader and the compressed TPM simulator firmware into the FPGA on-chip flash. Non-volatile storage is currently held in a small serial EEPROM outside the FPGA, to reduce wear-out of the FPGA's on-chip flash memory.

Figure 7.4 shows the experimental setup we use to validate our TPM hardware simulator prototype. The FPGA board with the TPM hardware simulator is at right bottom of the photo. The laptop on the left side is the development machine used to debug and monitor the simulator. It is connected to the USB port of the on-board JTAG probe and to the serial trace port of the simulator SoC design.

The tiny circuit board in the picture connects the development laptop to the I^2C interface of the FPGA board. The laptop on the right side has been modified, we carefully desoldered its original v1.2 TPM and attached probe wires to the relevant LPC bus pins. These probe wires connect the LPC bus of the HP notebook's motherboard to the FPGA board.

At time of this writing the prototype is fully usable via its network interface. The same set of simple TPM v2 commands as with the pure Linux software simulator (Section 7.3) run. The connection via the I^2C and LPC-bus interface are still work-in-progress.

7.5 Conclusion and Outlook

In this chapter we have put the "fit for automation" claim in the TPM v2 specification to the test. We have successfully assembled a working TPM v2 simulator solely from the official PDF specifications.

The first achievement was the development of an extraction process and script that almost fully automatically extracts a working TPM v2 software simulator from the public specification. Only open source tools are used in our process. We expect the public availability of an open-source simulator to contribute to the development of software for the new TPM v2 generation. Furthermore, the availability of an intermediate data format of the specification allows everyone to generate TPM v2 code and data structures for any programming language.

Based on the results of a working software emulator, we have developed a working proof-of-concept prototype of a hardware TPM v2 simulator. This hardware simulator prototype complements our software simulator, by enabling open development and validation of trusted platform software on actual target hardware (once bus connections are fully implemented). A hardware TPM simulator allows the testing of trusted software and hardware corner case behaviour without the risk of rendering parts of the test platforms permanently unusable or locked.

8 Conclusion and Outlook

To conclude this thesis, we review the challenges we have faced and show how we addressed these challenges and the results we obtained. Finally, we reflect on future challenges that may arise in each problem domain.

8.1 Review of Challenges and Achievements

We have set out in Chapter 1 with the realization that our world is changing. In fact our daily lives are changing due to novel digital devices and services we use every day. Mobile devices now host a computing power and a constantly online network connection only available in desktops computer not long ago. These new devices bring new challenges. How can one trust a certain platform or device to behave as expected? How can one trust a remote service at a cloud provider to be what it claims to be? How is privacy implemented in a mobile device plus remote cloud service scenario? Consequently, in this thesis we have focused on a selection of challenges in this overall setting.

The first challenge was the challenge to attest platform state. More precisely, the problem was discovering how complex it is to measure, report and enforce a certain state of a common PC platform, running a general-purpose operating system. In Chapter 3 we tackled this challenge. There, our prototype platform integrates the use of Intel Trusted Execution Technology (and consequently a Trusted Platform Module) into a Debian Linux platform. A throughout measurement of the components during platform boot is enforced in our prototype. User applications are only able to start, if the platform comes up in a state defined by the platform administrator, otherwise they remain in an encrypted state. The prototype we developed comes with management infrastructure and

tools that simplify administrator installation of the platform and management of platform states and applications

Building on the effort of Chapter 3, we then aimed for reduction in (implementation) complexity in Chapter 4. The question was that if a permanent software installation is not an option, how a single platform could be turned into an attestable processing node to become part of a cloud network. In Chapter 4 we took advantage of the Intel TXT knowledge and tools developed in the previous chapter. However, the new target platform was Android. With Android as basis, we show platform boot into a measured state is easier to implement, compared to the effort required for the previous Linux-based platform. Furthermore, in this chapter we presented a protocol enabling the platform to join a cloud network—but only if the platform is in a state acceptable to the central cloud control entity. As in the future ARM platforms may play a significant role, we prototyped this approach on x86 and ARM platforms. Onto the ARM platform we had to integrate a Trusted Platform Module to the platform ourselves.

As both prototypes in Chapters 3 and 4 build on Trusted Computing and the Trusted Platform Module, there was a need for a service which issues certificates for Trusted Computing keys. More precisely, the service developed in Chapter 5 is a PricacyCA service, to issue certificates for Attestation Identity Keys. These keys are used by a Trusted Platform Module to cryptographically sign platform state reports. In order to verify such a report, naturally the signing key must be attested—with a certificate. The prototype service we implemented in Chapter 5 is robust and minimal. We achieved robustness through formal specification of the service protocol and partly autogeneration of the network parser code. Furthermore, we achieved reduction of the runtime size through deliberate removal of components not absolutely required.

With the challenge of building an attestable platform explored, we then proceeded to the problem of privacy in Chapter 6. If a smartphone client wishes to use (and pay) a remote, attestable cloud service, can this be done in a privacy preserving way? In Chapter 6 we thus developed a scenario where a client pays for resources consumed at a specific cloud provider. The payment is performed in an anonymous and unlinkable way. Consequently, the cloud provider is always paid, however, the privacy of his clients is protected. In our approach we adopted a suitable accounting scheme and ported it to a variety of platforms to evaluate its performance. Our proof-of-concept implementation showed the scheme is sufficiently fast, even on resource-limited mobile platforms. Consequently, such a scenario is basically feasible in practical use—someone just needs to implement the whole infrastructure in all its details.

The Trusted Platform Module is a core component we used in the previous chapters. In Chapter 7 we continued with a study of the recent Trusted Platform Module v2 specification. More precisely, we studied the specification promise that it is written in a way suitable for automation tools. As TPMv2 chips are still rare, our goal was to construct a TPM v2 simulator solely from the specification. We have achieved this. First, we showed how to extract informa-

tion from the specification in an automated process. Secondly, we constructed a TPM v2 software simulator on the Linux platform. Thirdly, we ported this software simulator to an FPGA platform, which in the near future is destined to completely simulate a hardware TPM on common bus connections.

8.2 Future Work

The challenges of computer security are constantly evolving. Still, starting from the work we have accomplished in this thesis, we can identify nearby challenges and even speculate what additional challenges the future may bring.

First, our work shows the complexity of integrating Trusted Computing technology into a common operating system. Our approach was measurement of large chunks of code and data, and not individual, small components. It follows from our experience that measurements complexity is still an unsolved problem. Once there are too many components (=measurements), certain features like, for example, sealing data to a certain platform state, may no longer be feasible. However, while the complexity on a general-purpose platform may be too much, Trusted Computing may work for a more restricted runtime environment in the cloud. For example, the OpenAttestation project [157] currently develops Trusted Computing infrastructure components for OpenStack-based clouds—it will be interesting to see what emerges from this combination.

Secondly, as the common desktop PC is replaced by new devices such as smartphones and tablets, computer security problems also show up on these platforms. One approach to isolating sensitive data processing from a (large) operating system is Trusted Execution Environments (Section 2.8). In our work on privacy-preserving payment (Chapter 6) we consider this technology. However, while there are standards for Trusted Execution Environments (TEEs) from Global Platform [43, 44], this technology is still not widely spread. The level of market adoption and future applications for TEEs are open questions.

Thirdly, privacy-preserving payment remains an active area of research. We have shown in this thesis that in a restricted scenario a payment algorithm can provide anonymity and unlinkability at a level of computation complexity that allows the scheme to run fast enough on mobile platforms. However, a solution for a much wider scenario, an Internet-wide scenario with many participants and many cloud services is needed. The popularity of the Bitcoin [76] digital currency protocol is a sign of demand, however it only provides pseudonymity[1], but not anonymity and unlinkability like with our scheme. There is research done in this domain, for example the recent Zerocoin [73] proposes one way to integrate anonymity into Bitcoin. The market for privacy-preserving digital payment is still seeking a universal, customer-friendly solution.

Fourthly, the first TPM v2 chips are now available. Our TPM v2 simulator (Chapter 7) allows everyone to experience this new generation, however soft-

[1] A pseudonym identifies a holder, that is, one or more entities who possess something but do not disclose their true identity. Bitcoin's transaction log is completely public, and user identity is his pseudonym.

ware support is not there yet. Design and implementation of an application side library (TSS) to communicate with the TPM is still an open problem for TPM v2. Also, for the TPM v2 to be adopted beyond mainly the PC platform, support software should also be available to mobile and embedded platforms. Furthermore, while in our work we have concentrated on EK and AIK certificates (Chapter 5), the certificates, their content and a service infrastructure similar to a PrivacyCA must be updated for the TPM v2 generation.

Publications and Cooperations

Contributions to this thesis were previously issued in a series of publications at international conferences, workshops, summer/winter schools and as interim results of ongoing research projects. Naturally, the written content of this thesis resembles the content of these publications—indeed some text passages are quoted verbatim. At the beginning of each chapter prominent grey boxes provide indication of this fact and reference to this chapter.

Assembly of the distinct original publications into one thesis provided an opportunity to update, improve and augment the original texts. Consequently, this thesis goes into more detail for better understanding. Also, it shows interconnectedness of individual challenges tackled in each chapter hereof.

In the following we enumerate original publications and collaborations with other researchers and describe how the content of each publication maps (roughly) to the content of this thesis.

Chapter 2

The background chapter draws on material from all publications (see below). It provides an introduction to preliminaries in the respective problem domains. A reader of this thesis may refer to the background in this chapter for better understanding, or for additional references to resources on certain topics.

Chapter 3

The "acTvSM" platform—"A Dynamic Virtualization Platform for Enforcement of Application Integrity"—was presented in a series of publications:

- M. Pirker and R. Toegl.
  Towards a virtual trusted platform.
  Journal of Universal Computer Science, 16(4), 2010. [88]
- M. Pirker, R. Toegl, and M. Gissing.
  Dynamic enforcement of platform integrity (a short paper).
  In A. Acquisti, S. W. Smith, and A.-R. Sadeghi, editors, *Trust and Trustworthy Computing (Proceedings of the Third International Conference TRUST)*, volume 6101 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2010. [89]
- R. Toegl, M. Pirker, and M. Gissing.
  acTvSM: A dynamic virtualization platform for enforcement of application integrity.
  In L. Chen and M. Yung, editors, *Trusted Systems (Proceedings of the Second International Conference (INTRUST), Revised Selected Papers)*, volume 6802 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2010. [125]
- M. Gissing, R. Toegl, and M. Pirker.
  Management of integrity-enforced virtual applications.
  In C. Lee, J.-M. Seigneur, J. J. Park, and R. R. Wagner, editors, *Secure and Trust Computing, Data Management, and Applications*, volume 187 of *Communications in Computer and Information Science*. Springer Verlag, 2011. [42]

The work presented in Section 3 is the core Linux platform that integrates a measuring boot process anchored in Intel's Trusted Execution Technology. The primary virtual application prototype payload for the platform is the "TvSM", co-designed and implemented by Ronald Toegl and Florian Reimair as documented in Ronald Toegl's PhD thesis [123]. Andreas Niederl contributed to the low-level scripting, testing and integration of the acTvSM platform while Michael Gissing contributed to the TVAM script.

Chapter 4

The "Lightweight Distributed Heterogeneous Attested Android Clouds" effort was presented in:

- M. Pirker, J. Winter, and R. Toegl.
  Lightweight distributed attestation for the cloud.
  In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER)*, 2012. [94]
- M. Pirker, J. Winter, and R. Toegl.
  Lightweight distributed heterogeneous attested android clouds.
  In *Trust and Trustworthy Computing (Proceedings of the 5th International Conference TRUST)*, volume 7344 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2012. [95]

Ronald Toegl contributed his background on cloud security while Johannes Winter contributed his extensive expertise in the low-level TPM bus interface and ARM/FPGA development. Johannes continued his explorations of the TPM bus interface in [162], which finally concluded in his master's thesis [161].

Chapter 5

The challenges of "A PrivacyCA for Anonymity and Trust" were originally presented as:

- M. Pirker, R. Toegl, D. Hein, and P. Danner.
  Advances on PrivacyCAs.
  In *Proceedings of the Workshop of Challenges for Trusted Computing*, European Trusted Infrastructure Summer School (ETISS) 2008. [90]

- M. Pirker, R. Toegl, D. Hein, and P. Danner.
  A PrivacyCA for anonymity and trust.
  In L. Chen, C. J. Mitchell, and M. Andrew, editors, *Trusted Computing (Proceedings of the Second International Conference TRUST)*, volume 5471 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2009. [91]

Ronald Toegl contributed his expertise with jTSS. Daniel Hein and Peter Danner provided helpful feedback and suggestions on the service.

Chapter 6

The work of privacy-preserving payment "Practical Privacy-Preserving Cloud Resource-Payment for Constrained Clients" was presented at:

- M. Pirker, D. Slamanig, and J. Winter.
  Practical privacy-preserving cloud resource-payment for constrained clients.
  In *Privacy Enhancing Technologies (Proceedings of the 12th International Symposium (PETS))*, volume 7384 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2012. [87]

Daniel Slamanig is the original inventor of the underlying privacy-preserving scheme [117], he contributed his original code for adaptation and was a collaborator on further development of the scenario. Johannes Winter contributed his expertise in porting and low-level debugging on ARM platforms.

Chapter 7

The work of "Semi-Automated Prototyping of a TPM v2 Software and Hardware Simulation Platform" was first presented at:

- M. Pirker and J. Winter.
  Semi-automated prototyping of a TPM v2 software and hardware simulation platform.
  In *Trust and Trustworthy Computing (Proceedings ot the 6th International Conference TRUST)*, volume 7904 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2013. [93]

Johannes Winter contributed his expertise and prefab pieces on the FPGA platform and is currently adding the missing TPM bus interfaces. At time of this writing the resulting full simulator package is not yet publicy released.

Related Publications

Some topics of this thesis have also been explored in the following collaborations:

- M. Pirker and D. Slamanig.
  A framework for privacy-preserving mobile payment on security enhanced ARM TrustZone platforms.
  In *Proceedings of the 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE Computer Society, 2012. [86]

This paper also discusses privacy-preserving payment, however in contrast to [87] of Chapter 6 the scenario is anonymous payment for public transportation tickets.

- S. Kraxberger, R. Tögl, M. Pirker, E. P. Guijarro, and G. G. Millan.
  Trusted identity management for overlay networks.
  In R. Deng and T. Feng, editors, *Information security practice and experience*, volume 7863 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2013. [65]

In this paper the EK+AIK cycle of Trusted Computing is also taken advantage of like we did for our protocol in Chapter 4. However, here the focus is on robust identities for overlay networks.

- J. Winter, P. Wiegele, M. Pirker, and R. Toegl.
  A flexible software development and emulation framework for ARM TrustZone.
  In *Proceedings of the Third international conference on Trusted Systems (INTRUST'11)*, volume 7222 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2012. [163]

This paper describes the initial porting effort of jTSS to Android and the work to make it run with a software emulated TPM on an ARM platform.

- R. Toegl, F. Reimair, and M. Pirker.
  Waltzing the Bear, or: A trusted virtual security module.
  In S. Capitani di Vimercati and C. Mitchell, editors, *Public Key Infrastructures, Services and Applications, 9th European Workshop, EuroPKI 2012, Pisa, Italy, September 2012, Revised Selected Papers*, volume 7868 of *Lecture Notes in Computer Science*. Springer Verlag, 2013. [126]

This paper describes the “TvSM”, which was the original virtual application demonstration payload for the acTvSM platform developed in Chapter 3.

- R. Toegl and M. Pirker.
  An ongoing game of Tetris: Integrating trusted computing in Java, block-by-block.
  In D. Grawrock, H. Reimer, A.-R. Sadeghi, and C. Vishik, editors, *Future of Trust in Computing*. Vieweg+Teubner, 2009. [124]

This paper describes the development of Java support for Trusted Computing. All the Trusted Computing Java libraries and tools used throughout this thesis depend on this base layer.

Bibliography

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, San Jose, California, USA, 2006. ACM.

[2] Advanced Micro Devices. *AMD64 Virtualization: Secure Virtual Machine Architecture Reference Manual*, May 2005.

[3] Android x86 Team. Android-x86 – porting android to x86. `http://www.android-x86.org/`, 2011.

[4] E. Androulaki, M. Raykova, S. Srivatsan, A. Stavrou, and S. M. Bellovin. PAR: Payment for anonymous routing. In N. Borisov and I. Goldberg, editors, *Privacy Enhancing Technologies, 8th International Symposium (PETS 2008)*, volume 5134 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2008.

[5] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP'97. IEEE Computer Society, 1997.

[6] ARM Ltd. ARM security technology – building a secure system using TrustZone technology. `http://infocenter.arm.com/`, 2009. Whitepaper.

[7] ARM Ltd. TrustZone technology overview. Introduction available at: `http://www.arm.com/products/esd/trustzone_home.html`, 2011.

[8] M. H. Au, W. Susilo, and Y. Mu. Practical anonymous divisible e-cash from bounded accumulators. In G. Tsudik, editor, *Financial Cryptography and Data Security, 12th International Conference (FC 2008)*, volume 5143 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2008.

[9] A. M. Azab, P. Ning, and X. Zhang. SICE: A hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS'11. ACM, 2011.

[10] B. Balacheff, L. Chen, D. Plaquin, and G. Proudler. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall, 2002. ISBN 0-13-009220-7.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03. ACM, 2003.

[12] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41. USENIX Association, 2005.

[13] Ben Lynn. PBC: Pairing-based cryptography library. `https://crypto.stanford.edu/`.

[14] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 305–320, 2006.

[15] S. Berger, R. Cáceres, D. Pendarakis, R. Sailer, E. Valdez, R. Perez, W. Schildhauer, and D. Srinivasan. TVDc: Managing security in the trusted virtual datacenter. *ACM SIGOPS Operating Systems Review*, 42, 2008.

[16] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith. TOCTOU, traps, and trusted computing. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, volume 4968 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2008.

[17] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM, 2004.

[18] A. Brown and J. S. Chase. Trusted platform-as-a-service: A foundation for trustworthy cloud-hosted applications. In *Proceedings of the 3rd ACM workshop on Cloud Computing Security Workshop*, CCSW '11. ACM, 2011.

[19] R. Cáceres, C. Carter, C. Narayanaswami, and M. Raghunath. Reincarnating PCs with portable soulpads. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05. ACM, 2005.

[20] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. In *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2005.

[21] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In M. Franklin, editor, *Advances in Cryptology*

– *CRYPTO 2004, 24th Annual International Cryptology Conference*, volume 3152 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2004.

[22] D. Chaum. Blind signatures for untraceable payments. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *Advances in Cryptology: Proceedings of Crypto 82*. Springer, 1983.

[23] Y. Chen, V. Paxson, and R. H. Katz. What's new about cloud computing security? Technical Report UCB/EECS-2010-5, University of California, Berkeley, 2010.

[24] Y. Chen, R. Sion, and B. Carbunar. XPay: Practical anonymous payments for Tor routing and other networked services. In *WPES '09: Proceedings of the 8th ACM workshop on Privacy in the Electronic Society (WPES)*. ACM, 2009.

[25] Computer Security Center. *Trusted Computer System Evaluation Criteria*. Department of Defense, 1983. CSC-STD-00l-83.

[26] A. De Caro and V. Iovino. jPBC: Java pairing based cryptography. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*. IEEE, June 2011.

[27] Debian Developers. Debien Lenny release. `https://www.debian.org/releases/lenny/`, 2012.

[28] W. Denk et al. Das U-Boot – the universal boot loader. `http://www.denx.de/wiki/U-Boot`, 2010.

[29] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, IETF, 2008.

[30] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[31] L. Duflot and Y.-A. Perez. Can you still trust your network card? CanSecWest 2010, 2010.

[32] L. Duflot, Y.-A. Perez, and B. Morin. What if you can't trust your network card? In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11. Springer-Verlag, 2011.

[33] P. England. Practical techniques for operating system attestation. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Trusted Computing – Challenges and Applications (Proceedings of the 1th International Conference on Trusted Computing and Trust in Information Technologies TRUST)*, volume 4968 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2008.

[34] S. Farrell and R. Housley. An internet attribute certificate profile for authorization. `http://www.ietf.org/rfc/rfc3281.txt`, 2002.

[35] Federal Government of Germany. Federal government's comments on the tcg and ngscb in the field of trusted computing. Technical report, 2004.

[36] M. Felleisen and R. Cartwright. Safety as a metric. In *Proceedings of the 12th Conference on Software Engineering Education and Training*, CSEET '99. IEEE Computer Society, 1999.

[37] M. Franz, P. Williams, B. Carbunar, S. Katzenbeisser, A. Peter, R. Sion, and M. Sotáková. Oblivious outsourced storage with delegation. In G. Danezis, editor, *Financial Cryptography and Data Security, 15th International Conference (FC 2011)*, volume 7035 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2011.

[38] Freescale Semiconductor Inc. i.MX51 evaluation kit. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MCIMX51EVKJ`, 2010.

[39] C. Fruhwirth. New methods in hard disk encryption. Technical report, Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology, 2005.

[40] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03. ACM, 2003.

[41] C. Gebhardt and A. Tomlinson. Secure virtual disk images for grid computing. In *Proceedings of the 2008 Third Asia-Pacific Trusted Infrastructure Technologies Conference*, APTC '08. IEEE Computer Society, October 2008.

[42] M. Gissing, R. Toegl, and M. Pirker. Management of integrity-enforced virtual applications. In C. Lee, J.-M. Seigneur, J. J. Park, and R. R. Wagner, editors, *Secure and Trust Computing, Data Management, and Applications*, volume 187 of *Communications in Computer and Information Science*. Springer Verlag, 2011.

[43] GlobalPlatform. TEE client API specification v1.0. `http://www.globalplatform.org/specificationsdevice.asp`, July 2011.

[44] GlobalPlatform. TEE internal API specification v1.0. `http://www.globalplatform.org/specificationsdevice.asp`, December 2011.

[45] GNU Project. GNU Grub. `https://www.gnu.org/software/grub/`.

[46] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009. ISBN 978-1934053171.

[47] Grml Developers Team. GRML Live Linux. `http://grml.org/`.

[48] J. Großschädl, T. Vejda, and D. Page. Reassessing the TCG specifications for trusted computing in mobile and embedded systems. In M. Tehranipoor and J. Plusquellic, editors, *Proceedings of the 1st IEEE Workshop on Hardware-Oriented Security and Trust*, HOST 2008. IEEE Computer Society, 2008.

[49] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation – virtual machine directed approach to trusted computing. In *Virtual Machine Research and Technology Symposium*, 2004.

[50] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 public key infrastructure certificate and certificate and CRL profile. http://www.ietf.org/rfc/rfc3280.txt, 2002.

[51] IAIK. The IAIK provider for the Java cryptography extension (IAIK-JCE). http://jce.iaik.tugraz.at/sic/Products/Core-Crypto-Toolkits/JCA_JCE.

[52] IBM. IBM's software trusted platform module (TPM). http://ibmswtpm.sourceforge.net/.

[53] Intel Corporation. Intel active management technology (AMT). http://www.intel.com/technology/platform-technology/intel-amt/index.htm.

[54] Intel Corporation. Trusted boot. http://sourceforge.net/projects/tboot/.

[55] Intel Corporation. Intel Low Pin Count (LPC) interface specification. http://www.intel.com/design/chipsets/industry/25128901.pdf, August 2002. Revision 1.1.

[56] Intel Corporation. Intel Trusted Execution Technology software development guide. http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html, 2009.

[57] Intel Corporation. Production SINIT ACM downloads. https://software.intel.com/en-us/articles/intel-trusted-execution-technology, 2014.

[58] ISO. Information technology – automatic identification and data capture techniques – QR Code 2005 bar code symbology specification. ISO ISO/IEC 18004:2006, International Organization for Standardization, 2006.

[59] Junjiro R. Okajima. Aufs – advanced multi-layered unification filesystem. http://aufs.sourceforge.net/.

[60] M. Kapeundl. An integrated workflow for design and implementation of security modules., 2013. Master's thesis.

[61] B. Kauer. OSLO: Improving the security of trusted computing. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. USENIX Association, 2007.

[62] A. Kerckhoffs. La cryptographie militaire. In *Journal des sciences militaires*, volume IX, 1883.

[63] A. Kivity, V. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *OLS2007: Proceedings of the Linux Symposium*, pages 225–230, 2007.

[64] F. J. Krautheim, D. S. Phatak, and A. T. Sherman. Introducing the trusted virtual environment module: A new mechanism for rooting trust in cloud computing. In *Trust and Trustworthy Computing (Proceedings of the Third International Conference TRUST)*, volume 6101 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2010.

[65] S. Kraxberger, R. Tögl, M. Pirker, E. P. Guijarro, and G. G. Millan. Trusted identity management for overlay networks. In R. Deng and T. Feng, editors, *Information security practice and experience*, volume 7863 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2013.

[66] U. Kühn, M. Selhorst, and C. Stüble. Realizing property-based attestation and sealing with commonly available hard- and software. In *Proceedings of the 2007 ACM workshop on Scalable Trusted Computing (STC)*, STC'07. ACM, 2007.

[67] J. Lyle and A. Martin. On the feasibility of remote attestation for web services. In *Proceedings of the 2009 International Conference on Computational Science and Engineering – Volume 03*. IEEE Computer Society, 2009.

[68] J. Marchesini, S. Smith, O. Wild, and R. MacDonald. Experimenting with TCPA/TCG hardware, or: How i learned to stop worrying and love the bear. Technical Report TR-2003-476, Department of Computer Science/Dartmouth PKI Lab, Dartmouth College, 2003.

[69] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010.

[70] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2008.

[71] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-Is-Believing: Using camera phones for human-verifiable authentication. In *IEEE Symposium on Security and Privacy*, 2005.

[72] C. Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[73] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13. IEEE Computer Society, 2013.

[74] M. Myers, X. Liu, J. Schaad, and J. Weinstein. Certificate management messages over CMS. `http://www.ietf.org/rfc/rfc2797.txt`, 2000.

[75] S. H. Mysore and P. Hallam-Baker. XML key management specification (XKMS 2.0). W3C recommendation, W3C, 2005. `http://www.w3.org/TR/2005/REC-xkms2-20050628/`.

[76] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2009.

[77] National Institute of Standards and Technology (NIST). Secure hash standard (SHA), 1995. FIPS PUB 180-1.

[78] Open_TC Consortium. The open trusted computing project (Open_TC). `http://opentc.net/`, 2005-2009.

[79] Oracle Corporation. The K virtual machine. `http://www.oracle.com/technetwork/java/ds-137153.html`.

[80] Oracle Corporation. OpenJDK. `http://openjdk.java.net/`.

[81] Oracle Corporation. Oracle press release, Oracle announces Java 8. `http://www.oracle.com/us/corporate/press/2172618`.

[82] Oracle Corporation. Project Jigsaw. `http://openjdk.java.net/projects/jigsaw/`.

[83] B. Parno, J. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. SpringerBriefs in Computer Science. Springer, 2011. ISBN 978-1-4614-1459-9.

[84] B. Pfitzmann, J. Riordan, C. Stueble, M. Waidner, and A. Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Zurich, 2001.

[85] M. Pirker. Experimental PrivacyCA responder web service. `http://privacyca.iaik.tugraz.at/`, 2009.

[86] M. Pirker and D. Slamanig. A framework for privacy-preserving mobile payment on security enhanced ARM TrustZone platforms. In *Proceedings of the 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE Computer Society, 2012.

[87] M. Pirker, D. Slamanig, and J. Winter. Practical privacy-preserving cloud resource-payment for constrained clients. In *Privacy Enhancing Technologies (Proceedings of the 12th International Symposium (PETS))*, volume 7384 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2012.

[88] M. Pirker and R. Toegl. Towards a virtual trusted platform. *Journal of Universal Computer Science*, 16(4), 2010.

[89] M. Pirker, R. Toegl, and M. Gissing. Dynamic enforcement of platform integrity (a short paper). In A. Acquisti, S. W. Smith, and A.-R. Sadeghi, editors, *Trust and Trustworthy Computing (Proceedings of the Third International Conference TRUST)*, volume 6101 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2010.

[90] M. Pirker, R. Toegl, D. Hein, and P. Danner. Advances on PrivacyCAs. In *Proceedings of the Workshop of Challenges for Trusted Computing*, ETISS, 2008.

[91] M. Pirker, R. Toegl, D. Hein, and P. Danner. A PrivacyCA for anonymity and trust. In L. Chen, C. J. Mitchell, and M. Andrew, editors, *Trusted Computing (Proceedings of the Second International Conference TRUST)*, volume 5471 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2009.

[92] M. Pirker, R. Toegl, T. Winkler, and Others. Trusted computing for the Java platform. `http://trustedjava.sourceforge.net/`, 2009.

[93] M. Pirker and J. Winter. Semi-automated prototyping of a TPM v2 software and hardware simulation platform. In *Trust and Trustworthy Computing (Proceedings ot the 6th International Conference TRUST)*, volume 7904 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2013.

[94] M. Pirker, J. Winter, and R. Toegl. Lightweight distributed attestation for the cloud. In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER)*, 2012.

[95] M. Pirker, J. Winter, and R. Toegl. Lightweight distributed heterogeneous attested android clouds. In *Trust and Trustworthy Computing (Proceedings of the 5th International Conference TRUST)*, volume 7344 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2012.

[96] S. Podesser and R. Toegl. A software architecture for introducing trust in Java-based clouds. In J. Park, J. Lopez, S.-S. Yeo, T. Shon, and D. Taniar, editors, *Communications in Computer and Information Science*, volume 186. Springer Verlag, 2011.

[97] J. Poritz, M. Schunter, E. V. Herreweghen, and M. Waidner. Property attestation: Scalable and privacy-friendly security assessment of peer computers. Technical report, IBM Research, 2004.

[98] J. A. Poritz. Trust[ed|in] computing, signed code and the heat death of the internet. In *Proceedings of the 2006 ACM Symposium on Applied computing*, SAC '06. ACM, 2006.

[99] G. Proudler. Technical Solutions to the Problem of Trust in Computing – A look backwards and forwards. *Future of Trust in Computing, Berlin*, 2008.

[100] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[101] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 1978.

[102] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), 2012.

[103] J. Rushby. A trusted computing base for embedded systems. In *Proceedings 7th DoD/NBS Computer Security InitiativeConference*, pages 294–311, Gaithersburg, MD, 1984.

[104] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *Proceedings of the 2004 Workshop on New Security Paradigms*, NSPW '04. ACM, 2004.

[105] D. Safford, J. Kravitz, and L. v. Doorn. Take control of TCPA. *Linux Journal*, 2003(112):2, 2003.

[106] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, San Diego, CA, 2004. USENIX Association.

[107] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[108] V. Scarlata, C. Rozas, M. Wiseman, D. Grawrock, and C. Vishik. TPM virtualization: Building a general framework. In N. Pohlmann and H. Reimer, editors, *Trusted Computing*, pages 43–56. Vieweg+Teubner, 2008.

[109] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW'10. ACM, 2010.

[110] A. Schmidt, N. Kuntze, and M. Kasper. On the deployment of mobile trusted modules. In *Wireless Communications and Networking Conference*, WCNC 2008. IEEE, 2008.

[111] B. Schneier. *Liars and Outliers: Enabling the Trust that Society Needs to Thrive*. Wiley, 2012. ISBN 978-1-118-14330-8.

[112] Security Research Labs. Badusb – on accessories that turn evil. In *Proceedings of Black Hat USA*, 2014.

[113] J. Sheehy, G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, L. Monk, J. Ramsdell, and B. Sniffen. Attestation: Evidence and trust. Technical Report 07 0186, MITRE Corporation, 2007.

[114] E. Shi, A. Perrig, and L. van Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2005.

[115] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06. ACM, 2006.

[116] D. Slamanig. Dynamic accumulator based discretionary access control for outsourced storage with unlinkable access. In A. D. Keromytis, editor, *Financial Cryptography and Data Security (FC 2012)*, volume 7397 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2012.

[117] D. Slamanig. Efficient schemes for anonymous yet authorized and bounded use of cloud resources. In V. S. Miri, Ali, editor, *Selected Areas in Cryptography (SAC 2011)*, volume 7118 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2012.

[118] S. W. Smith. *Trusted Computing Platforms: Design and Applications*. Springer Verlag, 2005. ISBN 0-387-23916-2.

[119] M. Strasser and H. Stamer. A software-based trusted platform module emulator. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Trusted Computing - Challenges and Applications*, volume 4968 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2008.

[120] C. Tarnovsky. Hacking the smartcard chip. Blackhat DC'10, 2010.

[121] The Trusted Computing Platform Alliance. Building a foundation of trust in the PC, 2000. TCPA Whitepaper.

[122] A. Thurston. Ragel state machine compiler. `http://www.complang.org/ragel/`.

[123] R. Toegl. *On Trusted Computing Interfaces*. PhD thesis, Graz University of Technology, 2013.

[124] R. Toegl and M. Pirker. An ongoing game of Tetris: Integrating trusted computing in Java, block-by-block. In D. Grawrock, H. Reimer, A.-R. Sadeghi, and C. Vishik, editors, *Future of Trust in Computing*. Vieweg+Teubner, 2009.

[125] R. Toegl, M. Pirker, and M. Gissing. acTvSM: A dynamic virtualization platform for enforcement of application integrity. In L. Chen and M. Yung, editors, *Trusted Systems (Proceedings of the Second International Conference (INTRUST), Revised Selected Papers)*, volume 6802 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 2010.

[126] R. Toegl, F. Reimair, and M. Pirker. Waltzing the Bear, or: A trusted virtual security module. In S. Capitani di Vimercati and C. Mitchell, editors, *Public Key Infrastructures, Services and Applications, 9th European Workshop, EuroPKI 2012, Pisa, Italy, September 2012, Revised Selected Papers*, volume 7868 of *Lecture Notes in Computer Science*. Springer Verlag, 2013.

[127] R. Toegl, T. Winkler, M. Pirker, M. Steurer, and R. Stoegbuchner. IAIK Java TCG software stack - jTSS API tutorial. `http://trustedjava.sf.net`, 2011.

[128] Trusted Computing Group. TCG web homepage. `https://www.trustedcomputinggroup.org/developers/`, 2003.

[129] Trusted Computing Group. TCG infrastructure reference architecture for interoperability (part i). `http://www.trustedcomputinggroup.org/resources/infrastructure_work_group_reference_architecture_for_interoperability_specification_part_1_version_10`, 2005.

[130] Trusted Computing Group. TCG credential profiles specifications version 1.1, rev 1.014. `https://www.trustedcomputinggroup.org/resources/infrastructure_work_group_tcg_credential_profiles_specification`, 2007.

[131] Trusted Computing Group. TCG software stack (TSS) specification. `http://www.trustedcomputinggroup.org/resources/tcg_software_stack_tss_specification`, 2007.

[132] Trusted Computing Group. TCG specification architecture overview rev. 1.4. `http://www.trustedcomputinggroup.org/resources/tcg_architecture_overview_version_14`, 2007.

[133] Trusted Computing Group. Trusted Computing Group announces certification program starting with trusted platform module certification. `http://www.trustedcomputinggroup.org/media_room/news/35`, April 2009.

[134] Trusted Computing Group. TCG mobile trusted module specification version 1.0 revision 7.02. `http://www.trustedcomputinggroup.org/developers/mobile/specifications`, 2010.

[135] Trusted Computing Group. Do you know? a few notes on trusted computing out in the world. http://www.trustedcomputinggroup.org/community/2011/03/do_you_know_a_few_notes_on_trusted_computing_out_in_the_world, 2011.

[136] Trusted Computing Group. TCG virtualized platform specifications. http://www.trustedcomputinggroup.org/developers/virtualized_platform, 2011.

[137] Trusted Computing Group. TPM Main part 1: Design principles. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 1 March 2011. Specification version 1.2 Level 2 Revision 116.

[138] Trusted Computing Group. TPM Main part 2: Structures. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 1 March 2011. Specification version 1.2 Level 2 Revision 116.

[139] Trusted Computing Group. TPM Main part 3: Commands. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 1 March 2011. Specification version 1.2 Level 2 Revision 116.

[140] Trusted Computing Group. TPM main specification level 2 version 1.2. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2011.

[141] Trusted Computing Group. PC client work group specific implementation specification for conventional bios specification, version 1.2. https://www.trustedcomputinggroup.org/resources/pc_client_work_group_specific_implementation_specification_for_conventional_bios_specification_version_12, 2012.

[142] Trusted Computing Group. Trusted platform module library family 2.0, level 00 revision 00.93. http://www.trustedcomputinggroup.org/resources/trusted_platform_module_specifications_in_public_review, 2012.

[143] Trusted Computing Group. PC Client TPM certified products list. http://www.trustedcomputinggroup.org/certification/certificationtpm_certified_products_list, 2014.

[144] Trusted Computing Group. TPM library specification familiy 2.0. http://www.trustedcomputinggroup.org/resources/tpm_library_specification, 2014.

[145] Trusted Computing Platform Alliance. Main specification version 1.1b. hhttp://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2002.

[146] J. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. Technical report, Carnegie Mellon University, 1991.

[147] Various Authors. BusyBox: The swiss army knife of embedded linux. http://www.busybox.net/.

[148] Various Authors. Embedded Gentoo. http://www.gentoo.org/proj/en/base/embedded/.

[149] Various Authors. IcedTea project. http://icedtea.classpath.org/.

[150] Various Authors. Java new operating system design effort (JNode). http://jnode.org/.

[151] Various Authors. LVM2 resource page. https://sourceware.org/lvm2/.

[152] Various Authors. Ruby: A programmer's best friend. https://www.ruby-lang.org/en/.

[153] Various Authors. Sanos operating system kernel. http://www.jbox.dk/sanos/.

[154] Various Authors. The Syslinux project. http://www.syslinux.org.

[155] Various Authors. TrouSerS - the open-source TCG software stack. http://trousers.sourceforge.net/.

[156] Various Authors. uClibc: A C library for embedded linux. http://uclibc.org/.

[157] Various Authors. OpenAttestation Project. https://github.com/OpenAttestation, 2012.

[158] A. Vasudevan, J. McCune, N. Qu, L. van Doorn, and A. Perrig. Requirements for an integrity-protected hypervisor on the x86 hardware virtualized architecture. In A. Acquisti, S. Smith, and A.-R. Sadeghi, editors, *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (TRUST)*, volume 6101 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2010.

[159] C. Wachsmann, L. Chen, K. Dietrich, H. Löhr, A.-R. Sadeghi, and J. Winter. Lightweight Anonymous Authentication with TLS and DAA for Embedded Mobile Devices. In M. Burmester, G. Tsudik, S. Magliveras, and I. Ilic, editors, *Information Security, 13th International Conference (ISC 2010)*, volume 6531 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2010.

[160] D. Wallom, M. Turilli, G. Taylor, N. Hargreaves, A. Martin, A. Raun, and A. McMoran. myTrustedCloud: Trusted cloud infrastructure for security-critical computation and data managment. In *IEEE 3rd International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011.

[161] J. Winter. Trusted computing and local hardware attacks. 2014. Master's thesis at IAIK, Graz University of Technology.

[162] J. Winter and K. Dietrich. A hijacker's guide to communication interfaces of the trusted platform module. *Computers & Mathematics with Applications*, 65(5), March 2013.

[163] J. Winter, P. Wiegele, M. Pirker, and R. Toegl. A flexible software development and emulation framework for ARM TrustZone. In *Proceedings of the Third international conference on Trusted Systems (INTRUST'11)*, volume 7222 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2012.

[164] R. Wojtczuk and J. Rutkowska. Attacking Intel Trusted Execution Technology. Technical report, Invisible Things Lab, 2009.

[165] R. Wojtczuk and J. Rutkowska. Attacking intel TXT via SINIT code execution hijacking. Technical report, Invisible Things Lab, 2011.

[166] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another way to circumvent Intel Trusted Execution Technology. Technical report, Invisible Things Lab, 2009.

[167] L. Xi, K. Yang, Z. Zhang, and D. Feng. DAA-Related APIs in TPM 2.0 revisited. In T. Holz and S. Ioannidis, editors, *Proceedings of 7th International Conference on Trust and Trustworthy Computing (TRUST)*, volume 8564 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2014.

[168] Xilinx Inc. Spartan-3A/3AN FPGA starter kit board user guide. `http://www.xilinx.com/support/documentation/boards_and_kits/ug334.pdf`.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)