Graz University of Technology
Faculty of Computer Science
Institute for Applied Information Processing and Communications
IAIK

On Trusted Computing Interfaces

A PhD Thesis Presented to the Faculty of Computer Science in Fulfillment of the Requirements for the PhD Degree

by

Ronald Tögl

Assessors:
Prof. Roderick Bloem, PhD
Dr. Andrew Martin

November 2013

On Trusted Computing Interfaces

by

Ronald Gregor Tögl

A PhD Thesis
Presented to the Faculty of Computer Science in Partial Fulfillment of the
Requirements for the PhD Degree

Assessors

Prof. Roderick Paul Bloem, PhD (Graz University of Technology, Austria)
Dr. Andrew Martin (University of Oxford, United Kingdom)

November 2013

Graz University of Technology

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

Abstract

Trusted Computing platforms are general purpose computers augmented with hardware-based security mechanisms such as the Trusted Platform Module to protect software services. The objective of this thesis was to study the interactions of security mechanisms and services, developers and users and the Internet by focusing on the interfaces that connect them.

First, a Trusted Computing Application Programming Interface is proposed to allow Java programmers easy access to the features of the Trusted Platform Module. Based on specific requirements and goals, a high-level design is derived. The Application Programming Interface has been standardized as JSR 321 in the Java Community Process. Standardization itself has been performed in an open, transparent way and supported by implementations and intensive testing.

Second, a wireless interface is proposed for the Trusted Platform Module to serve as direct communication channel to the user. Through Near Field Communications, the trustworthiness of public kiosk computers can be queried with the help of a commodity smart phone. A protocol and modifications to the Trusted Platform Module are proposed and evaluated in a series of experiments and prototypes. Alternative approaches and potential improvements are discussed.

Third, formal methods are applied to services that build upon the Trusted Platform Module to investigate the security of cryptographic protocols and of a virtual security module. A protocol that leverages Trusted Computing is studied through model checking, unveiling a potential security issue. Through a rigorous model and security specifications a proposed improvement is proved to be correct. Further, a Trusted virtual Security Module is presented and integrated into a virtualization platform which offers integrity protection and runtime isolation on commodity PC platforms. The module's Security Application Programming Interface is rigorously specified and designed to protect cryptographic key material, especially signature keys. The interface is verified to be robust against logical attacks. The module's implementation is described and benchmarks provided.

The principal result is that by studying the interfaces of Trusted Computing platforms, the overall security can be improved while making programming less complex, user interaction more intuitive and key protection more affordable.

Acknowledgements

This thesis would not have been possible without the encouragement and dedication of many people. I am indebted to all colleagues, co-authors and friends who supported me during the work for this thesis.

I would like to express my deep gratitude to Roderick Bloem who I greatly appreciate for many a thing, among them being the adviser of this thesis. I am very thankful to Andrew Martin for welcoming me to a visit to Oxford and for being the external assessor of this thesis. I thank all past and present members of the Secure and Correct Systems Group at IAIK for countless, both fruitful and enjoyable discussions. Special thanks go to Martin Pirker for many years of joint work and research.

A googol thanks go to my family, especially Angelika, Daniel, Gero, Lennart, Reinhard, and Thea. Without them, this work would not have been possible at all. Finally, I would like to thank my girlfriend Sabine for her encouragement, love and faith.

Ronald Toegl
Graz, November 2013

Table of Contents

List of Tables

List of Figures

Acronyms

ACM	Authenticated Code Module
ARM	Acorn RISC Machine, a micro-architecture
AES	Advanced Encryption Standard
AIK	Attestation Identity Key
API	Application Programming Interface
BIOS	Basic Input/Output System
CPU	Central Processing Unit
DAA	Direct Anonymous Attestation
DH	Diffie-Hellman(-Merkle) key exchange
DMA	Direct Memory Accesses
DRTM	Dynamic RTM
EAL	Evaluation Assurance Level
EC	Executive Committee (of the JCP)
ECC	Elliptic-Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
EG	Expert Group (of the JSR)
EK	Endorsement Key
EM	Electromagnetic
FSM	Finite State Machine
HMAC	Keyed-Hash Message Authentication Code
HSM	Hardware Security Module
I^2C	Inter-Integrated Circuit (a computer bus)
IDE	Integrated Development Environment
IT	Information Technology
JCP	Java Community Process
JIT	compiler Just-In-Time compiler
JNI	Java Native Interface
JRE	Java Runtime Environment
JSR	Java Specification Request
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
LCP	Launch Control Policy
LOC	Lines Of Code
LPC	Low Pin Count bus
MAT	Mobile Attestation Token
MLE	Measured Launch Environment

MTM	Mobile Trusted Module
NFC	Near Field Communication
NIST	National Institute of Standards and Technology (of the USA)
OS	Operating System
PAL	Piece of Application Logic
PCB	Printed Circuit Board
PCR	Platform Configuration Register
PDA	Personal Digital Assistant
PDF	Portable Document Format
PKCS	Public-Key Cryptography Standards
PKI	Public Key Infrastructure
RAM	Random Access Memory
RF	Radio Frequency
RI	Reference Implementation (of the JSR)
RPC	Remote-Procedure Call
RSA	Rivest, Shamir and Adleman (crytographic algorithm)
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
SHA-1	Secure Hash Algorithm 1 (a NIST standard)
SML	Stored Measurement Log
SOAP	Simple Object Access Protocol
SoC	System-on-Chip
SRK	Storage Root Key
SRTM	Static RTM
SVM	AMD Secure Virtual Machine
TBS	TPM Base Services
TC	Trusted Computing
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCK	Technology Compatibility Kit (of the JSR)
TCP	Transmission Control Protocol
TCPA	Trusted Computing Platform Alliance
TCS	TCG Core Services (of the TSS)
TDDL	TCG Device Driver Library (of the TSS)
TIS	TPM Interface Specification
TLS	Transport Layer Security
TPM	Trusted Platform Module
TSP	TCG Service Provider (of the TSS)
TSS	TCG Software Stack
TvSM	Trusted virtual Security Module
TXT	Intel Trusted eXecution Technology
USB	Universal Serial Bus
UUID	Universally Unique Identifier
VLP	Verified Launch Policy

VM	Virtual Machine
vTPM	virtual TPM
x86	Intel 8086-compatible micro-architecture
XML	Extensible Markup Language

1 Introduction

1.1 Introduction

The open and flexible design of today's computers has been the driver of the information age. Yet, these computers often lack security, and countless successful attacks result in loss of users' data, severe damage to enterprises and failure of critical infrastructures.

In *Trusted Computing*, specialized mechanisms are added to computer systems to improve their security. These security features are either integrated into existing hardware components, added through extra hardware devices, or programmed into the operating system and the software it supports.

Trusted Computing mechanisms are therefore embedded into conventional, feature-rich and vulnerable systems. The isolation of these mechanisms from the surrounding 'sea' of features creates small 'islands' of security, which may well be trusted with a high level of confidence. However, unlike to geographical islands, there has until now been only limited interest in the 'beaches', the interfaces that separate the Trusted Computing mechanisms from the encompassing platform, the outside network and the users.

This thesis examines these *interfaces* in the light of computer security research, while being based on prototypes and experiments. We initially present the state of the art of Trusted Computing and then move on to study three distinct aspects of interfaces for Trusted Computing. First, we present an innovative application programming interface which serves as gateway for application software and ultimately the human developer. Second, we propose mechanisms that help in the interaction with the user, so that he can decide to trust public computer terminals, or not. Third, we study how a software's interface towards

Declarations
The curriculum for doctoral studies in technical sciences of Graz University of Technology requires each dissertation to contain a commented list of publications.

A list of relevant scientific publications by the author is provided in Appendix B starting on page 151.
Furthermore, each chapter is accompanied by a grey box explaining the chapter's relation to the previous publications of the author. Also, the contribution of co-authors, co-workers and supervised students are declared in those sections.

This chapter extensively adapts, cites and reuses previously published material from the author, especially

[91] K. Dietrich, T. Vejda, R. Toegl, M. Pirker, and P. Lipp. Can you Really Trust your Computer Today? — Emerging architectures for Trusted Computing. *ENISA Quarterly*, 3(3):8–9, Jul–Sep 2007.

The front page illustration is copyright (c) Natascha Eibl and Moritz Lipp, 2013. Used with permission.

the Internet can be designed to guarantee an expected behavior. A discussion of the results concludes the thesis.

In this chapter we introduce our thesis. First, we motivate the basic idea of Trusted Computing and sketch out a Trusted Platform. Second, we identify open research challenges, based on the literature. Third, we discuss what is achievable and present the contributions of this thesis. Fourth, we present the structure of this thesis and give an outlook on the remaining chapters.

1.2 Motivation

Computer and Communications Security is a field as old as (electronic) computing [210] and has made great strides forward over the last decades, towards the provision of confidentiality, integrity and authenticity of data and code for users, enterprises and organizations. Unfortunately, also the complexity and interconnectivity of systems has increased, thus making it ever harder to protect them against a multitude of threats. Overall, the chances of breaching security on a general purpose computing system appear to be higher than ever before.

Instead of pursuing perfect security, major players in the industry have moved to modify the existing system architecture to provide a gradual improvement on platform security. The idea is to create the mechanisms that allow one to find out whether a given platform can be *trusted* for an intended purpose. This approach, which will discuss in much more detail in Chapter 2, is called Trusted Computing

[211]. To amend systems accordingly, a set of functionalities which can give assurance on their configuration and indication of their expected behavior has been added. The most significant of the created technologies is the *Trusted Platform Module* (TPM) [345], specified by the *Trusted Computing Group* (TCG) [346] industry consortium.

The TPM is a small cryptography co-processor, physically bound to a desktop PC, server or mobile device. It is designed to protect selected, critical material from malicious software or remote attackers. The TPM is a passive device, a receiver of external commands. It thus depends on a set of system software that manages its resources and provides well-defined interface for software that intends to use it. One characteristic way to use the TPM is to build a chain-of-trust: here, all software on a platform is broken down into smaller pieces and starting with power-on, all software is examined before it is executed. The so gathered measurement values are stored inside the TPM and can then be reported by the TPM. Analysis of this report will allow another party to decide whether a software service executed is considered secure for an intended task and therefore whether the overall platform can be used securely for this purpose. Alternatively, access to selected data can be restricted to such a trusted configuration.

The basic idea of such a Trusted Computing platform is sketched out in Figure 1.1. In this simplified perspective, we can identify three parties: the user, the Internet which represents any number of other platforms, and one specific Trusted Computing-enabled platform. This platform consists of its hardware assembly, which includes the TPM, and a software configuration, which includes the *Operating System* (OS) and a potentially productive software service. This service needs capabilities to access the TPM to make use of its features such as measurement and reporting. The service will then use the TPM to gather a report on its configuration which it will then present, if challenged, via a network protocol to other services on the Internet. The report should prove the good intentions of the service. Likewise, users wishing to interact with the Trusted Computing platform will need a similar report, in a form that allows them to make a sound trust decision.

Overall, the vision of Trusted Computing promises [32] a number of benefits. Users will be able to authenticate platforms, to have confidence in behavioral integrity and to trust a system which is not under their control. Companies will profit from enhanced security, will have a technological foundation for privacy and will be able to give feedback of this security to the users and to employ trustworthy digital signatures.

1.3 Research Challenges

Proposed a decade ago, Trusted Computing had a slow start. Realizing a Trusted Computing platform is not simple, despite commercial availability of many components and compatible hardware being widely sold [296]. Also, early in the

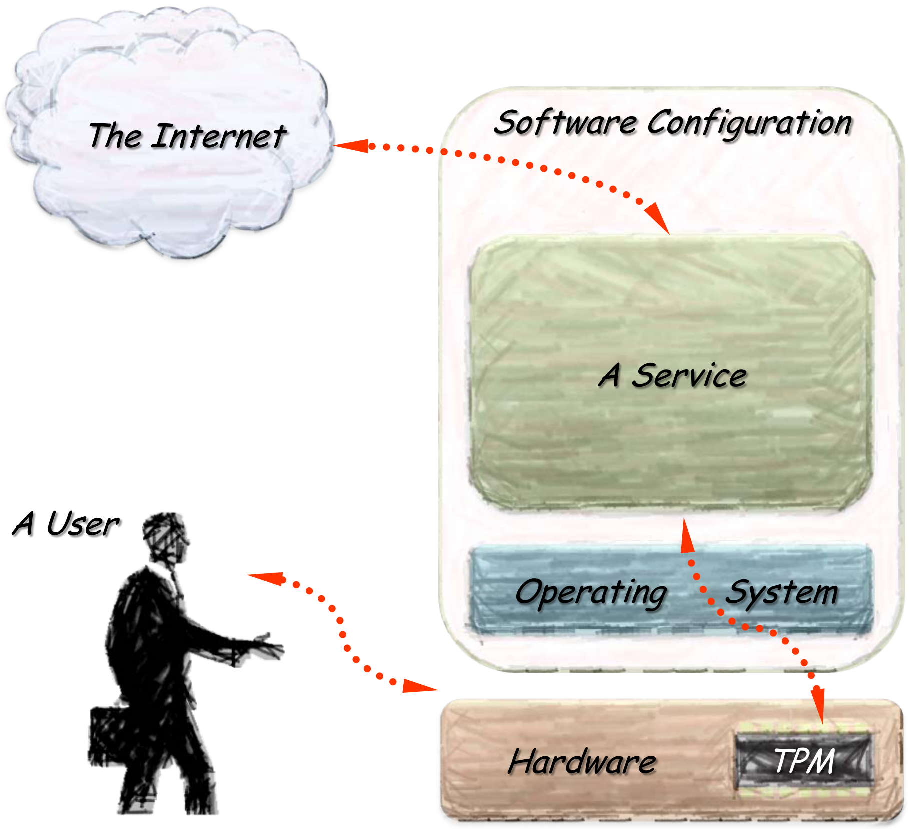


Figure 1.1: A simplified sketch of a Trusted Computing-enabled Computer System with a selection of interfaces highlighted.

development of the Trusted Computing approach, there were a number of general concerns [14] on privacy and on economical and political sincereness, which might have slowed down the deployment.

Even considering this, Trusted Computing has seen such slow an adoption, that suggests that also a number of more technical issues exist. A number of concrete challenges for the Trusted Computing technology have been set out by Balfe et al. [34], and Vishik et al. [354]. A subset of those are that Trusted Computing mechanisms suffer from a low usability, there is a lack of specifications and standards, especially for the interaction between heterogeneous devices and, that digital evidence provided to gain trust should be consumable both by digital entities and human beings.

More specifically, we have identified the following specific challenges from the literature.

- There is a notorious lack of software that actually uses [291] the TPM technology. One reason may be the relatively high complexity of the accompanying technical specifications for programming interfaces, which causes a steep learning curve for programmers and leads to "complicated" [308] programs. Furthermore, there has been a lack of established, standards-compliant interfaces for other programming languages than C.

- Applying the TPM to help users decide whether a public computing device can be trusted with confidential information is difficult, and the user needs another device to help her perform the cryptographic protocols with the TPM. A perfectly trustworthy hardware token as proposed by McCune et al. [216] could perform this task, yet remains an unattainable ideal. Among the open issues here are the challenges of universal physical connectivity while offering resistance against relay attacks and how to decide on the trustworthiness of the highly variable software configurations [105, 149] in today's PCs.

- While for general purpose cryptographic protocols promising automatic tools are emerging [1, 198, 217, 219] that allow protocol designers to find potential attacks, protocols involving Trusted Computing commands are notoriously difficult to assess for their security. For instance, TPM operations might give different results, depending on whether the platform is in a trusted state or not.

- Hardware Security Modules are devices designed to provide maximum security to cryptographic materials, by handling them in a physically shielded environment and only through a software interface that enforces strict security policies. While these specialized modules are typically very expensive, the technology to implement them converges with that of Trusted Computing-enabled general purpose PCs [300]. A software implementation on a modern PC platform thus would be cheaper and still could, to some degree, rely on hardware-based security.

- The *Application Programming Interfaces* (APIs) of hardware security modules, TPMs and Trusted Computing-protected software services should be designed with formal rigor: yet tools to reason formally on stateful components [145] are only emerging and the verification of a full security application programming interface has not been demonstrated yet.

1.4 Contribution

Clearly, there is a need for improvement and novel solutions to these shortcomings and challenges. Yet, in the industry-driven context of Trusted Computing an academic thesis can only do so much; most technologies have to be accepted as they are. The complexities of real-world systems and the economical environment make short-term modification to most of the system components unlikely. However, the way technologies are used can be modified, and the ways they should be applied improved and furthermore, it is the role of academia to propose futuristic concepts, which may or may not find commercial acceptance in the longer term.

Consequently, we do not desire to re-invent the general computer architecture and, instead of re-designing hardly modifiable components, we focus on the gateways between those components, the interfaces that connect them.

Trusted Computing system can profit by novel interfaces, through platform-independent programming interfaces that make using the TPM more effective and efficient, by directly and physically connecting the TPM to a trustworthy user device, and by rigorously defining software service interfaces such that their behavior meets security specifications.

In this thesis, we report on the author's main results achieved in the years 2007–2013, through our participation in several national and international research projects and individual research. Over the course of those years, most of the results have been published in peer-reviewed journals and conference proceedings. We present our contributions from these publications, including extensions and added details in three chapters.

Overall, this thesis presents the following main contributions.

- The design of an easy-to-use, high-level API for the TPM, which is targeting the Java language. The result is a small, compact interface that offers access to the core concepts that together enact Trusted Computing. The API integrates well in the Java environment and takes the needs of application developers into account. It has been shown to be effective in teaching, developing Cloud applications and embedded systems.

- The release of this API as an open industry standard. We not only designed *the* "Trusted Computing API for Java", but also turned it into a standard by completing the difference phases of the Java Community Process. The standardization was done in an innovative way, incorporating elements of agile software development and striving to be as open and transparent as possible. Also, a reference implementation and a thorough test suite were created.

- A protocol and concept for attestation between public computer terminals and smartphones, using a short-range only radio technology. We show, how flexible off-loading of trust decisions to a remote server can benefit the users' right to self-determination in their decision to trust a device. We further show that TPMs could be equipped with a very short range radio interface, by proposing suitable integration in the TPM and an evaluation of the expected over-the-air performance.

- The formal analysis of a cryptographic protocol which uses the TPM to distribute secrets. As convenient push-button tools do not support the stateful TPM API, we apply model checking. We are able to show a potential weakness. Furthermore, we prove that a proposed improvement prevents that protocol failure.

- We present the design and realization of a virtual hardware security module which can be implemented in software only, thus being cheap and fast. Even on commercial-off-the-shelf PCs, its integrity is protected at boot-time and at runtime: we leverage a complete, uninterrupted chain of measurements of the complete software configuration, from firmware to user-mode service; a virtualization platform provides isolation from other services on the same machine.
- The complete security API of this virtual security module is formally verified to provide a comprehensive, correct key policy. We precisely define the API signatures, set key policies and specify which data must remain private. An executive model on a model checker optimized for security analysis allows experiments on API definitions, so that we can exactly determine which policy checks need to be done where in an implementation of the API.

1.5 Outline

The following chapters group these contributions and integrate them into a many-sided account. Figure 1.2 presents the structural outline of this thesis, which consists of five chapters, starting with this introduction. It is followed by a background chapter that serves as basis for the three distinct technical chapters in which we have grouped our contributions. A final chapter presents the conclusions.

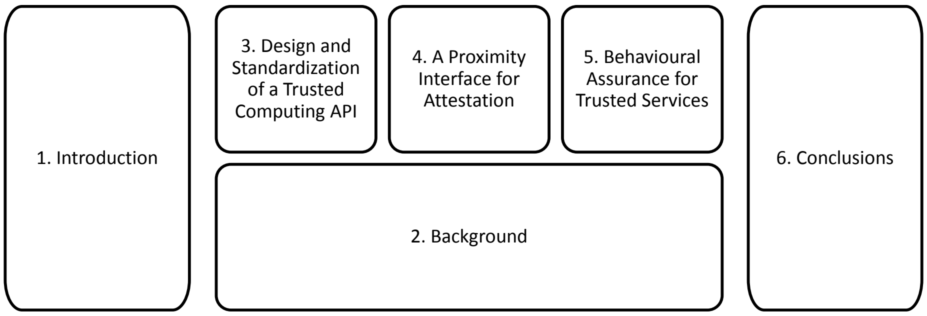

1. Introduction
3. Design and Standardization of a Trusted Computing API
4. A Proximity Interface for Attestation
5. Behavioural Assurance for Trusted Services
2. Background
6. Conclusions

Figure 1.2: Outline of the Structure of this Thesis.

In Chapter 2, we will lay the technological groundwork for all following chapters. The discussion of background will set Trusted Computing in context with computer security and its history, and discuss the very term “trust” in more detail. Then Trusted Computing technologies are introduced, especially the TPM and the TCG Software Stack. We then present system architectures that leverage these mechanisms, and categorize them by the initial measuring component and the use of virtualization. Going from a single platform to a computer network, we give an overview of cryptographic protocols. We continue with a review of methods for the rigorous analysis of protocols especially using automated tools. Thus the state of the art is presented.

Chapter 3 presents the design of an API for the Java language that allows programmers to use the TPM. The need for a novel interface is motivated, especially with regards to the extensive Java environment. We present a review of existing proposals that stem from the TCG Software Stack and high-level alternatives. From this, we derive requirements for a new, high level API. We set technological goals and assumptions on developers using and technologies for implementing the API, with the desire for ease-of-use being paramount. The design itself is influenced by a formal process, as one of the goals of the API is official standardization in the Java Community Process. Our approach to standardization as such is also reported as an attempt in transparency and openness, together with a time-line of events. The chapter then outlines the API's classes and methods and gives a code example. Implementation and testing aspects complete the technical presentation. We report on experiences with the API, including its use in a networked Cloud environment. The chapter concludes with a summary of the process which lead to the successful standardization of this API called *Java Standardization Request #321* (JSR 321).

In the next Chapter, 4, we consider a novel physical interface for interaction between the TPM and a user through a token, i.e. a small mobile device. The idea is to allow users to determine whether a physically present, local machine fulfills their security policy, using their smart phone. After a motivation, we introduce Near Field Communication, a wireless industry-standard interface which we use for our purposes. We then study the literature on attestation in scenarios that consider locality and identify open challenges and propose a possible usage scenario. We then present our concept for a token for attestation which consists of a protocol, of a set of changes to the TPM, and of the integration of the solution. We then report on our validation of selected aspects of the concept, such as implementations of mobile and stationary software, and air interface performance assessments. Several related research results help complement the picture and underline the viability of our proposal: under other performance assumptions, our scheme can be simplified; the attestation token software can also be replaced by a specialized hardware device; mobile phone security mechanisms can be employed to optimize our scheme further.

The work in Chapter 5 has its foundations in the insight that Trusted Computing mechanisms alone cannot guarantee a secure behavior of software services. However, by rigorously applying formal methods on the interfaces of the service, we can then specify a service, that cannot behave maliciously if implemented correctly. We study two cases: First, an existing cryptographic protocol [292] that uses the TPM to distribute secrets to trusted platforms. As available automated tools find their limitations in the stateful definition of trust, we present a manually tailored model of the protocol, and show the existence of a flaw. We

suggest an improved version and show it to be correct.
Second, we present the full design of a virtual security module, which replaces — on commodity desktop hardware — specialized hardware security modules with an isolated and integrity protected software component for handling key material. Our architecture is based on a trusted virtualization platform specifically designed for this purpose and a security module designed for correct behavior. To achieve this, we i) specify the modules API formally, and ii) verify that the API cannot, by accident or manipulation, disclose private key material. This verification is achieved through model checking the key policies of the API. Implementation aspects and performance results are provided as well.

In Chapter 6 we summarize and synthesize the results of the previous chapters. A number of directions for future work is suggested. Finally, we conclude.

2 Background

2.1 Introduction

This chapter provides the technical background for the remainder of this thesis. At first we will put computer security in a historical perspective and study definitions of 'trust'. The subsequent topic will be trusted platform technologies proposed by the industry, such as the Trusted Platform Module and the TCG Software Stack. We will then study different proposed Trusted Computing platforms that seek to apply these technologies, often together with platform virtualization. The chapter then moves on from individual platforms to communicating systems by introducing security protocols in general and their analysis using symbolic formal methods in particular.

Declarations

This chapter extensively adapts, cites and reuses previously published material from the author, especially

[325] R. Toegl, G. Hofferek, K. Greimel, A. H. Y. Leung, R.-W. Phan, and R. Bloem. Formal analysis of a TPM-based secrets distribution and storage scheme. In *Proceedings TRUSTCOM 2008, in: Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 2289–2294. IEEE Computer Society, 2008.

[153] M. Hutter and R. Toegl. Touch'n' Trust: An NFC-enabled trusted platform module. *The International Journal on Advances in Security*, 4(1 & 2):131–141, 2011.

[333] 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*, volume 6802 of *Lecture Notes in Computer Science*, pages 326–345. Springer Berlin / Heidelberg, 2011.

[339] R. Toegl, T. Winkler, M. Nauman, and T. W. Hong. Specification and Standardization of a Java Trusted Computing API. *Softw. Pract. Exper.*, 42(8):945–965, 2012.

2.2 Of Security and Trust in Computing

2.2.1 Secure Computing

Computer security has been a growing concern ever since the first computing machines were created with mostly military applications during the Second World War. Early installations were protected against espionage and sabotage by physical security such as guards and organizational measures like strict confidentiality regulations imposed on the people working there. After the war, early batch processing devices could still be protected with similar measures, even in commercial environments. Yet, with the introduction of time-sharing [306] in the 1950s and 60s, several users would process different data on the same machine at the same time. Consequently, computer systems were extended with hardware and software features that restrict access to data, discern between supervisors and non-privileged users and protect the operating system from manipulations. Saltzer and Schroeder [280] provide an interesting overview of the 1970s' state of the art in system security.

In 1976, Harrison, Ruzzo and Ullman [139] proved desirable security properties, the non-leakage of rights in access control systems, to be undecidable problems. Determining the security of a generic piece of software may therefore be undecidable too.

On a more practical level, the so-called *Orange book* [77,78] provided classification guidelines for the security assessment of system architectures. It has now been superseded by ISO 15408, *Common Criteria for Information Technology Security Evaluation* [76]. In the same decade, also cryptography turned from black magic (secret spy-craft) into an open science with the advent of public standards such as DES [231] and public-key algorithms [92, 103, 272].

For several decades, more and more security features have now been integrated in systems to provide what is traditionally called the "CIA" triad: *Confidentiality*, *Integrity* and *Authenticity* as well as *Non-Repudiation* and *Availability* to computer systems, programs, data and ultimately the user. Textbooks like those by Anderson [15] or Bishop [44] provide a broad introduction of current IT security.

Alas, also the complexity of individual systems has grown exponentially over the last decades [229, 285] and as well as their inter-connectivity with the rise of the Internet. A short look at the level of security currently achieved may be warranted. In 2011, around 2.2 billion people, 32.7% of the world's population, were using the Internet [225]. Despite of all its positive effects, this global network offers the possibility of abuse and malicious exploitation. The Symantec Internet Security Threat Report [113] reports that more than 286 million unique variants of malware were encountered by Symantec products in 2010, and 6253 new vulnerabilities were recorded. In that year, Symantec held less than 9% of the world market of anti-virus products [241], but still recorded an overall of 3 billion malware attacks. These numbers, which hint that on average every user is attacked at least once per year, seem to suggest that after decades of research, computer security on is still an important problem that affects essentially everyone and every general purpose system on the Internet and which is far from solved.

2.2.2 Hardware Security Modules

Software alone is vulnerable to viruses, inadvertent erasing, malicious hackers, and complications from system failures. Physical barriers, such as vaults, and secured entrances are (in most cases) cost-prohibitive due to the expense of the initial investment and the on-going maintenance costs and do not adequately protect against insider attacks. Thus, for security critical purposes, specialized devices that offer relatively high, certified levels of security and assurance are used. Such a *Hardware Security Module (HSM)* is a physical device in form of a plug-in card or an external security device that can be attached to general purpose computers and servers. The goals of a HSM are the secure generation, secure storage, and secure use of cryptographic and sensitive data material. Hardware security modules are separate computing entities, and are often shielded, and able to detect physical manipulation[1] or even feature active countermeasures. HSMs in addition provide logical protection of these materials from non-authorized access.

HSMs can enable an institution to support controlled access to the activation and use of important cryptographic keys. The foremost field of applications is the protection of cryptographic keys that are used to create digital signatures, a cryptographic scheme to demonstrate the authenticity of digital content (see Section 2.3.2). Typically, the content of a hardware module can be backed up to other hardware devices. In case one device is attacked or destroyed, a backup remains either on a duplicate hardware device, in a spare token or smart card.

Anderson presents a concise survey of hardware security co-processors [16] with the market offering a wide variety of functions and performances. The performance achieved often determines the price of a module. A typical smart card based HSM can sign data in a few seconds; some high speed HSMs with cryptographic hardware acceleration reach more than 10 000 signature creations per second. Well-documented commercial designs are those of the IBM 4758 [96,301] and IBM 4764 series [25]. A less commercial, rather open hardware and software design was proposed by Gutmann [136].

HSMs do not offer generic computing services, but are accessed through an *Application Programming Interfaces (API)*, which either follows established standards, or provides custom-built functionality.

2.2.3 'Trusted' Computing

For general purpose, interconnected computers and networks, absolute security seems to be unreachable in theory and practice. Yet, people, businesses and organizations still need to rely on, indeed to *trust* in their computing devices day by day.

So what does 'trust' mean? And how can this concept mostly encountered in social interactions be mapped onto computer technology? In the brief definition of the Merriam-Webster On-line Dictionary [221] "trust" is 1) the "assured reliance on the character, ability, strength, or truth of someone or something" and 2) "dependence on something future", or a "hope". Rousseau et al. [274], suggest a cross-disciplinary definition, that "trust is a psychological state comprising the intention to accept vulnerability based upon positive expectations of the intentions or behavior of another." Even less optimistic, but at least easier to falsify, the US National Security Agency (NSA) defines (reported by [15]) a *trusted* entity as one "whose failure can break the security policy", as opposed to a *trustworthy* entity "that will not fail".

Accordingly, 'trust' does *not* depend on or even promise perfect assurance in all cases. This is especially true for computer security. Here 'trust' should be seen as a certain acceptance of dependence, or technologically, as *a gradual improvement on platform security.* In this spirit, major industry vendors have

[1]For instance, all data in a HSM may be automatically destroyed if the case is opened or an inside sensor detects some light, suggesting manipulation.

decided to modify general purpose computers to contain a security hardware anchor. As a standardization body, the *Trusted Computing Platform Alliance* (TCPA) was founded in 1999 with the mission of "enhancing trust" [317] in the PC platform. Trusted Computing in the TCPA's sense is to provide platforms with "a declaration by a known authority that the platform can be trusted for an intended purpose" [317]. This implies that, to the user, perfect security is not necessarily guaranteed by the platform, but rather that certain expectations should be fulfilled and evidence or reliable statements of this provided. In this context Pearson et al. [250] identified *trust* as a synthesis of social and technological means of providing assurance in the process of forming an opinion. They differentiate between *persistent trust*, which represents long-built confidence in institutions and technologies, and *dynamic trust*, which is formed in the short term depending on ad hoc information on context or configurations. According to [250], a computer system may then be dynamically trusted if it is designed and certified by persistently trusted authorities and if the deciding entity accepts any provided evidence of this as credible and sufficient for the current situation and environment.

Thus, *Trusted Computing* (TC) is a practical approach that adds functionality to hardware and software and provides certification schemes to improve platform security. Describing the technological mechanisms needed for this, the TCPA declared that "a trusted platform should provide at least three basic features: protected capabilities, integrity measurement and integrity reporting" [317]. Protected capabilities are operations with exclusive access to shielded locations in the system, for instance inside a security co-processor. The integrity of software can be measured by calculating a cryptographic check-sum over pieces of binary code. Finally, integrity reporting of such a measurement can consist of a statement about a list of measurements that is vouched for by a suitable entity. The TCPA derived a basic set of specifications and its members started the development of hardware and software.

Yet, this endeavor did not last for long. Incomplete privacy protection and the general fear that Digital Rights Management (DRM) would be widely enforced, soon lead to extensive discussions and harsh criticism in the media and also in academic discourse [14, 352]. The perceived risk was that some of the technologies involved might be used to limit a user's right of self-determination on the software she executes on a platform. In 2003, apparently after losing the public's (persistent) trust, the TCPA dissolved. Its activities were subsequently picked up by the Trusted Computing Group (TCG) [346] under a more open and transparent management. More recent TPM specifications have been designed to protect the privacy of users by hiding their platform identifiers through sound cryptographic mechanisms. This unlinking of platform and trust services protects against many types of misuse, but only if the services used follow rights-protecting guidelines.

Today, the Trusted Computing Group defines most relevant technologies and specifications for Trusted Computing. In its current technical specifications (and the TCG glossary [346]), "trust is the expectation that a device will behave in a particular manner for a specific purpose". This is an appealing and quite popular, but still unfortunate definition. First, the statement itself is so vague that is is almost meaningless. Second, the determination of any behavior or non-trivial property of software is just another undecidable problem [271] as difficult as creating perfectly secure programs to begin with.

Yet, the impact of these specifications is significant. Most of the major computer manufacturers are producing desktop and notebook computers containing TCG specified hardware extensions, and the market research company IDC has estimated that nearly 100 million TPM-equipped computers have been shipped in 2010 [296]. Major technology providers like Intel, AMD and ARM also offer vendor-specific mechanisms that build upon, and extend TCG standards or provide competing security features for a wide choice of platforms.

2.3 TCG Hardware Architecture

2.3.1 Trusted Platform Components

The TCG proposes to extend the general purpose x86 PCs, mobile devices and network components with a security subsystem to turn it into a Trusted Platform [32, 346]. The case of a PC is depicted in Figure 2.1, where the components of this subsystem are indicated. The most prominent one is the *Trusted Platform Module* (TPM), a security co-processor. The Trusted Platform Module offers trusted functionality to the BIOS and operating systems. The TPM acts as *Root of Trust for Storage* (RTS), by offering a set of protected memory registers and contains a *Root of Trust for Reporting* (RTR), a unique cryptographic identity that can be used to vouch for security reports. The TPM is accompanied by a *Root of Trust for Measurement* (RTM), which is a trustworthy functionality provided either by the BIOS or the chip-set to establish integrity of software. An essential third technical component is a set of *trusted computing software* that integrates the hardware with the operating system, services and applications.

The cooperation of these technical components creates improved security and can help establish dynamic trust in a platform. Persistent trust is provided by following a set of public *standards and specifications* by trusted authorities such as the TCG. Yet, it is arguably difficult to assess the security of a system, unless all code and even the compile tool chain can be inspected. Thompson, in his Turing award lecture [318], pointed out, that the sheer size of modern systems makes full code inspection practically impossible. Instead, in real-world systems, such assurance is provided by trusted, possibly third, parties that perform a best-effort analysis and then give evidence in form of cryptographic *certificates*. Thus, a persistently trusted and capable entity vouches for a certain level of, or rather effort spent on the assessment of, security.

While not a part of the TCG-specified security subsystem, modern trusted platforms often include another important technology: *system virtualization*, which allows for the sharing of the hardware resources between different operating system instances. Virtualization has become a key technology for extending

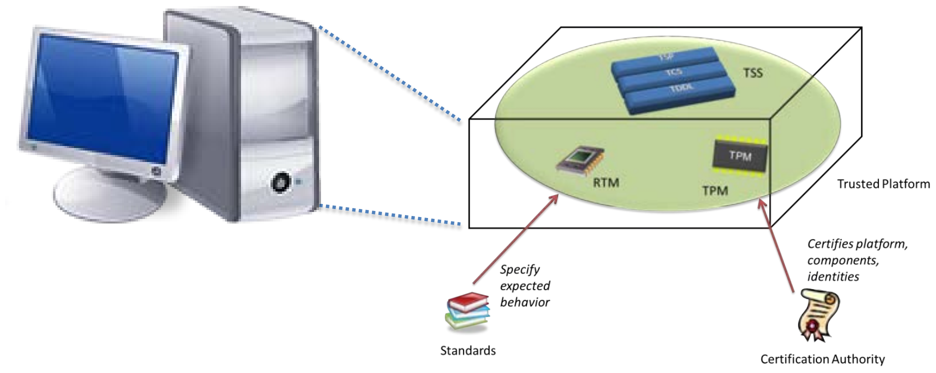


Figure 2.1: A Trusted Platform includes a TCG-specified subsystem containing a Trusted Platform Module, a Root of Trust for Measurement and a TCG Software Stack.

TPM-based security guarantees to operating systems and applications.

The remainder of this subsection is dedicated to the introduction of the most prominent technical components for Trusted Computing and to the security mechanisms they introduce.

2.3.2 Cryptographic Methods used by the TCG

In this section, we will make a brief excursion to the basic cryptographic mechanisms used by the TCG. We will only summarize the basic ideas and will not go further into the cryptological and cryptanalytic aspects, as this would be out of scope for this thesis. Starting points for further reading may be [220, 298].

In symmetric-key cryptography two parties share the same key k. Symmetric key ciphers can be implemented as block ciphers, over a fixed block-length of bits, or stream ciphers, which process bit-streams. The most important examples for block ciphers are the *Data Encryption Standard* (DES) [231] and the *Advanced Encryption Standard* (AES) [234]. Block ciphers can be used in different modes of operation, which results in different characteristics in performance and achievable security goals.

The main drawback of symmetric cryptography is key management, as for each pair of parties a fresh key needs to be agreed upon. Asymmetric key cryptography [92] overcomes this limitation by introducing a public and a private key for each party. While the public key is made known to (arbitrary many) other parties, the private part is kept secret. Only this private key can then perform certain operations. In the RSA [272] cryptographic system, keys are generated by choosing two large primes p, q, computing the modulus $N = p \cdot q$, and choosing an exponent $e, \gcd(e, (p-1)(q-1)) = 1$. (N, e) is the public key. A second exponent $d, e \cdot d \equiv 1(\bmod (p-1)(q-1))$ is calculated, and forms the

private key (d, p, q). For large p, q, it is computationally infeasible to learn d without knowing the factors of N.
The encryption of a message m is the ciphertext $c, c = m^e (\mathrm{mod}\, N)$; it can be done by any party in possession of the public key. Only the holder of the private key can then recover the cleartext $m = c^d(modN)$.

RSA signatures are created analogously to encryption/decryption; the signature s on message m is $s = m^d (\mathrm{mod}\, N)$, which can only be done by the holder of the private key. (m, s) is the signed message. The message m' can be recovered by anyone in possession of the public key: $m' = s^e (\mathrm{mod}\, N)$. The signature is verified by checking if $m = m'$. For real world applications, cryptographic hash functions are applied on m to ensure $m < N$ and the integrity of m.

A cryptographic hash function H maps an arbitrarily long message m on a fixed-size hash value $h, h = H(m)$. It should be computationally unfeasible for an attacker to find a pre-image m' for a given h, a second pre-image $m' : h = H(m) = H(m'), m' \neq\ m$ for for given h, m or any collision $h, h = H(m) = H(m'), m \neq\ m'$. One important example for a cryptographic hash function is SHA-1 [232].

2.3.3 Trusted Platform Module and Core Concepts

In the TCG approach, security is bootstrapped from a small dedicated piece of secure hardware called the *Trusted Platform Module* (TPM). Its specifications, currently at version 1.2 [345], are developed by the TPM Working Group within the TCG.

The TPM hardware component contains a tamper-resilient integrated circuit that implements RSA [272] public-key cryptography, key generation, secure hashing, and random-number generation. Using these elements, the TPM can enforce security policies on hierarchies of secret keys to protect them from software attacks by a remote attacker. As a consequence, the cryptographic keys are only controlled through *handles*, and never directly exposed to the host platform. One of these keys it the Endorsement Key (EK) which provides it with a unique identity, and which is injected by the TPM manufacturer, or created upon machine deployment.

At the time of creation, TPM keys are assigned an internal policy which restricts their use to specific operations. There are key types that discern keys for storage, signature and identity management operations as well as legacy tasks. Keys can also be marked *migratable*, which indicates the possibility to back them up. Migration of keys has the characteristic that private, 'migratable' keys can be sent in encrypted form from one TCG-compliant TPM to another. Still, the key will never be available in clear for anyone but the receiving TPM. For 'non-migratable' keys however, no entity but the creating TPM will be able to access the private key.

Thus, a TPM can be used to perform cryptographic operations, like *signatures* and *encryption* on user-provided data using hardware-protected private keys. However, due to limited TPM memory, keys have to be swapped out of the TPM when not in use. To protect these keys, a parent *storage key* specified upon key creation is used to wrap (encrypt) the private part of the child key when it is exported from the TPM. This implicitly creates an hierarchy of keys, where the different storage keys can be assigned for instance to groups of users thus representing and enforcing organizational structures.

At the top of the key hierarchy is the *Storage Root Key* (SRK), created when the platform's owner initially sets up the TPM. Three Boolean flags control the initialization state of the TPM: Enabled, Activated, Owned. Only after enabling, activating and taking *ownership*, the full functionality of the TPM is available. Clearing ownership is a non-reversible mechanism to revoke all user-created keys by deleting the SRK. At creation time, keys are assigned two user-supplied *secrets* for the purpose of authentication and backup, and optionally a system state that must be attained before using the key for cryptographic operations.

The capability to record the current system state is another main feature of the TPM. This is achieved by cryptographically hashing a software component and storing the resulting *measurement* value in a specially-protected *Platform Configuration Register* (PCR). PCRs are reset at platform boot[2]. PCRs can only be written to via the non-invertible, non-commutative *extend* operation. For each call, a PCR with index i in state t is extended with measurement x by setting

$$\mathrm{PCR}_i^{t+1} = \text{SHA-1}(\mathrm{PCR}_i^t \,||x). \tag{2.1}$$

Thus the TPM's PCRs can potentially be used to exactly describe the software executed on a machine by following a transitive trust model, in which each software component is responsible for measuring its successor before handing over control [121]. For the TCG's technical realization to work, each caller needs to compute a hash value of the expected next executable code and to extend a PCR with the result, before control is passed to that subsequent, and thus measured code. In the simplest case, this is done starting from the BIOS, covering boot loader, kernel, and system libraries etc., up to application code. Ultimately, the exact configuration of the platform is mapped to a set of PCR values; a so-called *chain-of-trust* is established.

To evaluate a system state, a snapshot of the current PCR contents is not enough. Whenever a PCR is extended, an entry is written into a *Stored Measurement Log* (SML). This log then contains all individual steps steps that led to the current PCR values, and represents the software configuration on the system. The TPM-protected PCR-values can prove that the SML is correct.

If such a PCR configuration fulfills the given security or policy requirements, we refer to the system state as a *trusted state*. In the *Quote* operation, the TPM signs these values, thus enabling more complex protocols such as *Remote Attestation* [64, 75, 121, 278, 311]. Here, a remote verifier can analyze the result and decide whether to trust the configuration for a given purpose or not.

A drawback of this scheme is the potential privacy loss, if the same key is used in several Quote signatures which are provided to different service providers.

[2]On more modern platforms (cf. Section 2.5.3) there are more ways to reset PCRs. We will discuss this in more detail in Section 5.7.1.

Collaborating providers might then find undesirable links between different user accounts, if they are used from the same hardware platform. This is especially critical, as each (sufficiently initialized) TPM contains a strong identifier, the cryptographically unique EK. On the one hand, to protect the platform owner's privacy in such a scenario, the EK must not be directly used for the Quote signature. On the other hand, any Quote signature is worthless, if there is no proof that the PCR values signed are protected by a real, TCG-compliant TPM. To this end, a Public Key Infrastructure (PKI) [6] is needed to link private to public keys, and to establish the identities of stakeholders, such as the TPM manufacturers.

For users' keys, however, a special pseudonym is used, which nonetheless certifies that the signing key is protected by *some*, yet actual TPM: an Attestation Identity Key (AIK). AIKs are highly restricted and can only be used to sign TPM internal data structures. The authenticity of an AIK can be certified by an on-line trusted third party, called PrivacyCA (see [255] for more details). An alternative to the PrivacyCA protocol is the the more complex group-signature based DAA [52] scheme.

This protocol allows a TPM to prove that it is within a group of TCG-compliant TPMs without revealing its precise identity, the EK. However, the scheme described in the TPM and TSS specifications has been found to be complex to use and very slow in practical implementations [89]. AIKs can also be used to *certify* that another TPM-based key is actually under the protection of the TPM and confirming to its key policy.

Data can be encrypted by the TPM mainly using two mechanisms, binding and sealing. *Binding*, which is done in software, potentially even on a remote host, is the encryption of a limited amount of data with a public RSA key using either PKCS #1 version 1.5 [172] or OAEP [40] paddings. If the corresponding private key is unique and held by a TPM this implies that only this TPM can decrypt the data. In an even stronger mechanism called *sealing*, the encryption is performed on-chip incorporating a unique secret[3] and a set of PCR registers. Sealed data can only be unsealed by precisely the same TPM in the desired PCR configuration. Data may be sealed to a specific, but also possible future set of values of the PCRs. Thus, access to the data can be restricted to a single trusted state of the TPM's host computer. Overall, the TPM 1.2 API feature 125 *ordinals* which discern the different function calls, for example `TPM_Quote()`.

Further useful features of the TPM are a true random number generator, a tick counter that can potentially be correlated with real-time, and a monotonic counter mechanism.

Currently the TCG is considering[4] new or updated features for the next generation of TPMs, such as cryptographic algorithm agility, higher performance and simplified support for virtualization. Further modifications may be made to TPM lifetime management, the key hierarchy and the privacy protecting

[3]The internal `tpmProof` random number is freshly generated at taking ownership, and only known to the TPM.

[4]The author is member of the TCG's TPM working group and therefore obliged not to disclose further details.

mechanisms. Chen and Ryan [66] recently proposed an improved authentication protocol which will also be part of future specifications.

For mobile platforms, the *Mobile Trusted Module (MTM)* [343] has been specified. The MTM standard is more flexible as it allows defines different sets of features (profiles) that implementers can choose from. All of these profiles cover the same core concepts as explained above.

TPM 1.2 are intended as distinct security co-processor hardware, and there are rather well protected (cf. 2.7), Common Criteria EAL-4 certified and tested products on the market from vendors of different geopolitical origins. For MTM and TPM 2.0 different implementation approaches are permissible. They range from discrete security controllers, over simulation in the chipset to pure software constructs protected by secure CPU modes (In Section 4.5.3 we will briefly discuss a possible implementation technology.).

2.4 TCG Software Architecture

The working groups of the TCG intended the TPM to be implemented in a cost effective way, for instance on smart card architectures with very restricted resources. Consequently, the functionality of the TPM is restricted to a predefined set of operations; the TPM is not able to execute user code, and even most of the mechanisms offered require auxiliary functionality to be implemented in the software of the host platform.

The *TCG Software Stack (TSS)* [342] is responsible to access and manage the TPM and also to provide a programming interface for TC applications. The standard document is accompanied with C header and WSDL interface definition files. The target language for the standard is the C programming language [167].

The TSS offers a set of function calls that help perform a number of operations. These functionalities cover the setup and *administration* of the TPM, such as taking ownership, setting of configurations or querying properties. With regards to the chain-of-trust, it is the task of the TSS to record the SML for tracing the measurements that led to the current PCR values. The life cycle of *cryptographic keys* is also controlled through the TSS, starting with the triggering of the creation of public-private key pairs inside the TPM. The limited resources of the TPM necessitate external, encrypted storage of the cryptographic material, either at runtime by swapping out keys from the limited hardware key slots into main memory or filing in *persistent storage* on the hard disk. Furthermore, the TSS supports different mechanisms of *key certification* and the key *migration* protocols.

The TSS is also specified to facilitate the Identity Management operations. AIKs are created in a process that involves the TPM and the trusted third party PrivacyCA. Here, the TSS is the entity that collects all required information and certificates to assemble the appropriate data structures for communication between the local TPM chip and the remote PrivacyCA service. In [255] we, together with Pirker, present more details on the scheme and an implementation of this service. The alternative protocol of Direct Anonymous Attestation [52],

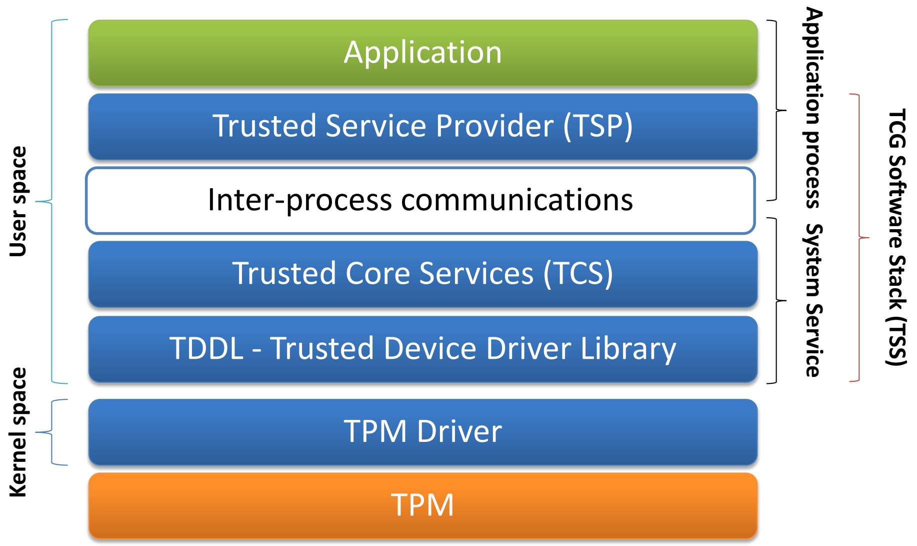

User space
Kernel space
Application
Trusted Service Provider (TSP)
Inter-process communications
Trusted Core Services (TCS)
TDDL - Trusted Device Driver Library
TPM Driver
TPM
Application process
System Service
TCG Software Stack (TSS)

Figure 2.2: The TCG Software Stack (TSS) architecture consists of several software layers within a trusted platform.

is also mainly performed by the TSS, with only a small, yet critical subset of operations being performed by the TPM.

The TSS also provides interfaces to sign user data using TPM-protected keys, which were generated with type information that allows them to perform RSA-signature operations. The TSS also supports binding and sealing by marshalling the payload data into the appropriate TPM data structures.

From a software engineering perspective, the TSS specification define a layered architecture shown in Figure 2.2. Just below the TSS, and not part of it, is the TPM driver. The TPM driver can be either vendor specific or follow the TPM Interface Specification (TIS) standard [344]. It is the task of the lowest layer of the TSS to abstract this driver and expose an *Operating System* (OS) and vendor independent set of functions that allows the upper layer basic interactions with the TPM. This lowest layer is called the *TCG Device Driver Library* (TDDL). The TDDL serves as a single-instance, single-threaded component and allows for sending commands as byte streams to the TPM and receiving the responses.

The next layer up, the *TCG Core Services* (TCS), should be implemented as a singleton system service or daemon. It is the single instance that manages the TPM's resources and accesses it. It generates synchronized command streams from concurrent API commands to be transferred through the TDDL. The TCS takes care of the management of TPM key slots as well as permanent storage of TPM key material. Keys are assigned a *Universally Unique Identifier* (UUID) [189] that is used to identify stored keys. The TCS also maintains the SML where all PCR extend operations are recorded. The upper layers of the software stack may access the TCS via inter-process communications according to the platform-independent *Simple Object Access Protocol* (SOAP) [355] interface. The SOAP interface is standardized in the form of a *Web Service Description Language* (WSDL) [355] by the TCG.

The highest layer, the TCG Service Provider (TSP) provides Trusted Computing services to applications in the form a shared library. The TSP interface is defined as grouped function signatures and data structures in the C programming language. The TSS was also designed to allow partial integration with existing high-level API libraries, such as PKCS #11 [275]. This enables the use of the cryptographic primitives provided by the TPM by legacy software. A limitation of this approach is that these legacy cryptographic APIs do not account for advanced Trusted Computing concepts such as sealing. Also the TCG's key typing and padding policies need to be considered [63] and might not match all application areas.

Recent years have seen successful integration of generic TPM 1.2 hardware drivers into major operating systems. Several proprietary implementations and one open source implementation, IBM's TrouSerS [156], of the TSS in C exist.

2.5 Trusted System Architectures

In this section we will review how trust can be established in software, especially considering the TCG mechanisms. More specifically, we will consider how trust can be provided to processes or even whole operating system instances including application services. In the nomenclature of the Orange Book [77, 276], each service has an individual *Trusted Computing Base* (TCB), which is defined as the sum of all hardware, firmware and software which influences the service's integrity and behavior. The TCB can consist of previously executed and currently active components. Ideally, the TCB should be as small as possible as a smaller code size might happen to contain fewer bugs, be easier to inspect and pave the way towards formal verification. Yet in practice, the TCB is rather large in many general purpose systems, and not likely to offer strong guarantees on security. Instead, the "TCB" often becomes a nebulous collection of millions of lines-of-code, executed in nearly random order.

On Trusted Computing platforms, the TPM can potentially serve as measuring device to collect information on a TCB: The complete TCB can be put into the chain-of-trust collected at load-time with the help of the PCRs. The PCRs will then represent a software configuration that can be analyzed and potentially be trusted for selected purposes. Given dynamically trusted software running on top of an actually *trusted* computing base in isolation from other pieces of software, it may be plausible to assume the expected behavior. We will now review the literature on system architectures proposed to achieve such trusted software.

The TPM's PCR mechanism will exactly document the software executed, if applied rigorously. This allows one to widen the trust in a dedicated, trustworthy hardware security module (the TPM) onto the operating system and services of a general purpose platform. A trustworthy Root of Trust for Measurement (RTM) serves as initial starting point for this process. RTMs are either a component of the *static* BIOS software or the result of a *dynamically* invoked special CPU instruction. Remember, that from the root onwards, the measurement process then needs to continually cover the complete boot procedure across the operating system and into the application layer. To create a software architecture for achieving such a full coverage while maintaining a meaningful state in the PCRs a non-trivial challenge that has lead to several generations of experiments and prototypes.

2.5.1 Static Chain-of-Trust

The first code entity that starts the measurement chain is called the *Static Root of Trust for Measurement* (SRTM). This first piece of software executed after power-on is typically a component of the BIOS, which will measure and initialize other BIOS modules, and which will in turn prepare the execution of the bootloader; of course, the bootloader is measured as well. When control is passed over from the BIOS to to the boot loader, the bootloader takes over the responsibility to measure the next component, i.e., the operating system kernel. By implementing the measurement mechanism in each following software component, a continuous chain-of-trust can be forged.

Hardware manufacturers provide BIOS support to measure the first phases of system boot into the TPM. Boot loaders such as TrustedGrub [185] demonstrate TPM integration. Ideally, the bootloader should pass control over to an Operating System (OS) that establishes at least a partial chain-of-trust.

There have been several attempts to achieve such a coverage. A historical example of extending the trust from dedicated hardware security modules into applications on a custom microkernel is given in the Dyad System [348] of 1991-4. AEGIS [19] is an early mechanism to support secure boot on PC platforms for conventional OSes, assuming a trusted BIOS. After the introduction of the TPM, the Enforcer platform [209] showed integrity protection mechanisms that can be applied as basis for possible security applications such as the Bear software security processor [205]. Similarly, IBM's Integrity Measurement Architecture [279] integrates PCR measurements of file accesses in a Linux environment.

This *first generation* of trusted has experimented with hardware-guaranteed integrity measurement, and it is possible to collect precise, bit-wise information on a system configuration. Yet there is a number of practical problems that prevent to use the gather data and preclude the widespread use for real-world applications.

- Different services are only separated on the operating system-level, through process separation and memory protection. A single implementation error, in the OS kernel or other privileged components, could potentially open an attack path from one process (or the network) to others.

- Measurements are taken file-by-file, whenever they happen to be accessed. This often results in a large number of individual hashes in no particular order. Using the sealing mechanism then becomes almost impossible, as the same configuration of software elements may hash to different values according to external factors such as user inputs, platform temperature (affecting the CPU speed, thus timing of events), or network events.
- Measuring a large number of files also causes problems with remote attestation, as the number of possibly good configurations becomes very high. Keeping track of known good system PCR configurations is a challenge and reaching a trust decision by using only the quote result and SML is a tedious and complex task, especially as there is no data available on which configurations are actually, and in general "good". The number of possible combinations of *white-listed* secure software configurations in today's open system architectures is very large and quickly growing [105] in general purpose systems. However, for specific use cases such as servers for Web-services Lyle and Martin point out that only a limited number of combinations need to be considered [203].
- Firmware code influences the static chain-of-trust as the individual hardware's BIOS, including option ROM, is part of the PCR state. Replacing a hardware component or updating its firmware will influence the PCR configuration. This precludes the definition of trusted configurations, for instance when sealing to a future state through software running on a platform prior to the hardware upgrade. Also a perfectly fine software configuration can be rendered untrusted when a failing hardware components is replaced.

2.5.2 Platform Virtualization

A technology that helps to reduce the length and complexity of the chain-of-trust and therefore eases analysis of system states is *platform virtualization*. Virtualization is a methodology of dividing the resources of a computer into multiple execution environments, by applying concepts such as time-sharing [306], hardware and software partitioning, machine simulation or emulation. Hardware architectures can be designed to offer complete virtualization [263] in hardware and then host multiple unmodified operating systems in parallel. Since 2005, the PC platform has been modified accordingly. Adams et al. [7] provide an overview of x86 virtualization approaches.

Commonly, virtualization is controlled by a singleton *hypervisor*, a superior control entity which directly runs on the hardware and manages it exclusively. It enables the creation, execution and hibernation of isolated *compartments*, each hosting a guest operating system and the applications building on it. Often, a specific application is combined together with a trimmed down OS into a preconfigured *virtual appliance*. Virtual appliances are eliminate the installation, configuration and maintenance overhead which results from managing individual configurations.

A hypervisor-based system provides multiple isolation layers: Conventional processor privilege rings and memory paging protect processes executing within a compartment. Hardware support for monitoring all privileged CPU instructions enables the hypervisor to transparently isolate virtualization instances from each other. Finally, the chip-set is able to block direct memory accesses (DMA) of devices to defined physical memory areas, thus allowing the hypervisor to control device I/O and assign devices exclusively to individual virtual machines.

Early demonstration of systems that make use of virtualization for security and trust are PERSEUS [251] and Terra [119]. The Nizza virtualization architecture [297] extracts security critical modules out of legacy applications, transferring them into a separate, trusted partition with a small TCB. Those early platforms did not actually use TPM features, thus an attestation whether the isolation mechanisms are configured correctly or are actually in use at all is not possible.

The advantages of combing a static chain-of-trust with virtualization are rather reflected in a number of later platforms which form a *second generation* of trusted platforms. Microsoft had plans for a Next-Generation Secure Computing Base [106], with the trusted Nexus kernel providing an environment for security critical services, while running a legacy OS in parallel. Actually commercially available from Sirrix AG is the EMSCB [104] TURAYA platform showcasing TPM-based Trusted Computing on an L4-based [192] virtualization platform.

Another research project, Open_TC [239], demonstrated a system based on a static chain-of-trust from the BIOS to the boot-loader via Trusted Grub to Xen [36] or L4 hypervisors, and into application partitions measured and loaded from CD images. The similar platform of Clair et al. [68] also considers deployment processes of platforms. Coker et al. [75] describe a Xen-based platform which is more focused on Remote Attestation than secure boot. Schiffman et al. [286] describe an all-layer (Xen hypervisor to application) integrity enforcement and reporting architecture for distributed systems. Cabuk et al. [56] propose to use a software-based root of trust for measurement to enforce application integrity in federated virtual platforms, i.e., Trusted Virtual Domains [60]. Instead of individual files, file system images have been used to transport user software and data with SoulPads [57] or Secure Virtual Disk Images in grid services [124] between virtualized platforms.

These platforms alleviate several issues, as the isolation between services can be improved by placing them in different compartments. The possible attack surface, the hypervisor, then becomes potentially much smaller and thus easier to construct correct and robustly. This containment of services therefore helps in building secure systems. Also, it becomes feasible to measure specialized software in separate compartments; the general purpose operating system needs not be trusted. Depending on how many services use the TPM, it may need to be shared or virtualized (We will briefly revisit this topic in Section 3.5.1.). Other challenges, such as hardware-dependence of the configurations measurement values, remain.

2.5.3 Dynamic Chain-of-Trust

When hardware platforms [8, 132, 133] were introduced that provide strong isolation of compartments on commodity hardware, also more advanced security mechanisms were added. Those platforms also extend the basic TCG model of a *static* chain-of-trust from hardware reboot onwards. In addition, they provide the option of a *dynamic* switch to a trusted system state. A special CPU instruction allows the system to switch into a well-defined secure state and then to measure and to prepare the launch of a piece of software. This is typically a hypervisor, and it can be given full control over the system. Close, hard-wired cooperation of CPU, chip-set and TPM guarantees that the result is accurate. Thus, influences from boot-time-only components such as the BIOS can be prevented.

This is a massive change in the PC architecture, as it introduces completely new CPU modes, different levels of machine (re-)boot and new security aspects in memory management, buses and device management. Consequently, the technology has not been immediately taken up by mainstream operating systems. Instead, there has been a limited number of academic experiments with the technology. Also, from our experience, it has been a challenge for PC system manufacturers to integrate the hardware features correctly into their platforms and for years it has been difficult to obtain hardware that would allow the use of the DRTM without problems[5].

The first experiments were concerned with how to activate the DRTM mechanism. The Open Secure Loader (OSLO) [174] is an OS loader module which implements a dynamic switch to a measured state in the OS boot-chain on *AMD Secure Virtual Machine* (SVM) [8] systems, whereas Trusted Boot (`tboot`) [160] is a loader which achieves this on Intel TXT platforms. This more complex boot process has the advantage to exclude BIOS and firmware-related code from the chain of trust. BIND [295] uses AMD's SVM protection features to collect fine grained measurements on both input and the code modules that operate on it so that the computation results can be attested to. Flicker [214] 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.

Such mechanisms have lead to the creation of a *third generation* of trusted platforms, where the DRTM allows to gain full control over a chain-of-trust and virtualization allows the execution of real-world applications. Vasudevan et al. [350] discuss general requirements for such systems. TrustVisor [213] is a small hypervisor initiated via the 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. Intel's P-MAPS [266] launches a tiny hypervisor parallel to a running OS to protect process integrity, hidden from the OS. The hypervisor authenticates code and data areas, which are protected in-place and can only be accessed via

[5]For instance we were faced with platforms where a call to the DRTM would trap the device in a permanent, non-recoverable error state.

well-defined interfaces. LaLa [123] performs a late launch from an instant-on application to boot a fully fledged OS in the background. BottleCap [179] uses the TPM to securely store capability data for authentication and restricts access to them to isolated, small-TCB Flicker sessions. Vasudevan et al. [351] present the Lockdown platform. It separates untrusted and trusted software in space and time; platform memory is statically split and (software) side channel attacks hindered by executing only either trusted or untrusted code within a time frame. When a secure service is needed, untrusted OSes are suspended through ACPI and control passed on to the trusted environment.

With acTvSM, Pirker and Toegl [253, 254, 333] provide a platform which enforces integrity guarantees to itself as a software platform and the applications and the services it hosts. acTvSM takes advantage of the Linux *Kernel-based Virtual Machine* (KVM) hypervisor, which is operated in a non-trivial security configuration started through `tboot` on TXT hardware. System components and applications are measured at file system granularity. Administrators achieve full control over current and future updated [126] system configurations. We will explain the acTvSM platform and its interaction with Intel's TXT mechanism in more detail in Section 5.8.2.

When compared to previous generations, the DRTM mechanism offers excellent improvements to the concept of a chain of trust. First, BIOS and hardware firmware components need not be considered, as the can be called at any time after the platform has booted. Second, the TCB can be minimal, as the DRTM can be configured to measure and execute a well-defined piece of code in system memory, irrespective of any other previous or later software configuration on the platform. Third, while in previous generations the chain of trust must consider a large number of fine-grained measurements of individual components such as binary files, with virtualization in place, the hypervisor can instead perform a single measurement of an entire compartment image file.

As we will see in Section 5.8.2, these improvements make it possible to achieve deterministic PCR values and offers the chance to calculate the PCR configurations of future trusted states, for instance after a planned hardware or software update. This makes the application of the sealing mechanism feasible for practical scenarios, as the updates that are necessary in practice can be implemented without circumventing the TPM's mechanism. The well-defined PCR values also simplify the comparison against known good values in a Remote Attestation scenario.

However, several open issues with Remote Attestation remain [149]. There is currently no business case for offering a whitelist of trusted software, and even if secure states are enforced, there is the risk of locking the platform into a state that may be desirable to some service provider, or to the user, but potentially not to both of them. There also remains the semantic gap [264] between binary hashes and (provable) security properties. Instead of bit-wise comparison, the attestation of abstract properties would be more desirable [265, 278].

2.6 Security Protocols and their Analysis

In many use cases, establishing a trusted system state on a local host is not enough; trusted endpoints rather serve as precondition to secure communications. In general, achieving a protected information exchange or agreement requires the execution of a multi-party algorithm, a *cryptographic protocol* between two or more hosts on a network.

2.6.1 Cryptographic Protocols and Protocol Failures

Cryptographic protocols [1, 95, 207, 220, 298] use cryptographic mechanisms to achieve one or more security goals such as key agreement or exchange, entity authentication, confidentiality, integrity protection, authentication and non-repudiation of messages.

Authentication protocols serve to establish the identity of an entity across a communication channel and are often used to control access to a resource. The security of an authentication protocol derives from the basis of identification, which can be 1) *something known*, like a password or Personal Identification Number (PIN), 2) *something possessed* like a cryptographic key, a smart card, or a digital passport, or 3) *something inherent*, for instance biometric features. Building on a *single* or *multiple* of these *authentication factors*, the protocols follow the goal of either accepting the identity of a claimant as authentic or terminating the protocol run with rejection. Conventional schemes built on time-invariant passwords are considered to provide only *weak authentication*, while interactive *challenge-response* protocols demonstrate the knowledge of a secret depending on a fresh, random challenge, without actually revealing the secret. As the secret does not leak, it cannot be maliciously replicated and this type of authentication is therefore commonly referred to as *strong authentication*.

Authentication can be unilateral or mutual, i.e., ensuring the identity of all involved parties. *Key establishment* protocols allow two or more parties to agree on cryptographic secrets. Such a secret can either be created by one party and then be *transported* to the others, or be agreed on through a function of the protocol participants' information[6]. A special case is the off-line *pre-distribution* of key materials which involves operational and administrative, and possibly manual, processes.

Cryptographic protocols can be application specific, such as remote attestation, or establish a generic secure channel. Secure channels can be established on various layers of the Open Systems Interconnection (OSI) model [244] such as Internet Protocol Security (IPsec) [175], Secure Shell (SSH) [370] or Transport Layer Security (TLS) [88] for applications.

[6]For instance, in the Diffie-Hellman-Merkle key exchange [92], two parties agree upon a public base g and a large prime p and each choose a secret exponent a, resp. b. After exponentiation, $g^a \bmod p, g^b \bmod p$, the resulting integers are exchanged. It is easy for the parties to calculate the shared secret key $s, s = (g^a)^b \bmod p = (g^b)^a \bmod p$, but infeasible for an attacker without knowledge of a or b.

A popular way to describe cryptographic protocols in computer security engineering is the "Alice and Bob" notation. There, a set of principals — by tradition called Alice (abbreviated A), Bob (B), Charlie (C), etc. — wish to communicate. In our notation, we concatenate protocol elements by a dot '.'.

Other such elements could be keys K, timestamps T, nonces N (random numbers intended for one time use) or messages m. For a public-private key pair (K_A, K_A^{-1}), we assume that all parties know the public key K_A and that they expect principal A to be in possession of the private part K_A^{-1}. For encryption of m under key K_A we write $\mathbf{E}(m)_{K_A}$. For the cryptographic signature with the private key we write[7] $\mathbf{S}(m)_{K_A^{-1}}$. Communication between two parties A, B is written $A \rightarrow B$.

As an instructive example, we will consider the simplified and reduced version [200] of the classical Needham-Schroeder public-key authentication protocol [236] shown in Figure 2.3. The protocol's goal is to establish mutual authentication between the initiator A and an responder B who each possess their public-private key pairs (K_A, K_A^{-1}), (K_B, K_B^{-1}). For each run of the protocol, A selects a unique, fresh nonce N_A and sends it and her identity encrypted under B's public key (Message 1) to B. Only B is able to decrypt this message, and returns N_A together with a nonce N_B of its choice, encrypted under K_A (Message 2). If A receives this message, she assumes that only B has received N_A, as it was protected by K_B. Finally, A returns N_B to B (Message 3) to assure B that A did receive N_B. This last step should assure B of A's identity and thus seems to complete the mutual authentication. We will, however, revisit this example in a little while.

$$
\begin{array}{lll}
1. & A \rightarrow B: & A.B.\mathbf{E}_{K_B}(A.N_A) \\
2. & A \leftarrow B: & B.A.\mathbf{E}_{K_A}(N_A.N_B) \\
3. & A \rightarrow B: & A.B.\mathbf{E}_{K_B}(N_B)
\end{array}
$$

Figure 2.3: The Simplified Needham-Schroeder Public Key Protocol.

As we have seen, the Alice and Bob notation represents the *intended trace* of a protocol run in a compact, human-readable way. Naturally, this notation is a high-level abstraction from the cryptographic primitives and does not cover all assumptions and requirements of the individual steps. Even though many protocols can be described in only a few lines, the creation and analysis of protocols is an error-prone task as attackers may exploit even the most subtle failure or oversight in the specification, design or implementation. A sufficient number of vulnerabilities has been discovered in published cryptographic protocols to cause survey work in this field. Building on Clark and Jacobs' extensive literature review [69], the Security Protocols Open Repository (SPORE) [187] lists 50 protocols, 52% of which have known failures [198]. The AVISPA consortium

[7]Note that RSA encryption and signing are *structurally* the same operation. Consequently, the (usually public) signature verification would effect in decryption of private messages. The same key pair must therefore not be used for both operations. However, even if they are, attacks might not be straightforward due to different hashing and padding schemes.

formalized a library [29] of 70 cryptographic protocols of which 24% show security failures [198]. Given this record, it is hardly remarkably that considering all aspects of a cryptographic protocol has been compared in difficulty to dealing with a mephistophelian opponent [17].

For more earthly applications, a concrete attacker model may be warranted. The intruder model proposed in 1983 by Dolev and Yao [94] is very powerful, yet it offers a practical abstraction that is frequently used [217]. The particular strength of the *Dolay-Yao intruder* is that it is in full control over the network: she is not only has the means to listen to any message sent from any party, but also to prevent messages from being received and capable of injecting messages of its choice. Still, complexity is reduced by the assumption that the underlying cryptographic primitives are *perfect* and that keys and message fields are *atomic*: The model does not cover attacks like statistical analysis or differential cryptanalysis [206], nor attacks through number-theory or on mathematical, stochastic properties of bit-streams generated by the underlying cryptographic algorithms. Thus the intruder is not able to learn partial information of a secret key or message, and the intruder either knows it completely or not at all. It is further assumed that all parties know all public keys. The intruder can read an encrypted message if and only if she knows the correct key. Without the key, no information can be learned about the plain text. Similarly, an intruder can only create signed or encrypted messages with signature or encryption keys it knows. The Dolev-Yao model paves the way for the *symbolic* treatment of protocols.

In general, an attacker can follow different strategies to exploit a cryptographic protocol, of which we now discuss a selection from the literature [220]. In *replay attacks*, a message previously sent out by Alice is re-used at a later point in time, for instance to answer a challenge. A possible reason for a protocol being susceptible to this attack is if there is no time-variant component in the message to guarantee freshness. Two or more concurrent protocol runs can be exploited in *parallel session attacks*, where the intruder uses the messages of one session to meet the challenges in another protocol run. With only one session, this strategy is referred to as *reflection attack*; if the same challenge-response protocol is used in both directions, the attacker can exploit the challenger as oracle by asking the challenger for the expected answers. In a *man-in-the-middle attack* the intruder makes two honest participants believe that they are talking directly to each other, while in fact the entire protocol run is under the control of the attacker. The intruder exploit both parties to gather knowledge and facts to assemble the expected responses.

A famous example of vulnerability against a man-in-the-middle is Lowe's attack [199, 200] on the protocol in Figure 2.3, which was only discovered 17 years after the original proposal was published. The attack is given in Figure 2.4. In this trace, we can observe the following: A tries to establish a session with intruder I, who then moves on to impersonate A in contacting B. In our notation, an intruder I might pretend to B to be another party A, e.g. $I(A) \rightarrow B$, which cannot be detected by B. B answers with the correct response for A, which I forwards to A. A then decrypts B's challenge and encrypts it

$$\begin{array}{lll}
1.1 & A \rightarrow I: & A.I.\mathbf{E}_{K_I}(A.N_A) \\
1.2 & I(A) \rightarrow B: & A.B.\mathbf{E}_{K_B}(A.N_A) \\
2.1 & I(A) \leftarrow B: & B.A.\mathbf{E}_{K_A}(N_A.N_B) \\
2.2 & A \leftarrow I: & I.A.\mathbf{E}_{K_A}(N_A.N_B) \\
3.1 & A \rightarrow I: & A.I.\mathbf{E}_{K_I}(A.N_A.N_B) \\
3.2 & I(A) \rightarrow B: & A.B.\mathbf{E}_{K_B}(N_B)
\end{array}$$

Figure 2.4: Lowe's Attack on the Simplified Needham-Schroeder Public Key Protocol.

with I's public key K_I. I uses its own private key K_I^{-1} to learn N_B and can then assemble the expected answer for B. Thus, A believes to have authenticated I, while B believes that the has successfully completed a protocol run with A. This breaks the goal of mutual authentication between A and B.

The fix provided by Lowe to prevent this failure is remarkable simple; it is sufficient that B explicitly states the sender of the reply (Message 2 in Figure 2.5). This prevents the intruder from forging Message 2.2 (in Figure 2.4) of the attack as A will notice that the identifier in the encrypted part is different from the claimed identity.

$$\begin{array}{lll}
1. & A \rightarrow B: & A.B.\mathbf{E}_{K_B}(A.N_A) \\
2. & A \leftarrow B: & B.A.\mathbf{E}_{K_A}(N_A.N_B.\underline{B}) \\
3. & A \rightarrow B: & A.B.\mathbf{E}_{K_B}(N_B)
\end{array}$$

Figure 2.5: The Improved Needham-Schroeder-Lowe Public Key Protocol.

The resulting protocol is referred to as Needham-Schroeder-Lowe protocol and widely used. Lowe also gave strong assertions that the fix is indeed effective. To this end, he provides a formal proof of correctness of the revised protocol.

2.6.2 Protocol Analysis

Formal methods have been a major driving factor for analyzing and designing cryptographic protocols and have evolved considerably over the last decades. The general idea is that an effective procedure allows to ascertain that a mathematical or logical model of a system meets its security requirements. For broader surveys see [1, 198, 217, 219], but we will now briefly discuss a selection of influential proposals.

Stemming from modal logics, Burrows, Abadi and Needham (BAN) [54] proposed a *belief logic* where the pedigree and freshness of keys match the authentication goals. The BAN security proofs tend to be compact, but it can be a subtle task to consider all assumptions of the proof. Later proof techniques, based on model checking, have unveiled attacks in BAN-verified protocols; for instance, a replay attack has been identified [313] in a version of the Yahalom protocol that was developed as example for security analysis and improvement in the original publication of the BAN approach [54]. Several extensions to BAN have been proposed, but factors like increasing complexity have been a hindrance for widespread success.

An abstraction for security protocols different from the Alice and Bob notation is provided in the SPI calculus [2]. It considers protocols as a number of interacting processes and allows models where the provenance of message components can be defined precisely. Security properties can be performed as equivalence checks. The SPI formalism allows quite insightful pen-and-paper analyses of small protocols.

Instead of reasoning by hand about protocol security, automated formal methods attempt to identify unintended traces, that can by created based on the protocol definition. The main idea is to create an symbolic opponent that performs attacks using the Dolev-Yao model to either construct protocol failures, or prove their absence through search space exhaustion. Formal methods have been subject to automation with efficient implementations appearing over the last fifteen years, with varying success in terms of complexity of protocols that can be modeled, of (known and unknown) attacks that can be found, and runtime performance.

To this end, *theorem provers* have been used to derive proofs from logical statements that model protocols and adversaries. This way it is possible to show that protocols fulfill certain properties, for instance as that an intruder never learns a secret message, or a private key. Of historical interest are early proposals to mechanize the analysis of security protocols, such as Interrogator [224] and the NRL protocol Analyzer [218]. Both are GUI-driven toolkits, written in Prolog [305], that apply (incomplete) heuristics or user interactions to explore the state space of protocols definitions and try to find suitable paths in the message sequence that represent attack strategies.

Paulson [249] proposes an approach to protocol analysis, where protocols are inductively defined sets of traces on which the correctness of security properties can be proven by induction. Thus the security guarantees can be established for all possible traces. The approach uses the Isabelle/HOL [238] theorem prover, but requires significant human expertise and interactions ("as little as a week or two") for concrete analyzes.

Blanchet has developed ProVerif [45, 46] a fully automatic tool that uses the deduction mechanism of Prolog to prove secrecy properties of protocols. Protocols are expressed from the attacker's perspective as Horn-clauses, which are disjunctions of literals with at most one unnegated literal. Optimizations, such as unification of non-attacker rules leads to an efficient and terminating solving algorithm. Verification is achieved by querying a fact that represents an attack; if this can be derived through a trace of the protocol, there is an attack; else the corresponding security property holds. ProVerif has been extended for several years and now provides support for different cryptographic primitives, including shared- and public-key cryptography and hash functions as well as several security properties. Through approximations, it can handle an unbounded number of sessions of even parallel protocols and an unbounded message space, albeit with the possibility of finding false positives (attacks).

Another competing method of proving the correctness of a system is *model checking* [70]. Model checking is used to formally prove or disprove that certain properties hold for a given model. A model checker is a tool that takes a finite model and a specification (set of properties) as input and returns `true` if and only if the model satisfies the specification. If the model does not satisfy the specification the model checker returns `false`. Most model checkers are able to give a counterexample showing why the model does not satisfy the specification. In the case of a security protocol, the model should define all possible, intended and unintended traces and the counterexample a flow of messages that constitute a possible attack. However, a generic Dolev-Yao attacker can create many complex, even encrypted messages in any order. This can easily let the number of possible states grow too large for a full coverage. This challenge has been taken up in an active research field, with several proposals on how security verification can be achieved for practically sized protocols.

Lowe's early use of the FDR model checker in his discussion of the Needham-Schroeder protocol [200] inspired further research [219] both on applying existing model checkers and designing specialized algorithms and tools. Mitchell et al. apply the Murφ [226] model checker to explore the state space of cryptographic protocols. The Casper tool [201] translates high-level, Alice and Bob protocol specifications into FDR. Zhang et al. [374] model check the basic Needham-Schroeder public-key authentication protocol in the SMV tool. In these early experiments, even with rather small protocols, the size of possible state combinations has a tendency towards explosion, restricting the size and number of parallel runs of the protocols that can be inspected.

To overcome this problem, the specifics of cryptographic protocols need to be considered for optimization purposes. Song [302] applies a strand-space model [110]: A strand is a sequence of events and represents the sequence of actions by either a legitimate party or by an attacker. A strand space is a collection of strands, where causal interactions (e.g. sending/receiving of messages) generate a graph structure. Correctness may be expressed by the connections between different strands. Strand spaces thus contain the causal relations, which helps reduce the search space for the tool.

Basin et al. [37] propose the On-The-Fly Model-Checker (OFMC) symbolic model checker that combines the lazy data types of Haskell with symbolic techniques to reduce the search space for attacks. OFMC is one of the back-ends of the AVISPA [22] platform. Recently, extended versions of OFMC have been renamed Open-Source Fixed-Point Model Checker [228] for the AVANTSSAR project [27]. Another AVISPA tool, SATMC [23] decides the security of a model against specifications by converting it through multi-set rewriting, linearizion and various logical optimizations into a finite satisfiability problem which is then fed into a SAT solver. Other modern tools that are able to perform unbounded verification are Scyther [81, 82] and Tamarind [289].

While there are differences in performance [83], these specialized tools for the analysis, falsification and verification of security protocols are not restricted to only small and simple protocols. Even complex protocols in parallel and modern production-grade authentication mechanisms can be studied. While graphical user interfaces are available, these tools still represent academic proof-of-concepts and expertise is needed in the specification of the models and the security goals.

Unfortunately, even these specialized protocol tools do not offer convenient support for the cryptographic mechanisms encountered in Trusted Computing scenarios, such as models for complex TPM instructions or distinction of trusted and untrusted platform states. We will revisit this problem, from the more general view of the analysis of (stateful) security APIs, in Section 5.7.2.

2.7 Vulnerabilities of Trusted Platforms

As security technology with potential high societal impact, Trusted Computing has been an attractive goal for studying vulnerabilities in the academic community.In this section we briefly outline a selection of relevant attacks from the literature, but for this thesis we restrict our scope on identifying rather general security boundaries.

Regarding software-based or logical attacks, a small number of weaknesses in the TPM API have been discovered by formal analysis. Gürgens et al. [135] show how the authentication protocols of the TPM can be confused, and manipulate the key certification mechanism to certify a different key than intended. Chen and Ryan [65] show a risk when using weak authentication secrets. An attacker can then guess the authentication secret off-line, thus bypassing the TPMs time-out based defenses against on-line dictionary attacks. In a follow-up attack [66], the authors show that if several users share the authentication secret to a TPM storage key[8], the storage capabilities (with regard to keys created after the attack) of the TPM can be faked. An improved authentication protocol is proposed as remedy. Bruschi et al. [53] claim a man-in-the-middle attack that keeps a session with the TPM open, which the user believes had been terminated. Lin [193] points out three potential weaknesses in the TPM's API specification, including one which could lead to an attack if the ambiguities in the specification documents were interpreted clumsily by a TPM implementor. Ables [3] showed that the attack fails on real TPM chips.

In 2005, the SHA-1 hash algorithm [358] has been shown not to be as collision-resilient as intended; while there is no direct risk to compromise the TPM mechanisms [129], this discovery has driven forward the design of generation 2.0 TPMs to feature more and more agile cryptographic algorithms. Also for Intel TXT, there have been flaws in earlier versions of the System Management Mode (SMM) software [365] and the SINIT software modules [366] (cf. Section 5.7.1) that implement the late launch mechanism that can be exploited to circumvent the protection of the hypervisor; the *known* critical security flaws in Intel TXT software components have been fixed.

[8]This is common practice for the SRK.

Hardware attacks are even more dangerous. While some TPM vendors base their implementation on high-end security microcontrollers that offer a high degree of protection against even well-equipped physical attacks, several main contributors to the standard intended the TPM to prevent only software-based attacks, especially when attempted remotely over the Internet [132, 133]. Grawrock, as original author of the TPM specifications assumes only resilience against very simple hardware attacks[9] [133].

It is therefore important to experimentally determine which kind of vulnerabilities can be expected. Winter and Dietrich [363] discuss a number of successful attacks, which we only briefly summarize here: Kauer [174] and Sparks [303] have independently shown an attack, where the TPM is reset by a short circuit[10] of the Low Pin Count (LPC) bus [159], which connects the TPM to the chip-set on PC main boards. The main CPU will continue uninterrupted and malicious software can then measure a fresh, fake state into the TPM. The LPC bus can also be eavesdropped [133, 186], which allows the attacker to read sensitive data, such as the results of the unseal operation. Winter demonstrates more passive and active attacks on the LPC and I^2C buses [363] that can also break the chain-of-trust on TXT equipped platforms.

These simple hardware attacks are however not able to uncover private key material and internal data from within the TPM chip. Some selected TPM implementations have been based on resilient Smart Card architectures and certified according to Common Criteria EAL 4+ and provide a robust security against all but the most sophisticated hardware attacks. However, those attacks are possible with sufficient effort. Such a successful attack has achieved exactly that [315] despite the TPM being implemented on a high-end security controller. Still, this attack is very resource (focused ion beam microscope) and time (6 months) intensive.

In summary, there are logical, i.e. remotely exploitable, flaws in the TPM which, while interesting, do not appear to break the core security features. TPM 1.2 and TXT-based systems seem to offer resilience against software attacks and the known vulnerabilities do not appear to be a major threat to the overall concept. The platforms' robustness against hardware attacks is small, and the specific risks depend on what mechanisms the attacker is targeting and how much resources she is willing to invest. In practice, most Trusted Computing mechanisms should be considered vulnerable to a physically present attacker. Against those, additional physical and organizational protection is needed.

2.8 Summary

After decades of progress, the security of computers is far from perfect, yet they need to be trusted in daily live. Trusted Computing improves on the current situation by taking an unconventional approach; highly-secure hardware features

[9]A simple hardware attack requires only an attacker with physical access who is able to open the system's casing and who is equipped with 20 USD worth of hardware [133].

[10]The attack can be done with just some wire as tool.

are added to otherwise unprotected general purpose machines. This allows the mapping of a certain level of security onto selected software components. The Trusted Platform Module is the most important component in this this respect, but it relies on a TCG Software Stack to operate it. To support applications, complex system architectures are needed, which have slowly matured through several generations. It is now possibly to enforce a software's integrity at boot time and provide isolation at run-time. While a single host can be booted to a trusted configuration, it still needs to use risky Internet communications. Cryptographic protocols serve this purpose, but are difficult to design and prone for security failures. Rigorous analysis is warranted, but not easy to do correctly either. The field of formal methods offers several promising approaches to automate the detection of weaknesses in protocols. Yet challenges remain, such as the consideration of Trusted Computing components that interact in protocols. Overall, Trusted Platforms offer effective improvements to security, but remain vulnerable to hardware attacks.

3 Design and Standardization of a Trusted Computing API

3.1 Introduction

In the Trusted Computing approach, security is bootstrapped from a small dedicated piece of secure hardware, the Trusted Platform Module (TPM). Most of the major computer manufacturers ship servers, desktop and notebook computers containing TPM's and several hundreds of million of machines provide this hardware device [296]. Despite this widespread penetration, however, operating system and application support for Trusted Computing remains limited. Major obstacles to the development of Trusted Computing enabled software have been the high complexity of the specification of the software stack that is used to manage the TPM and limited support for programming languages other than C [291, 308].

Declarations
This chapter extensively adapts, cites and reuses previously published material from the author, especially

[327] R. Toegl, P. Lipp, J. Nisewanger, D. D. Rao, T. Winkler, W. Keil, T. Hong, M. Nauman, B. Gungoren, and K. M. Graf. JSR 321 Trusted Computing API for Java. Java Community Process Specification Final Release `http://jcp.org/en/jsr/detail?id=321`, 12 2011. Java Specification Request # 321. Website accessed October 31, 2012.

[339] R. Toegl, T. Winkler, M. Nauman, and T. W. Hong. Specification and Standardization of a Java Trusted Computing API. *Softw. Pract. Exper.*, 42(8):945–965, 2012.

[338] R. Toegl, T. Winkler, M. Nauman, and T. Hong. Towards platform-independent trusted computing. In *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing*, pages 61–66, Chicago, Illinois, USA, 2009. ACM.

[330] 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*, pages 60–67. Vieweg+Teubner, 2009.

[262] 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, pages 45–53. Springer Berlin Heidelberg, 2011.

The API was designed together with the JSR 321 Expert Group, assembled, lead and moderated by the author. Parts of the Technology Compatibility Kit were done by Johannes Zlattinger in his bachelor project [377] supervised by the author. The Cloud computing case study was implemented by Siegfried Podesser in his Master's thesis [261], which was supervised by the author. The Technology Compatibility Kit and the Reference Implementation was created by the author together with the summer interns Michael Gissing, Siegfried Podesser, Josef Sabongui, and Robert Stoegbuchner.

In particular, there is insufficient support for platform-independent runtime environments like .NET [222], Android or Java. Such environments are particularly useful for implementing modern distributed computer systems, which require deployment of code and data across heterogeneous environments [125]. For instance, several billions of devices support Java, and Oracle claims [243] that the Java developer community, with nine million members, is the largest of its kind. It is of little surprise that there have been a number of attempts to provide TPM libraries that target the Java programming language. However, none of them have yet been established as a generally-accepted standard Application Programming Interface (API) for TPM access.

We now describe the design of a *high-level* Java API for Trusted Computing, which has been published as an official Java *standard* in [327]. Our goal in designing this API is to provide a simpler, high-level interface to the TPM while still adhering to the concepts and standards defined by the Trusted Computing Group. The creation of this *Java Specification Request No. 321* (JSR 321) has been an open, transparent process.

This chapter describes an approach to integrate Trusted Computing functionalities in the Java environment. Starting with a motivation of the research presented we will move on to discuss others' proposals for Trusted Computing APIs and our conclusions on those. We will then present our approach which goes considerably further and endeavors standardization in the Java Community Process which develops the Java platform. This chapter presents design choices, reports on implementations and testing and discusses related work. The result of this endeavor is the official industry standard JSR 321, which was released in 2012.

3.2 Trusted Computing in the Java Environment

3.2.1 Emerging Fields of Use

At the application layer, the Java programming environment has seen broad adoption ranging from large-scale business applications hosted in dedicated data centers to resource constrained environments found in mobile phones or Personal Digital Assistants (PDAs). By default, Java program code [131] is not compiled to native machine code but to a special form of intermediate code, called byte code. This byte code is then executed[1] by the *Java Virtual Machine* (JVM) [194]. This characteristic makes Java an excellent choice for development aiming at heterogeneous environments. In contrast to conventional programming languages such as C or C++, Java is equipped with inherent security features supporting the development of more secure software. Among those features are automatic array-bounds checking, garbage collection and access control mechanisms. Additional aspects that distinguish Java from other environments are code-signing mechanisms and the verification of byte code when it is loaded. The class-loading mechanism separates privileged code and creates a sandbox for remotely fetched classes [130].

Over time, Java has become one of the major development environments for business applications, especially in fields that highly depend on the security and

[1]In the JVM, bytecode is either interpreted, or "just-in-time" compiled to native code on demand.

trustworthiness of computer systems, e.g., financial service providers. Such commercial business environments are among the fields where Trusted Computing technologies is very attractive. Another area of application is network-based software, where Java is a logical choice for highly distributed applications that are deployed in heterogeneous environments. Java is also the programming language used in the Google Android environment which is especially popular on mobile phones and tablet computers. Here also, Trusted Computing is very promising to further improve security. While generic cryptography is well supported by the Java Security Architecture, there is currently no established standard API for Trusted Computing available.

Still, a large number of Java-based use cases have been demonstrated for Trusted Computing, using several existing approaches for Trusted Computing integration in Java.

3.2.2 Review of Existing Java Libraries

This section presents an overview of existing libraries and APIs that provide first, experimental support for Trusted Computing to Java developers. Additionally, we discuss strengths and weaknesses of the individual approaches.

Trusted Computing for the Java Platform and jTSS

In the Open_TC project [239], a team from IAIK has developed a number of Trusted Computing components for Java and have since made the results available in the open source "Trusted Computing for the Java Platform" project. The central component is an implementation of the Trusted Software Stack (TSS) for Java programs called jTSS [258]. It is a large library that provides Java programs with the TSS functionality that C programs currently enjoy.

Overall, the project offers two flavors of TSS implementations. See Figures 2.2 (p. 22) for an overview of the TSS layers and Figure 3.11 (p. 64) for an overview of how the individual layers can be combined. We will now discuss both alternative implementations, the jTSS Wrapper and jTSS.

jTSS Wrapper provides Java programs access to C-based stacks through an object-oriented API, which forwards calls to the native TSS. A thin C-back-end integrates the TCG Service Provider (TSP) system library. The Java Native Interface (JNI) [242] is a mechanism of the JVM to link into C-libraries. For jTSS Wrapper, JNI maps the constants and functions of the C-based TSP into a Java front-end. There, several aspects of the underlying library, such as memory management, the conversion of error codes to exceptions and data-type abstractions, are implemented. All other operations are then performed by a native TSS installed in the system. Unfortunately, this wrapping approach results in complex component interactions. Debugging across language barriers is a challenging task and thus increases the efforts needed to implement TC applications. In our experience, another major drawback is that implementation errors in the

C-based components seriously affect JVM stability. Also, different native TSSes behave slightly differently, making integration in available system services more difficult and rendering jTSS Wrapper rather arduous to maintain and use.

jTSS is a native implementation of the TCG Software Stack written completely in the Java language. It offers seamless support for Linux operating systems and all Windows versions later than Vista, demonstrating platform independence. The Java TCS also synchronizes access from multiple Java applications. Such a full Java TSS implementation clearly reduces the number of involved components and dependencies. Consequently, this approach results in fewer side-effects as it does not rely on third parties' possibly incompatible TSS implementations or their different interpretations of the TSS specification. Moreover, a pure Java stack can easily be ported to other operating systems and platforms.

The API exposed by both variants is the same, enabling Java application programmers to switch between the two seamlessly, with the choice of the back-end implementation depending on the surrounding platform. The API definition covers data types, exceptions and abstract methods — we refer to it as the *jTSS API* [340]. The jTSS API closely follows the original TSS C interface, permitting the user to stay close to the originally-intended command flows and providing the complete feature set of the underlying library. jTSS covers almost all of the functions specified by the TCG for communicating with the TPM at the fine granularity of TSS commands. As with every TSS, a complex sequence of commands is required to achieve functionality such as sealing, binding, attestation and key generation for application software. The project also provides `jTpmTools` (jTT), which is a sample implementation of the high-level programs that can be written using the jTSS API.

Of both libraries the first release was authored by Thomas Winkler, while the subsequent releases, maintenance and support activities have been done by the author. jTSS has since become a popular choice for Trusted Computing related research activities [10, 12, 33, 35, 50, 51, 61, 80, 90, 109, 126, 142, 148, 169, 171, 178, 180, 202, 203, 247, 248, 255, 288, 307, 314, 320, 326, 330, 333, 367–369, 376]. It is one of the most widely used, supported and regularly updated TSSes available today.

While jTSS does allow programmers the use of the TPM from Java, it still involves a significant learning curve for the average Java programmer, who may not be familiar with the procedural programming style that stems from the C-based TSS legacy. Overall, it is still a complicated API that requires a large amount of training and cross-language experience before it can be used in real-world projects.

TPM/J

Sarmenta et al. present TPM/J [282, 283], a high-level API that allows Java applications to communicate with the TPM. It is compatible with the Linux, Windows XP and Windows Vista and Mac OS X[2] operating systems thus living up to the promise of platform independence.

For the Java language, following the TSS specifications might not be ideal. Accordingly, TPM/J intentionally deviates from the design of TCG. From the point of Java developers this is an advantage, since the TSS specifications provide details specific to the structural programming paradigm and cannot be ported elegantly to the object-oriented perspective. A drawback of TPM/J is that the library does not feature a layered architecture. Therefore, the JVM must run with elevated privileges to access the TPM hardware resource. Moreover, a major concern for users of TPM/J is that it is not regularly maintained, thus making it unsuitable for large-scale adoption in the community. It has however been used to study monotonic counters [283], and an attack on the TPM [3].

TPM4JAVA

TPM4JAVA [144] is a Java library that provides an easy-to-use API to Java programmers for communicating with the TPM. Its design is based on three levels of abstractions:

1. High-level: It provides developers with conveniently usable functionality to execute selected commands such as taking ownership, computing hashes and generate random numbers.
2. Low-level: This is a less user-friendly approach that allows programmers to execute any of the commands supported by the TPM.
3. Back-end: This layer is used internally for communicating with the TPM device driver library.

While the high-level API makes several functions easily accessible, some operations, such as performing a quote during attestation, require several lines of code and a low-level understanding of the actual functioning of the TPM. This makes 'high-level' a misnomer and breaks the consistency when using the API. The project has not been maintained for several years. Finally, TPM4JAVA shares the limitation of the other approaches of not adhering to the TCG's specifications.

3.2.3 Other Trusted Computing Interfaces

In the process that has been employed to conceive the original TSS specification by the TCG, a working group devised a set of APIs to form an industry specification. TSS not only covers a user-oriented API (the TSP Interface), but also architectural and internal details clearly intended for developers who plan to build a full TSS. Still, the actual functionality is not elaborated in detail; in particular, the relationship between different commands in different layers

[2] At the time of writing, the inclusion of TPMs in Mac OS X compatible platforms has been discontinued.

(TSP, TCS, TDDL, and TPM) is neither sufficiently documented nor obvious by naming conventions. Unfortunately, implementors are not required to obey the complete specifications and no reference implementation is offered as guideline by the TCG. As a result, to the best knowledge of the author, no currently available implementation covers the complete specification. Indeed, several of the highly complex functions specified have not been successfully implemented nor tested in the years since the TSS standards were released. There are no test suites, or compliance tests supplied for the software components.

From a developer's point of view, the TSS design suffers from several drawbacks. It is challenging to develop applications with it, as even straightforward mechanisms of the TPM correspond to complicated instruction flows in the TSS API. There is little documentation, and the API's nomenclature is neither intuitive, nor well assisted by code completion IDE tools. Also, a lot of functionality specified in the API is not relevant for many typical use cases of Trusted Computing. This is especially true for heterogeneous environments or embedded platforms. Based on these insights, which we first brought forward in 2009 [338], several other proposals for higher level interfaces have recently been made for non-Java environments.

Stüble and Zaerin [308] propose a simplified trusted software stack (µTSS) for the C++ language. It mimics the TCG layered architecture (in the form of object-oriented oTDDL, oTCS and oTSP layers), with oTSP providing high level abstractions of selected functionality. It is noteworthy that oTCS offers access to all TPM instructions.

Also for C++, Cabbidu et al. [55] present the Trusted Platform Agent, a library that aggregates TSS functions into a higher-level API and also integrates other features missing in the language's standard library like cryptography and network communications. It therefore provides selected building blocks for trusted applications that can be applied with a low learning curve.

Reiter et al. [270] describe an alternative stack design that integrates in an open source cryptography API for Microsoft .NET.

Beneath the TCS layer of the TSS, the TPM Base Services (TBS) [223] in Windows Vista or later, virtualize the TPM for concurrent access and also offer a small set of management features to scripting languages.

However, each of these proposals needs to focus, based on the programming language, on a separate developer clientèle, different fields of application and other norms and conventions than we do in JSR 321.

3.2.4 Findings

While the aforementioned APIs all share the common goal of providing Trusted Computing functionality to developers, to date none of them has seen widespread adoption beyond research and academia.

One of the main reasons is that the interfaces exposed by the libraries often are difficult to learn and understand. This stems from several facts:

1. Trusted Computing by itself is a complex technology. The specifications

defining the two major components — the TPM and the TSS [342, 345] — together consist of about 1500 pages. The concepts are often not well presented for novice users and details have to be looked up in several different places. So it comes as little surprise that very few actual software products make use of the original C-based TSS to access the TPM. Indeed, a 2008 study [291] on the TSS concludes that, *"it is apparent that, until now, no application exists that makes use of this technology. Even the simplest applications [...] have not been applied yet."*

2. Implementations like jTSS try to mimic the interface defined by the TSS specification. This interface, however, was developed for procedural programming languages like C. Even though jTSS tries to map the TSS concepts to an object oriented API, it still does not fit well into the Java ecosystem and feels unnatural to developers familiar with other Java APIs. Furthermore, large and complex amounts of code are required to set up and perform basic Trusted Computing functions. This stems from the fact that in the original C-based TSS API, functions take long lists of parameters with many potentially illegal combinations. This makes the API error prone and complex to use for developers without detailed Trusted Computing knowledge.

3. Implementations like TPM/J and TPM4JAVA provide alternative interfaces to Trusted Computing functionality. While in the first case, the interface is at a very low level, the second one offers some higher level abstraction, but is neither consistent with Java or TCG conventions nor functionally complete.

4. A full-fledged TSS is a very flexible and powerful library, but practical experience has shown that its full capabilities are not actually required for the vast majority of typical Trusted Computing applications.

   From our experience of maintaining jTSS and supporting its users stems the insight that most adopters only follow existing code examples and test code. Few experiments create fresh functionality, and if so tend to follow a steep and tedious learning curve. Not only users are affected, but also developers of the TSS as such. The specifications have too much leeway, causing incompatibilities and implementations of the various TSS layers themselves are often difficult to maintain.

Therefore, a novel design is needed that improves on the identified shortcomings and provides a programming interface suitable for Java developers while considering the specifics of trusted hardware platforms, legacy software architectures, and, obviously, the Java environment.

3.3 API Design

We can now move on to describe the major influences on our specification and our resulting design decisions. Based on defined goals (Section 3.3.2) and clear assumptions on the developers (Section 3.3.3) that we target, we consider the restrictions imposed by the surrounding environment and discuss how the standardization process has influenced our proposal. From these constrains, we have implemented an agile specification process which enables us to derive the design of the JSR 321 API.

3.3.1 The Java Community Process

We now introduce the design and standardization process for the Trusted Computing API for Java, known as *Java Specification Request #321 (JSR 321)* within the Java Community Process (JCP) [170]. The Java Community Process aims to produce specifications using an inclusive, consensus-based approach. It is controlled by an elected *Executive Committee* (EC), which represents most major players in the Java industry. The central element of the JCP is to gather a group of industry experts who have a deep understanding of the technology in question and then have a technical lead work with that group to create a first draft. Consensus on the form and content of the draft is then built using an iterative review process that allows an ever-widening audience to review and comment on the document. While the JCP provides a formal framework with different phases and deliverables, an *Expert Group* (EG) may freely decide on its working style.

There are a number of phases in the process. First, a new specification is *initiated* by a community member and approved for development by the (appropriate[3]) EC. Then, a group of experts is formed to develop a preliminary draft of the specification. Feedback from *early reviews* is used to revise and refine the draft. Once considered complete, the draft goes out again for *public review*. Now, the EC decides whether the draft should proceed. If approved by the EC, a proposed final draft of the specification is published and the leader of the expert group sees that the reference implementation and its associated technology compatibility kit are completed. Then the EC decides on its *final approval*. Completed specifications are *maintained* and updated.

The process also requires a *Reference Implementation* (RI). Its purpose is to show that the specified API can be implemented and is indeed viable. With the *Technology Compatibility Kit* (TCK) a suite of tests, tools, and documentation that is used to test implementations for compliance with the specification has to be provided as well. This enables third parties to build their own, compatible implementations. The TCK must achieve 100% coverage over all the API's method signatures, i.e., the combinations of each method name and all its parameter lists.

3.3.2 Goals for a Novel API

As outlined in Section 3.2.4, the different existing Trusted Computing libraries do not fulfill all the desirable features of a standard specification for applied

[3]In the period of time when JSR 321 was standardized, there were two ECs. An EC for Java Standard Edition and an EC for Java Micro Edition, which have since merged.

usage of Trusted Computing. We therefore propose a new API and, in this section, present a set of goals the Expert Group has decided on.

Integration with Existing Trusted Computing Platforms. To the OS, the JVM appears just as an ordinary user process. Therefore, the TPM access mechanisms need to integrate with the surrounding environment, be it virtualized or not, and management services. The access should be possible with suitable privileges and not block other services and applications that might contest for the TPM resource.

User-Centric Design. An application *programming* interface is directed towards the programmer. A Trusted Computing API should therefore be designed to aid developers in writing security applications and the design should strive to be easy to use for as many developers as possible. However, TC is not a trivial technology and cannot be used naively; we believe that some appropriate knowledge can be expected of the developers. We will define this in the next Section 3.3.3.

Simplified Interface. We believe that, to make the new API fit into the Java ecosystem, a completely new and fully object-oriented interface is needed. For instance, generic entities (e.g., cryptographic keys, policy objects, etc.) in the TSS should be replaced with specific classes that represent the different types (e.g., a dedicated class for each type of key). This allows the set of offered operations to be limited to those actually applicable for a certain object type, thus enhancing usability and reducing the risk of errors.

Reduced Overhead. The TSS API requires a substantial amount of boilerplate code for routine tasks, such as key creation, data encryption or password management. The proposed API should attempt to replace these lengthy code fragments with simple calls using sensible default parameters where required.

Conceptual Consistency. Names in the API should be consistent not only within the API but also with the nomenclature used by the TCG and in Trusted Computing literature. If this goal is fulfilled, developers will be able to easily switch from other environments to the proposed API. Still, naming conventions of Java must be adhered to.

Testable and Implementable Specifications. The API design should target a small core set of functionality, based on the essential use cases of Trusted Computing. This restriction in size will allow for complete implementations and functional testing thereof. Also, limiting the functional scope makes it possible for all implementations to cover the full proposed API, a key requisite for true platform independence.

Extendability. The API should allow implementers and vendors to add functionality which is optional or dependent on the capabilities of the surrounding platform. This can be achieved through a modular, object-oriented design.

Standards Compliance. Having an industry-wide standard of accessing the TPM from software is indispensable for widespread use and for enabling code mobility between platforms. In the opinion of the EG, the TSS API is unfit for the Java environment. Thus a novel, independent industry standard is needed and the newly proposed API should be designed also considering the standardization process.

3.3.3 Expected Developer Knowledge

A major goal of the proposed JSR 321 API is to simplify Trusted Computing and make it accessible to a larger group of software developers. To achieve this, it is essential to understand the target audience and their skills before we can move on to create a programming interface for them. In the following we define which skills and knowledge we expect of a developer in order to make full use of the API.

In general, a developer using JSR 321 should be familiar with the cryptographic mechanisms provided in the Java Security Architecture [130, 196]. The concepts of data encryption, decryption and the creation of message digests using hash algorithms should be familiar. The algorithms in particular include SHA-1 and RSA (see Section 2.3.2) as used by current TPM implementations. Moreover, a general understanding of Trusted Computing concepts and the functionality provided by a TPM is required. This at least should include knowledge about the following topics:

TPM Life-cycle. Starting with its manufacture, a TPM goes through a number of different states. A developer must understand this life-cycle, for instance that the TPM is shipped in an unowned state and its owner must explicitly take ownership, activate, and enable it. When the machine containing the TPM reaches its end of life, the TPM may be cleared to ensure that any TPM protected data can no longer be accessed. To avoid data loss, appropriate mechanisms like key backup or migration must be executed beforehand. Also the implications of a transfer of ownership of a platform need to be considered.

TPM Key Management. A TPM supports a range of different key types, including storage, binding and signature keys. The developer is responsible for building and maintaining a consistent hierarchy. For instance, if certain keys are created as non-migratable this may rule out any backup of them.

Root- and Chain-of-Trust. Ideally a consistent chain-of-trust would be established by the operating system as described in Section 2.5. However, today's mainstream platforms fail to do so. Security architects and developers need to take extra care to consider the security level represented by the PCR values.

Trusted Storage. Care must be taken when the binding and especially sealing mechanisms are applied to data or user supplied key material. Again, the problem of backup arises, especially considering state changes which can render sealed data permanently inaccessible[4].

Attestation. A number of different protocols have been proposed to perform attestation to a remote verifier [64, 75, 278, 311]. An API can offer the means to create TPM quotes, but there is no established method and infrastructure for attestation. Therefore, architects and developers intending to leverage attestation will need to understand how to specify and implement appropriate communication protocols and also how to interpret measurement hashes.

3.3.4 API Scope Considerations

JSR 321 aims to be a simplified, compact and user-friendly API that should integrate in the complex ecosystem of today's Trusted Computing infrastructures and be consistent and viable. It it therefore essential to clearly focus the scope of functionality offered by the interface.

A natural starting point to derive a Trusted Computing API is from the complete TSS specifications. However, TSS is a system interface, while JSR 321 (and all Java code) is focused on applications and services. As a consequence, JSR 321 is not planned to and cannot fully replace the TSS in all its tasks. Instead, and as required by the nature of the JVM as a user process, it builds on and extends the TSS services offered by the operating system environment. As we will see in Section 3.5, a TSS can be used to implement the JSR 321 high-level API. Also, significantly different requirements stem from the regulations of the design processes and the targeted developer audiences.

By contrast to the process that lead to the creation of TSS (see. Section 3.2.3), in the standardization process of JSR 321 (see Section 3.3.1) the specification of APIs and functionality must come with implementations and tests. As any Java integration must rely on the TSS-based services of the operating system surrounding the JVM, this imposes natural restrictions to the functional scope of the JSR 321 API: only those parts of the TSS specification which are available and thoroughly tested in existing TSS implementations can be used to implement JSR 321.

Also, JSR 321 provides functionality focused on applications and middleware, rather than providing support for the low level BIOS or OS features of the TPM. This restriction matches the field of use of Java and permits a significant reduction in complexity; JSR 321 does not duplicate elements of the Java Cryptography Architecture, thus fitting into the existing library framework.

Finally, many TSS-specified functions are simply not needed in Java APIs: management of memory and other resources can and should be hidden from application developers; object initialization and destruction are natural features

[4]The chain-of-trust could even change in a way outside of the control of the operator, e.g. by hardware repairs, automatic software updates, random measurements, clearing ownership, etc.

of object-oriented languages; cryptographic primitives like hash functions are already well-supported in the Java Cryptography Extension.

3.3.5 Process Implementation: Transparency and Agility

As discussed in Section 2.2.3, it is important that a Trusted Computing library does not limit the users' right of self-determination on the software executed on a platform. To warrant persistent trust in a standard, is should not cause privacy concerns or even fear of back-doors.

In many cases, only the ability to check the blueprints of a platform will give interested parties the opportunity to evaluate and eventually certify the security level of an architecture. To allow evaluation from the widest possible audience, the release of code under open-source licenses is an important first step. Also, it has long been known that obscuring implementation details does not aid the design of cryptographic systems [176]. We believe that transparency is critical for the general success of Trusted Computing. Going even beyond that, the next step in providing transparent specification is to also perform the design process in a transparent way.

The JSR 321 Expert Group has committed itself to devise and follow a transparent process for developing the specifications. Besides industry representatives, the JSR 321 Expert Group has many members from academia and individual professional Java engineers who contribute their spare time. The EG endorses the principle of open research and open source software and strives to give as much transparency to the public as possible.

The JCP program requires that every Java Specification Request (JSR) be published and reviewed several times as Early Draft, Public Draft, Proposed Final Draft, and Final Release. Still, drafts might be made available more regularly than required. This also aids interaction with the community, as the examination of drafts helps contributors to make well-informed suggestions, recommendations or corrections. Also the incorporation of contributions can be traced better.

On the official web site [327] we have explained the JSR proposal, updated the current status in the process, and offered downloads for each review cycle, including the final release. Besides these obligatory information, more services are offered to the public in the JSR 321 project of the `java.net` community: `http://jsr321.java.net/`. One of the many services offered is a public Subversion (SVN) repository to enable collaboration between the experts as well as anyone who registers (for free) at `java.net`. Everyone can check out a documents from the repository and access the latest version of the source code, independent of the JCP program's stages.

Many industry standards, including TSS [342], follow the classical specify-first, implement-later approach. However, for JSR 321 we chose to implement first and specify later, much in the spirit of the Extreme Programming paradigm [38]. This approach allows us to operate with as much as agility as possible. Agility provides an efficient use of resources, short feedback cycles, and a chance to consider different approaches while still moving forward. This flexibility is

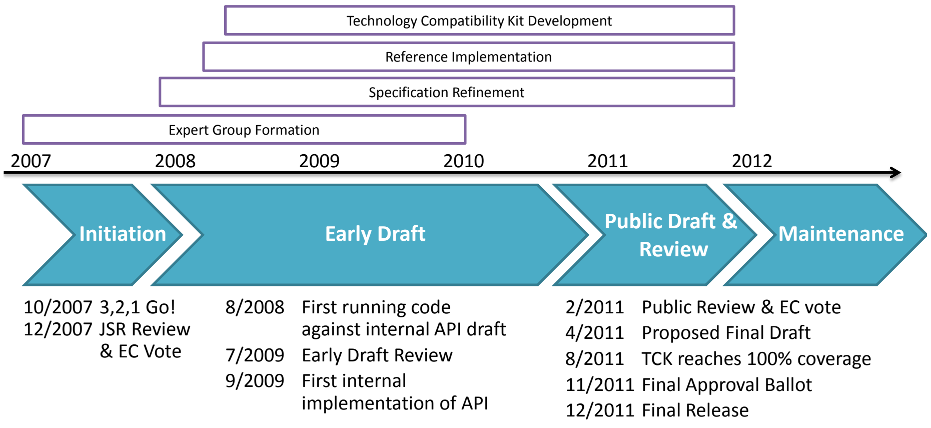

Technology Compatibility Kit Development
Reference Implementation
Specification Refinement
Expert Group Formation
2007
2008
2009
2010
2011
2012
Initiation
Early Draft
Public Draft & Review
Maintenance
10/2007 3,2,1 Go!
12/2007 JSR Review & EC Vote
8/2008 First running code against internal API draft
7/2009 Early Draft Review
9/2009 First internal implementation of API
2/2011 Public Review & EC vote
4/2011 Proposed Final Draft
8/2011 TCK reaches 100% coverage
11/2011 Final Approval Ballot
12/2011 Final Release

Figure 3.1: Development and Specification Time-line for JSR 321.

important as no comprehensive solution to the design challenge existed previously. It also allows the Expert Group to take small, easy-to-reach steps. For instance, we did not start with writing text, but worked with Java code from the beginning. We consider it the perfect medium to develop an API and modern refactoring tools make it possible to develop such drafts into high quality APIs. Then specification documents can be automatically derived from it. One other benefit this approach has for the team is that it builds more enjoyment into the process, as software engineers prefer writing code to describing specifications in prose.

We outline the overall time-line and milestones of the design, implementation and specification in Figure 3.1. Agility ideally supported the design process, as early implementation and several implementation iterations lead directly to the specification documents: After agreeing on the basic flavor of the API, the JSR 321 Expert Group developed a minimal prototype. This allowed us to learn more about the approach and its implementability. When, after several tweaks, we were satisfied with the behavior and usability of the prototype, it was easy to extract Java interface definitions from it. After only slight modifications we were able to release this as an Early Draft. In the next iteration, the interfaces were implemented and test cases created. Feedback, in the form of e-mails and reports from reviewers and implementations experiences lead to improvements of the API that were directly addressed in the Java code or comments in the generated documentation.

While we did not have the resources available for empirical testing of the usability of the API in a formal lab setting with different groups of developers, we did start out with a characterization of users [71] (Section 3.3.3) and the agile process together with its implicit peer reviews [111] has assisted us in creating an easy-to-use API. Test were done at the same time as the implementation, or sometimes created before making the implementation. Operating in this fashion allowed the group to make headway with the API design, implementation, and test kit all at the same time. The different versions of written specifications submitted are therefore snapshots of the API's Java code.

Our efforts for a innovative and smooth standardization were honored in the JCP through the 2010 ”Outstanding Spec Lead“ award.

3.3.6 Selected Features

We have now discussed our considerations on what the API should contain, who we intend to use it.To derive the functional scope of the API, the EG considered the commented complete list of TCG-specified TSP functions [63]. Based on the criteria and principles laid out previously, the EG selected those features that are required for core use cases that have high importance for practical applications. Figures 3.2–3.8 show the resulting mapping and represent the final decisions made by the JSR 321 Expert Group [327]. In the first column the original TSS C-function is named, and elaborated (by [63]) in the second column. Next, a brief assessment of its role is given followed by the design decision on whether to include it in the form of a `public` method in JSR 321. Finally, the Java class that contains this function, visibly or just internally, is given.

In summary, the design focuses on the most important *core concepts of Trusted Computing.* The second main goal is to provide *high usability.* At the same time, the API is designed to remain modular enough to be extensible with future developments.

3.4 Outline of the API

The unique name-space officially assigned to the JSR 321 API is `javax.trustedcomputing`. Within this name-space, a number of packages is specified, each representing a well defined set of functionality. These packages are:

`javax.trustedcomputing.tpm` This package contains all relevant functionality for connecting to a TPM. A TPM connection is represented by the central `TPMContext` object that acts as a factory for other objects specified by the API such as the `KeyManager` or the `Sealer`. The `TPM` interface is also defined in this package, which provides general TPM related information such as its version and manufacturer. Additionally, it allows PCR registers to be read and extended, as well it provides the `Quote` operation required for platform attestation.

`javax.trustedcomputing.tpm.keys` In contrast to the TSS specification, JSR 321 introduces specific interfaces for the individual key types supported by the TPM. This includes interfaces for storage, sealing and binding keys. Compared to having one generic key object, this approach reduces ambiguities in the API and allows the appropriate key usage, as it is hard-coded in the TPM, to be enforced already at the programming interface level. This also enables modern software engineering tools to offer sensible

TSS C-Function	Description	Reason for Removal or Inclusion	Visible in API	Assigned JSR321 Class
Tspi_GetAttribUint32	Find out the value of an integer attribute of an object.	Access to basic information on TSS	No	TPM
Tspi_GetAttribData	Get a non-integer attribute of an object.	Access to basic information on TSS	No	TPM
Tspi_GetPolicyObject	Find out the current authorization policy associated with the context.	Essential for processing commands	Yes	Hidden. Configured using Secret class
Tspi_Context_Close	Close a context.	Context Sessions are essential to TPM	Yes	TPMContext
Tspi_Context_Connect	Connect to a context after it is created.	Context Sessions are essential to TPM	Yes	TPMContext
Tspi_Context_Create	Create a context.	Context Sessions are essential to TPM	Yes	TPMContext
~~Tspi_Context_Free Memory~~	~~Free memory allocated by a Tspi-level function.~~	Java hides Memory Management	-	-
Tspi_Context_Get DefaultPolicy	Use the default authorization policy for the creation of an object.	Essential	No	Hidden. Configured using Secret class
Tspi_Context_Create Object	Create an object, such as a key object. After creating the object, the fields in the object need to be set.	TPM object live in Contexts	Yes	TPMContext
~~Tspi_Context_Close Object~~	~~Destroy an object.~~	Java manages resources	No	-
Tspi_Context_Get Capability	Get the current capabilities of the context.	Configuration of Context	No	TPMContext
Tspi_Context_GetTPM Object	Get the TPM object associated with a context.	Essential	Yes	TPMContext
Tspi_Policy_Flush Secret	Remove the authorization data from memory.	Desirable for security.	Yes	Secret destruction could be difficult in actual implementations (delayed garbage collection)
Tspi_Policy_Assign ToObject	How one assigns a policy to an object—for example, a key.	Essential for processing commands	No	Hidden. Configured using Secret object
Tspi_TPM_GetCapability	Get the set of capabilities of the TPM.	Access to basic information on TPM	No	TPM
Tspi_TPM_SetCapability	Set capabilities of the TPM.	Access to basic information on TPM	No	TPM

Figure 3.2: Mapping of TSS Functions to the JSR 321 API and its Classes (part 1).

Tspi_TPM_GetRandom	Return a random number of the specified size.	Useful feature	Yes	TPM
Tspi_TPM_StirRandom	A means of adding entropy to the internal random number generator. It is a good habit to call it with the current time. (Because it only adds entropy, it can never hurt.)	Useful feature	Yes	TPM
Tspi_Key_GetPubKey	Get the public key of a key pair.	Vital Feature	Yes	TPMKey
Tspi_Hash_Sign	Hashes and signs data with a given key.	Useful feature	No	Signer
Tspi_Hash_VerifySignature	Verifies the signature of given data.	Useful feature	No	RemoteSigner
~~Tspi_Hash_SetHashValue~~	~~Set a particular hash value if you don't happen to want to use SHA-1.~~	Standard feature in JCE	-	-
~~Tspi_Hash_GetHashValue~~	~~Determine the current value of a hash object.~~	Standard feature in JCE	-	-
~~Tspi_Hash_UpdateHashValue~~	~~Add new data into a hash object, which continues the hash in the way defined by the hash algorithm. Currently only SHA-1 is supported.~~	Standard feature in JCE	-	-
Tspi_Data_Unbind	Unbind data by decrypting with a private storage key. This takes place inside the TPM.	Useful feature	Yes	Binder
Tspi_Data_Unseal	Decrypt data sealed to a TPM when PCRs are in a determined state (and optional authorization data is present).	Useful feature	Yes	Sealer
Tspi_PcrComposite_SelectPcrIndex	Select a particular set of PCRs in a PcrComposite object.	Vital Feature	Yes	PCRInfo
Tspi_PcrComposite_SetPcrValue	Set what values the PCRs in a PcrComposite object should have. This is preparation for doing a seal.	Vital Feature	Yes	PCRInfo
Tspi_PcrComposite_GetPcrValue	Returns the current value of a PCR in a PcrComposite object.	Vital Feature	Yes	PCRInfo
~~Tspip_CallbackHMACAuth~~	~~Configurable mechanism for creating an HMAC for authorization data.~~	C-style callback functions are not needed in Java	-	-

Figure 3.3: Mapping of TSS Functions to the JSR 321 API and its Classes (part 2).

~~Tspip_CallbackXorEnc~~	~~Used to provide a means of inserting a secret to a TPM object (such as when doing a change auth) without allowing sniffing software to see what the new authorization is as it goes by.~~	C-style callback functions are not needed in Java	-	-
~~Tspip_CallbackTake Ownership~~	~~Take ownership of a TPM using a callback mechanism.~~	C-style callback functions are not needed in Java	-	-
~~Tspip_CallbackChange AuthAsym~~	~~Use a callback mechanism to change authorization.~~	C-style callback functions are not needed in Java	-	-
Tspi_Data_SealX	Just like Seal, except that it can also use locality and record historical PCR values for PCRs other than the ones it is locking to.	Nice to have	No	-
Tspi_TPM_Quote2	Provide more information (including locality stuff) than Tspi_TPM_Quote does.	Vital Feature	Yes	Attestor
Tspi_PcrComposite_ SetPcrLocality	Set the locality settings for a PcrComposite structure.	Nice to have	No	PCRInfo
Tspi_PcrComposite_ GetPcrLocality	Return the locality settings of a PcrComposite structure.	Nice to have	No	PCRInfo
Tspi_PcrComposite_ GetCompositeHash	Return the Composite hash of the PcrComposite structure.	Vital Feature	No	PCRInfo
Tspi_PcrComposite_ SelectPcrIndexEx	Because the new Pcr_long structure independently sets which PCRs to record historically and which to use for release, this command was needed to set them individually.	Hidden implementation detail	No	PCRInfo
Tspi_TPM_ReadCurrent Counter	Read the value of the current counter.	Nice to have, monotonic counters not supported in OSes	No	-
Tspi_TPM_ReadCurrent Ticks	Read the current tick value (which corresponds loosely to time) of the TPM.	Useful, but TCG specifications on time correlation are ambiguous	No	-
Tspi_Hash_TickStamp Blob	Sign data together with the current tick value and tick nonce.	Useful, but TCG specifications are ambiguous	No	-

Figure 3.4: Mapping of TSS Functions to the JSR 321 API and its Classes (part 3).

~~Tspi_NV_DefineSpace~~	~~Create a section of NVRAM and associates it with specific authorization (such as authorization data, PCR values, locality, or once per power on).~~	NV RAM Access is not needed for applications	-	-
~~Tspi_NV_ReleaseSpace~~	~~Put NVRAM space previously allocated back into the pool.~~	NV RAM Access is not needed for applications	-	-
~~Tspi_NV_WriteValue~~	~~Write a value to the NVRAM space previously allocated.~~	NV RAM Access is not needed for applications	-	-
~~Tspi_NV_ReadValue~~	~~Read a value from NVRAM space previously allocated.~~	NV RAM Access is not needed for applications	-	-
~~Tspi_TPM_DAA_Sign~~	~~Use a DAA credential to verify either a message or an AIK.~~	NV RAM Access is not needed for applications	-	-
~~Tspi_TPM_GetAudit Digest~~	~~Get the current audit digest of the TPM.~~	TPM Implementations do not support Audits	-	-
~~Tspi_TPM_SetOrdinal AuditStatus~~	~~Set an ordinal to be audited.~~	TPM Implementations do not support Audits	-	-
Tspicb_CallbackSealx Mask	Used when masking or unmasking data sent or returned with Data_SealX or Tspi_Data_Unseal operations.	C-style callback functions are not needed in Java	No	Sealer
Tspicb_CollateIdentity	Because it isn't clear what encryption algorithms will be required by a certificate authority, this command can be used to encrypt the collated information with any encryption algorithm.	Optional functionality for AIK Cycle	No	-
Tspicb_Activate Identity	Similarly, when a certificate is encrypted by the certificate authority, the decryption will be done entirely in software, so this command allows any decryption algorithm trusted by the certificate authority to be used.	Optional functionality for AIK Cycle	No	-

Figure 3.5: Mapping of TSS Functions to the JSR 321 API and its Classes (part 4).

~~Tspicb_DAA_Sign~~	~~Extend properties of the DAA protocol.~~	No DAA reference implementations available	-	-
~~Tspicb_DAA_Verify Signature~~	~~Extend the usefulness of the DAA protocol.~~	No DAA reference implementations available	-	-
Tspi_Key_LoadKey	Load a particular key into the TPM.	Vital Feature	No	TPMKey
Tspi_ChangeAuth	Create a new object with a different authorization.	Vital Feature	Yes	TPMKey
Tspi_ChangeAuthAsym	Create a new object with a different authorization (but the same other internal parameters) without revealing knowledge of the new authorization to the parent key.	Implementation Detail	No	-
Tspi_Context_LoadKey Blob	Load an encrypted key blob into the TPM, used when you have the key blob file.	Implementation Detail	No	KeyManager
Tspi_Context_LoadKeyBy UUID	Load a key into the TPM when you know its UUID.	Vital Key Management Feature	Yes	KeyManager
Tspi_Context_ UnregisterKey	Remove a key from a user or system key store.	Vital Key Management Feature	Yes	KeyManager
Tspi_Context_DeleteKey ByUUID	Remove a key from the TPM referenced by UUID.	Vital Key Management Feature	Yes	KeyManager
Tspi_Context_GetKeyBy UUID	Search for a key by its UUID, and returns a handle to it.	Vital Key Management Feature	Yes	KeyManager
Tspi_Context_GetKey ByPublicInfo	Search for a key by its public data and returns a handle to it.	Vital Key Management Feature	Yes	KeyManager
Tspi_Context_Get Registered KeysByUUID	Return a list of all the registered keys in a registry with their UUID.	Vital Key Management Feature	Yes	KeyManager
Tspi_TPM_GetStatus	Find out how bits in the TPM are set.	Basic TPM feature	Yes	TPM
Tspi_TPM_Quote	Uses an ID to sign the PCRs currently in the TPM. A nonce is used to guarantee freshness.	Vital Feature	Yes	Attestor
~~Tspi_Key_Convert MigrationBlob~~	~~Import a migration blob from a migratable key.~~	Migration is optional	No	-
~~Tspi_TPM_CertifySelf Test~~	~~Tells the TPM to use an AIK to certify the self-test results.~~	Not useful for applications	No	-
~~Tspi_TPM_GetTestResult~~	~~Get the self test result, unsigned.~~	Not useful for applications	No	-

Figure 3.6: Mapping of TSS Functions to the JSR 321 API and its Classes (part 5).

`Tspi_SetAttribUint32`	Set an integer attribute of an object.	Implementation Detail	No	-
`Tspi_SetAttribData`	Set a non-integer attribute of an object.	Implementation Detail	No	-
`Tspi_Policy_SetSecret`	How one associates authorization data with a policy, to be used, for example, in creating or using a key.	Key Feature	No	Hidden using `Secret`
`Tspi_TPM_PcrExtend`	Extend a particular PCR.	Vital Feature	Yes	`TPM`
`Tspi_Data_Bind`	Bind data to a TPM by encrypting it with a public storage key. This takes place outside the TPM.	Vital feature	Yes	`RemoteBinder`
`Tspi_Data_Seal`	Encrypt data to a TPM key and PCR values. It can be done only inside the TPM because it also registers historical data as to the PCR values in the TPM when the command is done.	Useful feature	Yes	`Sealer`
`Tspi_Context_Register Key`	Register a key into either a user's key store or a system's key store and returns the UUID.	Vital Key Management Feature	Yes	`KeyManager`
`Tspi_TPM_GetPub EndorsementKey`	Return the public portion of the endorsement key.	Optional functionality for AIK Cycle	No	-
`Tspi_TPM_Collate IdentityRequest`	Gather all the information a certificate authority will need in order to provide a certificate for an AIK.	Optional functionality for AIK Cycle	No	-
`Tspi_TPM_Activate Identity`	Take the encrypted returned data from the certificate authority, and use it to determine the decryption key used to return the certificate for an AIK to the owner.	Optional functionality for AIK Cycle	No	-
~~`Tspi_TPM_SetStatus`~~	~~Set bits in the TPM.~~	Not useful for applications	-	-
~~`Tspi_TPM_SelfTestFull`~~	~~Tells the TPM to execute a full self test.~~	Not useful for applications	-	-
`Tspi_TPM_PcrRead`	Read a particular PCR.	Useful Feature	Yes	`TPM`
`Tspi_Key_CertifyKey`	Create a certificate of a non-migratable key by signing it and its characteristics with an AIK (ID).	Useful Feature	Yes	`TPMKey`

Figure 3.7: Mapping of TSS Functions to the JSR 321 API and its Classes (part 6).

`Tspi_Key_CreateKey`	Create a new RSA key.	Vital Key Management Feature	Yes	`KeyManager`
`Tspi_Key_WrapKey`	Wrap an already extant RSA private key.	Vital Key Management Feature	Yes	`KeyManger`
`Tspi_Key_Create MigrationBlob`	Create a migration blob from a migratable key.	Migration is optional	Yes	-
`Tspi_Key_UnloadKey`	Remove a key in the TPM.	Vital Key Management Feature	Yes	`TPMKey`

Figure 3.8: Mapping of TSS Functions to the JSR 321 API and its Classes (part 7).

options for auto-completion. Using strong key types also relates well to results in formal API design and analysis research.

`javax.trustedcomputing.tpm.structures` This package holds data structures required for certain TPM operations. They include the `PCREvent` structure required for operations on the measurement log, `PCRInfo` used as part of platform attestation and `ValidationData` as returned by the TPM quote operation.

`javax.trustedcomputing.tpm.tools` In this package, there are interface definitions for helpers classes to perform TPM operations such as binding, sealing, signing and time stamping.
The `javax.trustedcomputing.tpm.tools.remote` sub-package offers abstract classes that allow a remote host without TPM to participate in Trusted Computing protocols. It provides the functionality to validate and verify signatures on TC data types.

For error handling, a single `TrustedComputingException` communicates the error codes of all lower layers. It offers the original TPM/TSS error codes, but also a human readable text representation, which is a great step forward in terms of usability. Despite using only a single exception class, implementations of the API should forward as much error information as possible. For illegal inputs to the JSR 321 API, default Java runtime exceptions are used. Finally, functions offering bit-wise access to status and capability flags are replaced by specific Boolean methods that allow developers easy access to application-relevant flags.

In JSR 321, the `KeyManager` interface defines methods for creating new TPM keys. Upon creation, a secret for key usage and an optional secret for key migration have to be passed as parameters. After a key is created, the `KeyManager` allows the key, wrapped by its parent, to be stored in persistent storage. As required, the `KeyManager` allows keys to be reloaded into the TPM, provided that the key chain up to the storage root key has been established (i.e., each parent key is already loaded into the TPM). Every time a new key is created or loaded from permanent storage, a usage secret has to be provided. This secret is represented by an instance of a dedicated class `Secret` that is attached to the

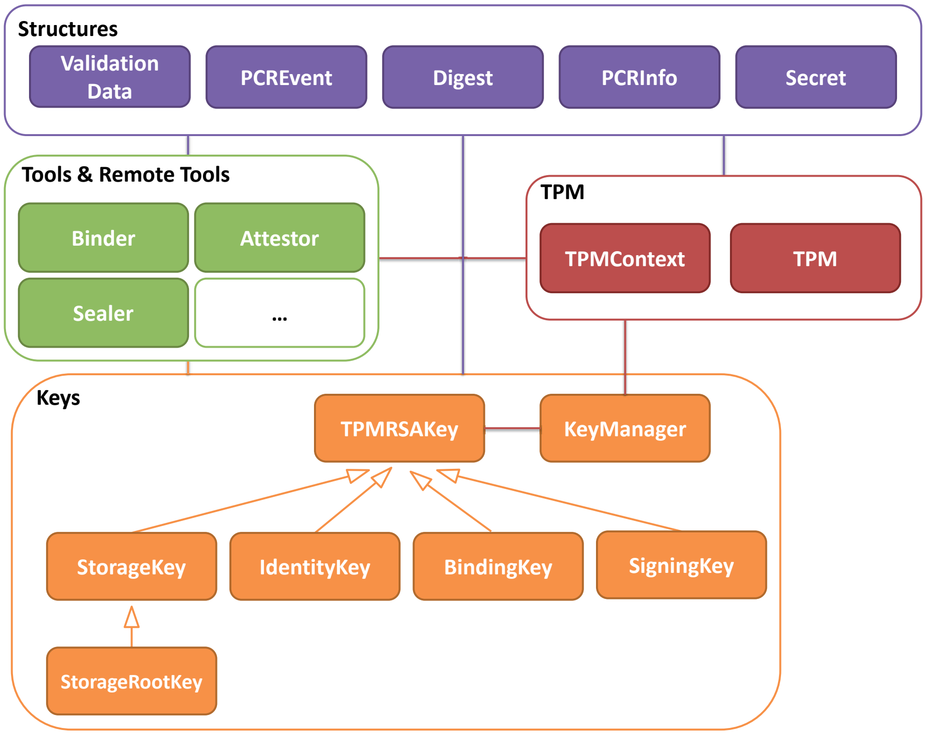


Figure 3.9: Illustration of the Relationship Between the Core Components, including the `TPMContext`, `KeyManager`, and Key Classes and the Tools.

key object upon construction. `Secret` also encapsulates and handles details such as string encoding, which are often a source of incompatibility between different TPM-based applications.

The extend-able `tools` package implements various core concepts of Trusted Computing. As each tool that accesses the TPM is already linked to a `TPMContext` at creation, there are few or no configuration settings required before using the tool. Each tool provides a small group of methods that offer closed functionality. For example, a `Binder` allows the caller to `bind` data under a `BindingKey` and a `Secret`, and returns the encrypted byte array. Usage complexity is minimal as no further parameters need to be configured and the call to `unbind` encrypted data is completely symmetric. Besides the core set of tools (`Signer`, `Binder`, `Sealer`, `Attestor`, `Certifier`, `Signer`), implementers of JSR 321 may add further sets of functionality. An example might be the tool `Initializer` which manages TPM ownership, if the Java library is implemented on an OS without tools for doing so. For a full reference and interactive outline of the API, the reader is referred to the original standard [327] and the JavaDoc contained therein. An uncommented list of classes and methods is provided in Appendix A.

```
try {

  TPMContext context = TPMContext.getInstance();
  context.connect(null);

  KeyManager keyManager = context.getKeyManager();

  StorageRootKey srk = keyManager
        .loadStorageRootKey(Secret.WELL_KNOWN_SECRET);

  Binder binder = context.getBinder();

  Secret keyUsageSecret = context.getSecret("Passphrase for using the
      key.".toCharArray());

  BindingKey bindingKey = keyManager.createBindingKey(srk,
        keyUsageSecret, null, false, true, true, 2048, null);

  byte[] plainData = new String("Data to be encrypted and
      bound.").getBytes();

  byte[] boundData = binder.bind(plainData,
              bindingKey.getPublicKey());

  context.close();

} catch (Exception e) {
  // Handle errors..
}
```

Figure 3.10: Example of JSR 321 Code that Performs Binding of Data.

In Figure 3.10 we list source code that demonstrates the API. The example shows Java code that first opens a TPM context session, creates a non-migratable cryptographic key with the following key policy: the key is a child of the Storage Root Key, its usage authenticated with `keyUsageSecret`; there is no migration secret set as the key is non-migratable; it's volatile, requires authentication, is a 2048 bit RSA key and is not restricted to a PCR configuration. Finally, the program binds data to the platform where the code is executed. This example also allows us to evaluate the expressiveness and complexity of writing code. In [309], Stüble and Zaerin use the number of Lines of Code (LOC) of code examples as measure to compare different Trusted Computing APIs. They compare implementations of the very same binding use case. According to them, achieving the same functionality requires 146 LOC with TSS, 30 with jTSS and 18 using µTSS. The JSR 321 program we present takes only 15 LOC. Besides this obvious reduction of code size, especially when compared to TSS, the naming conventions used throughout the API allow programmers the effective use of code-completion mechanisms found in modern Integrated Development Environments (IDE) such as Eclipse. In many cases, the IDE will automatically suggest a suitable parameter for method calls, thus considerably speeding up the development of Trusted Computing applications. JSR 321 programs are therefore shorter, and faster to write than TSS programs.

3.5 Implementation and Integration Aspects

For the specification of an interface, it should be enough to provide the complete specifications of classes and method signatures. However, experience suggests that implementation might provide valuable feedback to the design. The design of JSR 321 is implementation driven, and so in this section, we discuss three important aspects that have influenced the design of the standard's API: integration with the operating system for TPM access, technologies for a reference implementation and the design of a test suite that supports our design process.

3.5.1 Java Libraries and Services for Trusted Computing

In all major operating systems direct access to the TPM hardware device requires higher privileges. However, the Java Virtual Machine (JVM) appears as just another user mode process to the OS. It is the task of libraries to bridge the gap between the JVM and the specific operating system's Trusted Computing services. Depending on the surrounding environment, such a Java library will have to accommodate different levels of hardware access and handle different management tasks. At the same time it should maintain a high degree of platform independence.

An implementation of JSR 321 requires its support library to handle the following challenges. First, in the complex architectures that are typical in modern Trusted Computing Platforms, the TPM might actually be a virtualization or even only emulation of a hardware chip. Second, while the TPM has only limited resources that need to be managed by a Singleton [118] software component, several applications should be able to access it at the same time. Third, in most OSes, access to the TPM hardware requires root or administrator privileges. JSR 321 applications, or the JVM in general, should not have elevated system privileges. This would be a breach of the general security assumptions for a Java runtime environment.

In the following we outline the layers that may exist between a Trusted Computing-enabled Java Application and the hardware TPM. Figure 3.11 shows a selection of different library architectures that could be realized with the technologies currently available. The intention of the figure is to provide an overview of possible building blocks that can be combined by an implementor of the API. In most cases there are different ways to solve a problem and different available components in a deployment scenario.

For instance, sharing of the TPM capability may occur at different layers of the software stack for Java applications. This creates different requirements for the software that provides TPM access, resulting in different layered architectures. Still, a uniform API should be available at the top layer.

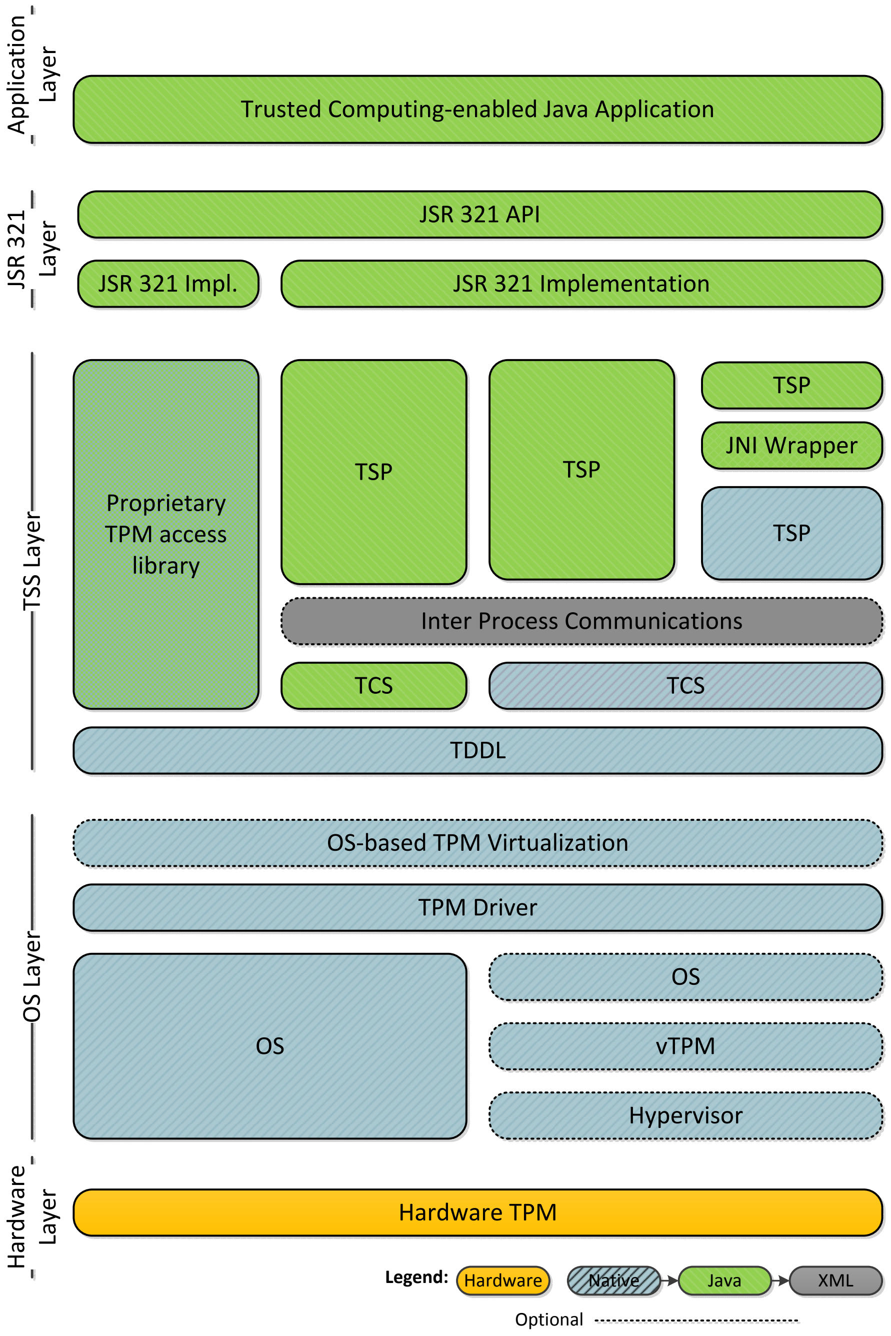


Figure 3.11: Integration Possibilities for the JSR 321 High-Level API in Selected Software Configurations.

In all cases we assume the existence of a hardware TPM. The operating system runs either directly on the hardware platform (left side of the OS layer in Figure 3.11) or within a virtual compartment. With virtualization (right side of the OS layer in Figure 3.11), a single *hypervisor* controls all hardware resources. It could forward the TPM device interface to its guests, but then only one compartment could access the TPM at any given time. Of course, such a limitation should be avoided as it restricts the ability to provide trusted applications. One possible solution is to let the hypervisor provide a separate *virtual TPM (vTPM)* [42, 43, 284, 310] for each compartment. Such a vTPM could be implemented in the hypervisor itself, in one separate compartment for all other compartments, in an extra compartment for each guest compartment and probably in several different ways more.

A similar indirection may occur within the operating system; it is desirable to allow several applications concurrent TPM access within a single instance of an OS. For instance, Windows TBS [223] provide OS-based virtualization by abstracting resource handles and blocking critical TPM commands. To prevent uncoordinated extends by different applications, the current TBS implementation blocks all PCR accesses by default. As an extension, [107] propose TPM *para-virtualization* where the OS is aware of being virtualized. It is then able to manage e.g. PCR access accordingly in cooperation with the hypervisor. Virtual or real, the TPM will then be available via a native driver interface.

In many cases, a managing component such as the TCS of TSS takes control over the TPM and manages its resources. This component also multiplexes accesses by non-privileged applications over some inter-process communication mechanism. Such two-tier libraries, with one layer managing the hardware and another linking to applications, strictly separate the required privileges. For the purpose of a Java library, such a managing component may either be implemented as native code, pure Java components or a combination of both.

Considering these different abstraction layers, and our experience with Java-based TPM-accessing applications, have we identified four suitable pathways to connect the application process to the TPM. These pathways are only a subset of the possible combination of components, but serve here as examples so that we can discuss the several important implications that follow from design choices. Figure 3.11 does not intend to be a visualization of these specific pathways, but may still offer useful guidance to the interested reader.

We discern the following implementation pathways by the way they share access between services running in parallel to the restricted TPM resource.

1. **No multiplex.** The application's JVM process may access and manage the hardware TPM directly and exclusively. As a consequence, all other system and application accesses are blocked. This is sufficient for testing and development purposes, but not suitable for wide deployments.

2. **Library Multiplex.** If exclusive access to the hardware TPM cannot be guaranteed[5], the Java environment needs to be integrated with existing

[5]For instance, if the OS or native applications might access the TPM in parallel.

TPM services, such as a TSS. The TCS will synchronize all TPM accesses.

3. **OS Multiplex.** With OS-based TPM virtualization, all processes are given multiplexed and equal access to the TPM without the need for full TCS functionality. Thus, a Java library may again access its TPM device instance freely.

4. **Hypervisor Multiplex.** In the special case of virtual applications respectively virtual appliances, where the JVM is the only application within a compartment, no other services will interfere. A Java library may then handle its vTPM or assigned hardware TPM exclusively.

Depending on the intended use case, implementers of high-level Java APIs such as JSR 321 have a considerable choice of implementation architectures. We now map these scenarios to the available libraries outlined in Sections 2.4 and 3.2.2.

Scenarios one and four have the most relaxed requirements. No special considerations for shared access need to be taken as no other application will interfere. Exclusive access rights can be granted to the device. Therefore, all library architectures are suitable.

In scenario two, integration with existing system services is needed for non-blocking TPM access, specifically in Linux where there is no OS-based TPM virtualization. Of the reviewed libraries (cf. Section 3.2.2), the *jTSS Wrapper* software package accomplishes this. While a TCS may be implemented as either native code or Java, a hybrid mode with native TCS and Java TSP is currently not available. This is because different TSS implementations are incompatible due to subtle implementation differences stemming from varying interpretations of the TSS specifications.

Scenario three represents the presence of Windows TBS which require Administrator privileges. Here, only split architectures can be used. Currently, only jTSS offers Java interfaces and fully implements split processes with a separate system background service in Windows.

3.5.2 Reference Implementation

Every JSR needs to supply a Reference Implementation (RI) in order to complete standardization and we have released ours under the “GNU General Public License, version 2, with the Class-path Exception” open source license. Open source availability of the reference implementation provides other implementors of the API access to the source code. We expect this to be a valuable guidance to implementors as this provides a precise mapping to the underlying TPM-related command flows and helps understanding the intended functionality of the API at a fine level of detail.

The discussion in Section 3.5.1 of implementation pathways allows us to make a design decision on the general architecture of the RI and to select a suitable implementation technology from the libraries discussed in 3.2.2. We conclude that *jTSS* is well suited to our needs. It is able to manage the TPM on its own, but also integrates with OS-based TPM virtualization. A further advantage is that it implements everything in pure Java, making the software architecture understandable and debuggable for Java developers. This corresponds to the scenarios 1, 3 and 4 discussed in the previous section.

Our RI is based on jTSS and therefore is fully platform-independent. Compared to the other Java libraries discussed, the command flow of jTSS remains intelligible for programmers that are already familiar with any other TSS-like library. The jTSS API is designed to stay very close to the original C-based TSS API and does not modify the logical flow of programs. As a direct consequence there is a high similarity of code structure, and developers can use our JSR 321 RI as a blueprint even if they target a different programming language. We expect that this will reduce the effort for implementations by the industry or open source community.

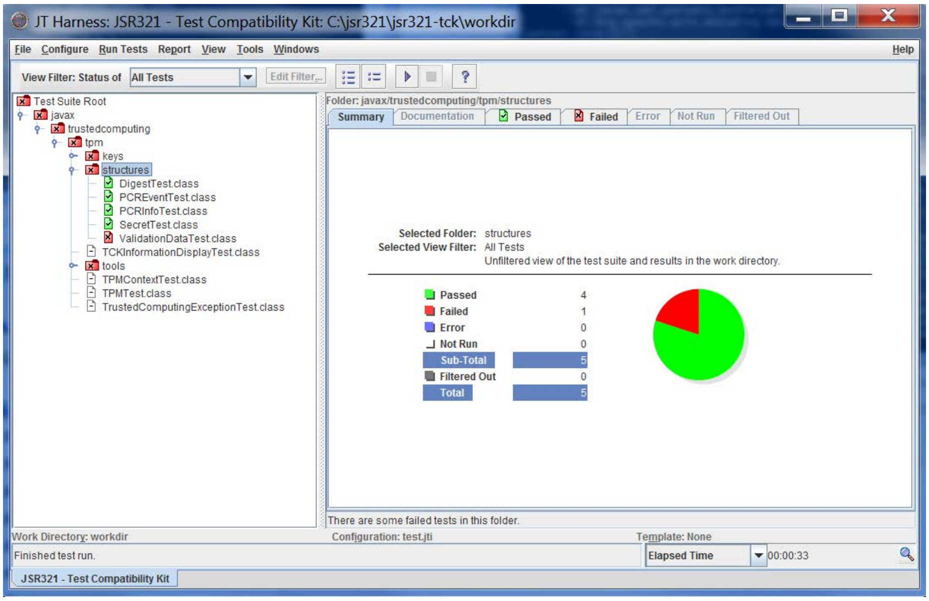
3.5.3 Technology Compatibility Kit

To allow different vendors the creation of independent implementations of the JSR API, a measure to verify whether or not a library meets the given specification is needed. The JCP calls for a Technology Compatibility Kit (TCK) to do this. The motivation for this is that end users should be able to switch between different API implementations transparently, for example because of performance differences or security certifications. A TCK is a set of automatic test cases that should be easy to run by a potential implementer. A JSR implementation needs to pass all of the tests specified by a TCK to be considered compatible with the reference implementation and therefore to meet the API specification of the corresponding JSR.

The importance of the test suite results in stringent requirements. The first one is that there must be a 100% method (signature) coverage, i.e., every public method of the API must be called at least once from within the TCK. The next rule is that it should be simple to run the test suite on supported platforms. The TCK Project Planning and Development Guide [312] contains guidelines on how to create such test suites.

In our TCK, we extend on this and combine a number of tools and technologies. We identify and execute test cases using the JUnit [39] framework. To define dependencies between tests and manage the overall suite we use JT Harness [146]. It provides a GUI to create, manage and execute test suites. A `TestFinder` component collects our JUnit test cases and combines them together with collected meta-data to a suite. After a set of classes containing tests is found by a test finder, it is executed by a `TestScript` which is responsible for running the tests and processing the test results. Additional parameters, such as which Java Runtime Environment to use, can be specified. When the test suite is fully loaded and configured it can be executed with reports being created in different formats like HTML, plain text and XML.

A challenge when writing software tests is to assess the quality of a test suite. A practical way to answer that question is to analyze what code is executed during a test run, i.e., to measure the code coverage of the test suite. We include


JT Harness: JSR321 - Test Compatibility Kit: C:\jsr321\jsr321-tck\workdir
File Configure Run Tests Report View Tools Windows Help
View Filter: Status of All Tests
Edit Filter...
Summary Documentation Passed Failed Error Not Run Filtered Out
JSR321 - Test Compatibility Kit

Test Suite Root
- javax
  - trustedcomputing
    - tpm
      - keys
      - structures
        - DigestTest.class
        - PCREventTest.class
        - PCRInfoTest.class
        - SecretTest.class
        - ValidationDataTest.class
      - TCKInformationDisplayTest.class
      - tools
      - TPMContextTest.class
      - TPMTest.class
      - TrustedComputingExceptionTest.class

Folder: javax/trustedcomputing/tpm/structures

Selected Folder: structures
Selected View Filter: All Tests
Unfiltered view of the test suite and results in the work directory.

🟩 Passed	4
🟥 Failed	1
🟦 Error	0
Not Run	0
Sub-Total	5
■ Filtered Out	0
Total	5

There are some failed tests in this folder.

Work Directory: workdir Configuration: test.jti Template: None

Finished test run. Elapsed Time 00:00:33

Figure 3.12: The TCK Graphical User Interface Provided by JT Harness.

the coverage analysis tool EMMA [273] to transparently measure the test class, method and basic block coverage while executing the test suite. In Table 3.1 we outline the measurement results of the TCK's coverage of our Reference Implementation. The requirement of full method coverage is met. Note that the basic block coverage does not reach 100%, as the API and TCK do not specify all error behaviors or execution branches.

Package	Coverage by		
	Class	Method	Basic Blocks
`javax.trustedcomputing`	100% (1/1)	100% (7/7)	100% (63/63)
`*.tpm`	100% (2/2)	100% (39/39)	73% (692/948)
`*.tpm.keys`	100% (10/10)	100% (40/40)	87% (1496/1715)
`*.tpm.structures`	100% (5/5)	100% (18/18)	92% (250/272)
`*.tpm.tools`	100% (6/6)	100% (31/31)	83% (828/997)
`*.tpm.tools.remote`	100% (4/4)	100% (20/20)	73% (537/735)

Table 3.1: Coverage Results of the JSR 321 TCK.

3.6 Experience

3.6.1 Third Party Implementation and Teaching Experience

Our API design has already been adopted by a third party, indicating the viability of the JSR 321 approach: Atego, as member of the TECOM research project [316] has independently created an implementation based on the Early Draft version of the specification in order to satisfy their need for a high-level Trusted Computing API for Java-based embedded systems. Their implementation was built on top of the previously described µTSS in C++ and Java. The feedback [290] from this external implementation effort has been very positive and helpful. Aside from minor ambiguities in the specification and small feature requests, no major difficulties were reported. TECOM concluded that JSR 321 "provides most functionality that the majority of users would probably need" and that the single interface layer and low level of background knowledge required gave it an advantage over other APIs for their use case of implementing a trusted smart meter.

Together with Winter, Wiegele, and Pirker [364] we demonstrated JSR 321 on an Android platform, with a hardware-protected TPM-emulator. While we refer to Section 4.5.3 for more details on how this can be implemented, we would like to point out that our API is thus compatible with Android-based mobile systems and smartphones.

In summer 2010, JSR 321 was used for teaching the 5th European Trusted Infrastructure Summer School (ETISS), at Royal Holloway, University of London. In a 90-minute 'TPM Lab' we provided an introduction to the central component of Trusted Computing, the Trusted Platform Module (TPM). The lab explained TPM activation control, basic operations, and high-level programming of the TPM with JSR 321. The concept of chain-of-trust was explored in a practical sealing experiment. We pre-configured HP desktop machines with Infineon TPMs for Ubuntu Linux and Eclipse as development environment. Although many of the participants had either no experience with TPMs or with Java, about two thirds of them were able to complete the implementation of an unsealing program within one hour. This clearly underlines the low initial threshold for using the JSR 321 API. JSR 321 was also used in two practicals for courses taught at Graz University of Technology. In the third year bachelor-class "Security Aspects in Software Development" of 2011, students were given the choice of implementing the signature component of a CA service either through JavaCard (using Oracle's API) or the TPM (using JSR 321). The three groups that chose JSR 321 did not require more supervision or advise than those choosing the more conventional technology. In the master-level "Selected Topics IT Security 1" class, in the summer term 2012 students were given the task to demonstrate remote attestation of an Android environment. To this end, they employed a software emulation of the TPM and accessed it through JSR 321. As the Android environment is source compatible with Java, this caused no additional complexity.

In a recent publication, Othman et al. [245] report on a JSR 321-derivate design on Android, without describing potential changes or improvements. The availability of a compatible TPM respectively MTM is assumed by the authors.

Thus, JSR 321 has proven that the specifications are fit for implementation by third parties not involved in the design process and that it is not restricted to jTSS technology as a basis. Also, from our experience, the API can be used in academic teaching much like other existing security technologies.

3.6.2 Case Study: Attestation in the Cloud

In the following brief excursus we will study how JSR 321 can be applied in an important, emerging scenario: the Cloud.

In our age of information, computing has joined the traditional utilities of water, power and communications, thus making computing a tradeable service. Previously named the Grid [114], its trading place is now commonly referred to as *the Cloud* [24, 373].

Typically, Cloud service offers can be characterized as offerings of either *Infrastructure*- (virtual hosts), *Platform*- (networked development frameworks), and *Software*- (domain-specific applications) *-as-a-Service* (IaaS, PaaS, SaaS). The services can be deployed in the *public* Cloud, somewhere on the Internet, in a *private* Cloud, which is physically located in-house at the user, and in *hybrid* installments which contain both public and private nodes. In essence, the Cloud offers scalable computing resources on demand. This is enabled through broadband networks, resource pooling, workload multiplexing, and virtualization on multi-tenancy platforms which are operated in datacenter by third party service providers. Together with automated self-service and billing mechanisms, the economies of scale have lowered the prices for computing considerably, especially when compared to conventional, 'private' data centers. Thus, the Cloud can lift the burden of technically operating a computing infrastructure, by outsourcing it to a third party. Indeed, Cloud computing can be regarded as a matter of "gracefully losing control" [72] over one's programs and data.

An extra challenge of information security then becomes "maintaining accountability even if the operational responsibility falls upon one or more third parties" [72] and many cloud users see[6] loosing control of security sensitive or private data or of processing such confidential digital assets as a dilemma.

A number of concrete security challenges occur in Cloud systems [212]. A report of ENISA [59] lists several security risks, many of those are still not solved. A very promising line of research [5, 41, 79, 177, 182, 183, 197, 204, 208, 277, 281, 287, 294, 299, 353, 357, 373] to overcome the security limitations of distributed computation networks is to incorporate Trusted Computing based on the TPM.

As reported in [262] and implemented in a Master's Thesis [261] by Siegfried Podesser, we have specifically studied the risk that code may not execute as intended. For instance, a remote node might accidentally [304] or even deliberately report wrong answers back. Code distributed could be compromised or, if

[6]Or are required to do so by the law.

```
public final class compute_secLevelC {

 @Gridify(taskClass = GridifyTask_secLevelC.class)
 public static Long compute(long x)
  {
    //compute...
  }
```

Figure 3.13: Example of Code Annotation in Cloud Computing Experiments.

it contains precious intellectual property, stolen. Data need to be secured and handled according to data protection requirements. It is especially important to prove that security facilities, such as role based access control, are actually used and not circumvented by malicious software.

Remote Attestation is a promising approach to convince the Cloud user that such security features are present. In our approach we further desire to include Remote Attestation seamlessly and transparently into the development of distributed applications, while keeping the process as developer friendly as possible.

The proposed architecture is realized in the Java programming environment; we choose GridGain [168] as cloud- and grid-computing framework, Permis [62] as authorization framework and JSR 321 to handle the TPM's Quote functionality. Java Annotations are used to tag individual functions with trusted state descriptors for deployment. GridGain's task is to distribute the workload to different nodes and aggregate the results. We employ remote attestation to determine for each node whether it is in a trustworthy configuration. The integration with Permis allows Cloud operators to define non-trivial policies, which may also cover the geographic location or the network operator.

With regards to the technical realization, JSR 321 offers the functionality for facilitating remote attestation in this Java middleware through the `quote(int[] PCRindices, IdentityKey key, Digest nonce)` method in `javax.trustedcomputing.tpm.tools.Attestor`.

Based on an AIK, created in a proprietary PKI, of the node, the identity of the node can be established. In the prototype described in [261], three exemplary security levels (A,B,C) are defined based on the location and organizational control of nodes. This leads to the distribution of tasks and data to nodes which are at least of the required level. Therefore, developers only need to write a simple Java annotation (as shown in Figure 3.13) for a task to guarantee that a piece of code and its arguments are only deployed on remote nodes which follow the specified policy, even when using a hybrid Cloud topology.

It should be noted that a large amount of complexity is introduced into the overall system architecture by applying TC mechanisms, especially in using the TPM and collecting a meaningful chain-of-trust. These issues can only be solved by changes to the system beneath the middleware-layer; Podesser's prototype implementation therefore leverages the acTvSM [126, 253, 254, 333] approach in much the same way as we will outline later in Section 5.8.2.

These experiments have shown that JSR 321 can provide the remote attestation mechanism in the context of Java-based Cloud software, given a robust chain-of-trust.

3.7 Summary and Outlook

In this chapter we outlined the current status of software libraries for TPM access and application-level integration of Trusted Computing. To date, several commercial and some free implementations of the TCG Software Stack have been published with varying levels of completeness and standard compliance.

As a basis for the work presented in this chapter, we reviewed and discussed the state of the art of available Trusted Computing software libraries. For native applications several native TSS and alternative approaches which intentionally provide a reduced and simplified interface exist.

Java is an important environment for the implementation of Trusted Computing applications. This is emphasized by the existence of several different libraries (cf. Section 3.5.1) and frameworks that have been proposed or prototyped for this language. Our review of existing approaches has uncovered a number of drawbacks including high complexity, inconsistent APIs, limited object-orientation or lack of features.

Despite the availability of libraries and tools, Trusted Computing is not yet widely used and has not found its way into commercial applications [291]. We and other designers of TC APIs [308] attribute this fact primarily to the high complexity of and developer expertise required by existing standards and APIs. We believe that a lower learning curve for the software interfaces can attribute to a more widespread use in the future.

Based on these findings, we have specified goals for a novel high-level Java API that aims to overcome these limitations. Specifically, we have focused on a simple interface for access to commonly used TPM functionality and define the technical knowledge expected of programmers using it. In contrast to the original TSS design, we propose an object-oriented approach that hides low-level details and provides additional guidance for developers by providing solid default configurations. Results from the reference implementations are encouraging and demonstrate the feasibility of the proposed approach. We also describe how implementations of the API can be integrated in virtualized systems and provided with access to the hardware TPM and how such implementations can be tested.

The aim of our API design was the release as the official Java standard API for Trusted Computing. Therefore, we have adopted an agile and transparent working style within the Java Community Process. The desirable set of API features has been selected based on open discussions. Feedback received from external reviewers and independent implementers has helped to adapt and extend the design. The API has been used in teaching and research and successfully applied in embedded, mobile and Cloud scenarios. After two publicly announced reviews and several votes within the Java Community Process, the standard was published [327]. Furthermore, the author has been awarded as “Outstanding

Spec-Lead" in 2010.

We believe that our basic approach of providing a high-level abstraction of core concepts of Trusted Computing will remain valid for future versions of the TPM specification. Any necessary changes to the API, which could become opportune by revisions of the TPM can be supported through updates in the Maintenance phase of the Java Community Process. In addition, we believe that the results of the JSR 321 specification could itself serve to help guide the specification of other Trusted Computing APIs. The author has received encouraging feedback from the TCG that a high-level Trusted Computing API approach as pioneered in the presented work would be highly desirable for future TCG specifications.

We believe that this effort towards an open, simple and consistent programming interface can considerably contribute to the future adoption of Trusted Computing. Even though the proposed JSR 321 API is designed for the Java programming language, we anticipate that the contribution of this work will not be limited to Java. Due to the clear and lightweight design of the API, implementations in other object-oriented programming languages should be possible with only minor adaptations.

4 A Proximity Interface for Attestation

4.1 Introduction

Trusted Computing introduces advanced security mechanisms into terminal hardware, yet there is often no convenient way to help users decide on the trustworthiness of a device. Especially, instant and ubiquitous access to devices such as public terminals raises several security concerns in terms of confidentiality and trust. A direct communication channel between the TPM and the user would appear to be very useful and would be an immediate improvement to the concept of remote attestation.

In this chapter, we propose a TPM-based architecture that includes Near Field Communication (NFC) and allows users to verify the security status of public terminals or kiosk computers. For this, we introduce an autonomic and low-cost NFC-compatible interface to the TPM to create a direct trusted channel. Here, NFC enables users to intuitively establish a communication between local devices. Users can access the TPM with NFC-enabled devices, which have become widely available in the form of smart phones.

This scheme helps users protect against malicious software in public kiosk computers, for instance ATMs or PCs in Internet cafés. While there is no defense against a deliberately malicious kiosk operator who would be able to mount any kind of hardware attack, our scheme is intended to protect against walk-in attackers who try to modify the installed software configuration, or even to confuse platform identities, for instance by manipulation of labels.

Declarations
This chapter extensively adapts, cites and reuses previously published material from the author, especially

[320] R. Toegl. Tagging the turtle: Local attestation for kiosk computing. In J. H. Park, H.-H. Chen, M. Atiquzzaman, C. Lee, T.-H. Kim, and S.-S. Yeo, editors, *Advances in Information Security and Assurance*, volume 5576 of *Lecture Notes in Computer Science*, pages 60–69. Springer Berlin / Heidelberg, 2009.

[326] R. Toegl and M. Hutter. R. Toegl and M. Hutter. An approach to introducing locality in remote attestation using near field communications. *The Journal of Supercomputing*, 55(2):207–227, 2011.

[152] M. Hutter and R. Toegl. A trusted platform module for near field communication. In *Systems and Networks Communications (ICSNC), 2010 Fifth International Conference on*, pages 136–141. IEEE, 2010.

[153] M. Hutter and R. Toegl. Touch‘n’ Trust: An NFC-enabled trusted platform module. *The International Journal on Advances in Security*, 4(1 & 2):131–141, 2011.

[341] R. Toegl, J. Winter, and M. Pirker. A path towards ubiquitous protection of media. In J. Lyle, S. Faily, and M. Winandy, editors, *Proceedings of the Workshop on Web Applications and Secure Hardware (WASH), Co-located with the 6th International Conference on Trust and Trustworthy Computing (TRUST 2013)*, volume 1011 of *CEUR Workshop Proceedings*, pages 32–38, London, United Kingdom, 6 2013. Sun SITE Central Europe, RWTH Aachen University. Position Paper.

Parts of the, originally Bluetooth-based, software framework for the mobile attestation token was implemented by Manuel Schallar and Herwig Guggi [134] in a master-level class project supervised by the author together with Martin Pirker and Tobias Vejda. The NFC transmission design, performance measurements and hardware experiments that help validating our approach were performed by Michael Hutter.

4.1.1 Motivation and Background

Security enforced by software can be manipulated by software based attacks. To overcome this dilemma, the Trusted Computing Group (TCG) [345] has defined a set of specifications of which the *Trusted Platform Module* (TPM) is the central component. It is a system component deeply embedded in a machine's hardware and software architecture. One of its mechanisms, called *Remote Attestation*, reports the platform's state to another host on the Internet. This helps to establish cryptographically qualified and tamper-evident assurance on the software configuration of a machine.

As attestation allows one to determine the absence of malicious software, it is desirable for a user to perform it prior to providing sensitive or confidential data to a computer system. This is especially interesting for computers openly available in public places where users simply walk up to such machines when they need the services offered. Currently, such computers are highly exposed and threatened by a variety of software-based attacks in the form of viruses, key-loggers and root kits. Therefore, public systems for ad-hoc use cannot be trusted to handle sensitive information such as account logins, passwords or other private data – unless their configuration were properly attested and found secure for the user's purposes. However, several additional challenges need to be overcome to achieve a telling indication of trustworthiness in a usage scenario where the user physically confronts the system.

While the TPM may be trusted to perform securely as specified, it does not offer a secure local interface to display the gathered results to a user. Therefore, the need for a trusted display or an indicative token arises, lest a malicious public machine fakes reports on its state. As McCune et al. [216] point out, it would be desirable to equip the user with an ideal, axiomatically trustworthy device, which they call *iTurtle*. It would then indicate the security of a device to the user. A more practical implementation of attestation in kiosk scenarios using off-the-shelf smart phones is demonstrated by Garris et al. [120]. Still, the TCG's attestation protocol does not guarantee that the TPM is located within the machine the user faces. Following this insight, Parno [246] proposes a direct link between the user and the TPM, so that human inter-actors can themselves establish the proximity of the attesting machine.

In this chapter we build on these previous results and introduce two novel improvements. First, considering the resource limitations of mobile devices, currently proposed schemes are not flexible and scalable enough. We demonstrate an efficient, user-friendly solution that combines smart phones with a trusted third party. Second, to include a proof-of-locality in the process, we propose to introduce Near Field Communication (NFC) technology in the TCG's security architecture and present a proof-of-concept implementation.

The remainder of this chapter is organized as follows. We next outline Local Attestation and NFC technology. We discuss related work and present the locality-aware scenario we consider, followed by an introduction to our approach, in Section 4.2. In Section 4.3, our Mobile Attestation Token Architecture is presented, and we discuss the integration of NFC in the TPM, introduce new TPM commands, and present the definition of an air-interface protocol between the TPM and a mobile NFC device. Section 4.4 gives implementation details and we discuss a number of possible extensions and related experiments in Section 4.5. We summarize in Section 4.6.

4.1.2 From Remote to Local Attestation

As discussed throughout Sections 2.3-2.5, the TCG remote attestation architecture requires that the host sends a very detailed description of its system state to a verifier, which is signed by the TPM. For privacy protection, only pseudonymous AIKs are used. An AIK key can be used one or several times, whenever an attester requests the attestation of the trusted platform. Verifiers can determine the correctness of the signature after confirming the validity of the certificate and querying a revocation service.

Essential to the overall attestation process is to collect and later analyze a complete and meaningful set of state information, either based on a static or dynamic RTM. Still, using only the quote result and measurement log, coming to a trust decision remains a tedious and complex task and the number of possible combinations of secure software configurations in today's open system architectures is often quite large. Alternative concepts to reach meaningful conclusions at state analysis include *Property-based Attestation* [64,185,278], where the state analysis is delegated to a specialized *Trusted Third Party* (TTP) which issues certificates for specific properties.

Attestation is useful to improve the security for a number of computing services, including not only remote but, as we believe, also physically present systems. In general various types of systems may be encountered in different usage scenarios.

For instance, a user might want to learn if a public general purpose desktop computer is secure for ad-hoc use. Customers would like to be assured that the software of a point-of-sales terminal in a shop will not collect their PIN together with the information on the magnetic stripe of their credit card for later frauds. The same holds true for other types of Automatic Teller Machines (ATMs) and payment terminals. For instance, according to a 2009 report by ENISA [108], the real-world theft of card details and PINs had been performed on Russian ATMs infected with malware.

Vending machines' software could be reconfigured by attackers to collect cash but not to release their goods; local attestation could prevent this too. Other security critical applications may also be found in embedded systems or even peripherals like printers or access points. Here, a service technician might find a method to identify the exact software configuration and its integrity to be useful. Giving voters a method to validate that electronic voting machines have not been tampered might assist to add trust to a poll's outcome.

For easier illustration of our approach in the remainder of this chapter, we will limit our description to just one specific and instructive scenario, kiosk computers, which we will describe in Section 4.2.2. Our solution does not follow assumptions defined for confidential architectures as found in banking applications or strict legal requirements as required for eVoting solutions. We believe that the proposed approach can be modified to specific needs of such other, more specialized scenarios with reasonable effort.

4.1.3 Near Field Communication

Near Field Communication (NFC) is a wireless communication technology [9, 359] that provides a platform for many applications such as mobile ticketing, contact-less payment, and interactive smart posters. One key feature of NFC is the simple data acquisition just by touching an object with an NFC-enabled reader. NFC readers might be integrated in mobile phones or digital cameras that transfer information to another device in their *proximity.* There exist two different modes for a communication between a reader (initiator) and a target device. In passive communication mode, the initiator provides an electromagnetic (EM) field which is used to power the target device and which allows both parties a bidirectional communication. In active communication mode, both the initiator and the target device provide alternately[1] generated EM fields so that both devices require an active power supply.

NFC is based on the Radio Frequency Identification (RFID) [112] technology that operates at 13.56 [MHz] frequency. As opposed to RFID, NFC follows several specifications that have been standardized by the International Organization for Standardization (ISO) and the European Computer Manufacturers Association (ECMA). These standards specify the used data modulation, coding, frame formats, data rates, and also the application-specific transport protocol. The NFC interface and protocol (NFCIP-1) is standardized in ISO/IEC 18092 and ECMA 340 and also in ISO/IEC 21481 and ECMA 352 (NFCIP-2). In addition to these specifications, there are additional definitions from the NFC Forum which is a non-profit organization that promotes the use of NFC in electronic devices. Among many other definitions, they defined a common frame format for transmitting different media types such as Multipurpose Internet Mail Extension (MIME) objects and Uniform Resource Locators (URL). This frame format is called NFC Data Exchange Format (NDEF) and can be used to automatically start an application on a mobile phone or to display a message after touching a target object.

In contrast to other wireless communication technologies which are designed for a large communication range, NFC enables short-distance communication between electronic devices. The typical operating distance between two NFC devices is only a few centimeters (up to 10 cm). Thus, a fixed location of an NFC tag (passively or actively powered) can provide evidence whether a mobile NFC device (or its user) is at that location. Besides this evidence, NFC offers a very intuitive way for the user to communicate with a target object by simply bringing the devices close together (touching). It follows the very natural principle for communication between only two locally present entities.

NFC offers a wide range of new applications with a key focus on easy-to-use products and touch-based solutions. Since 2004, NXP Semiconductors, Nokia, and Sony invited many other global leading companies from mobile industry, electronics, payment services, and multimedia enterprises to join the NFC Forum which, as of 2013, has more than 170 members. Around 150 million NFC-devices

[1]A device will deactivate its field while waiting for data.

were shipped in 2012, which corresponds to around 15% of the smartphone market [31].

4.2 Attestation Of Local Platforms

Remote Attestation, as devised by the TCG industry consortium, achieves trust decisions between different network hosts. However it cannot be applied in an important field of application — the identification of physically encountered computer platforms and their security status to the human user. The cryptographic protocols that actually perform the attestation do not provide for human-intelligible trust status analysis, easily graspable conveyance of results nor the intuitive identification of the computer platform involved. Therefore, the user needs a small portable device, a token, to interact with local computer platforms. It can perform an attestation protocol, report the result to the user, even if the display the user faces cannot be trusted and may be connected to the platform under test.

In recognition of this, McCune et al. [216] propose the idealistic concept of an *iTurtle* device which should by design be trusted *axiomatically.* To achieve such self-evident, yet *user-observable verification* it should be as simple as possible, even without support for cryptography, and thus easy to understand and certify. The authors envision an USB device with a mere two red and green LEDs indicating the trust status. Regarding integration in the TCG's cryptographic schemes, the authors argue that this would be too complex for such a reduced device and point out the challenge of state analysis on a restricted device.

In a more practical approach, together with Daniel Hein et al. we studied a scenario-specific, special-purposes cryptographically plug-in token that will release a cryptographic key to a trusted host only (see Section 4.5.2).

Alternatively to special hardware tokens, powerful PDAs and smart phones have demonstrated [240, 293] their applicability as trusted portable device to work in conjunction with a trusted server and an untrusted public terminal to act as a secure keyboard and GUI to the user, but without performing remote attestation.

In a first combination with the TCG architecture, McCune et al. [215] demonstrate a mobile phone application which uses 2-D barcodes on stickers to identify a public key of devices like printers or IEEE 802.11 access points. The authors also consider integration in TPM-based attestation protocols. This early architecture does not guarantee the identity and standard-conformance of the TPM, i.e., it lacks the proper use of the AIK credentials.

The specific case of attesting a public kiosk computer as available in lobbies or transportation terminals been studied in detail by Garris et al. [120], also using a mobile phone. A user wishing to use a kiosk first uses the camera of her smartphone to scan the barcode containing the hash of the AIK certificate of the kiosk. The phone then connects to the kiosk using Bluetooth. The kiosk now transmits the set of configurations it supports. The set is pre-defined and signed by the kiosk's operator, which has to be trusted. Now the user chooses a configuration and the kiosk reboots to build a fresh chain-of-trust. After it is on-line again, the phone performs an attestation protocol, compares the reported configuration against the chosen one and validates that the `TPM_Quote` result is indeed signed with the same AIK to make sure that the quote comes from the same physical terminal. The user is informed of the result, i.e., the trust status is displayed on her phone. She can then use the kiosk's applications or even take advantage of the virtualized kiosk software architecture. Here, the user may supply a private virtual machine image containing her choice of software and data, cf. [57]. In the end, the user logs out.

Thus remote attestation can be used in Kiosk scenarios. However, the TPM mechanisms do not prevent a relevant class of attacks, "*platform-in-the-middle attacks*". Parno [246], who analyzed these attacks, calls them *Cuckoo-attacks*. These schemes are sometimes also known as Mafia fraud attack [87] or chess grandmaster problem. In essence, an attacker uses an honest entity as oracle to solve the challenges of her target. Malware on a compromised local machine relays TPM messages to another TPM on a remote machine which is in a trusted state. The author concludes that a *local binding* between user and TPM is needed. If the user is in possession of a trusted hand-held device, this may be achieved physically via a special hard-wired interface or cryptographically by providing users with a key by means of a sticker on the machine casing. Regarding to wire-interfaces, Li et al. [191] describe adding a serial interface to the TPM to access protected data for backup and also for providing authorization to a few restricted operations that require the physical presence of the TPM owner. No additional functionality is offered to users.

An approach for the kiosk scenario is presented by Bangerter et al. [35]. They coin the term "Ad-hoc Attestation" for their scheme which is based on a commercially available token equipped with an optical sensor and a display. It generates a nonce which is manually entered into the kiosk by the user. Due to the restricted hardware resources of the token, the state analysis is performed on a server. The server is operated and controlled by the token vendor. Once the server has made a trust decision, it then *binds* the result to the kiosk's TPM. To complete the protocol, the user needs to point the token's sensor to the kiosk display. The kiosk then unbinds and communicates the trust result via flickering black-and-white images to the token. Thus, the server is able to remotely trigger the token to display the cryptographically protected result. Note, that the user is restricted in his choice of trustworthy configurations, as he has to trust in the commercial token vendor and server operator who performs the decisions in his stead. Due to the combination of the attestation protocol with the binding mechanism this scheme achieves protection against platform-in-the-middle attacks.

Winkler and Rinner present and interactive scheme [360] for visual user-based attestation of a smart camera which features a TPM: A smart phone displays an attestation request as QR-code which is shown to the camera by the user. The camera device then attests itself via a wire-less communication link. The mobile phone may consult a trusted third party to analyze the quote. If successful, this phase shows that the quote is processed by a TPM-equipped camera in a trusted state. In a second round, a second image is generated by the phone and the camera has to sign it with a key derived from the same AIK as before. This shows that the image was taken and processed by the same camera, which is known to be in a trusted state, and thus will not forward the challenge to another device. This achieves resilience against Cuckoo attacks. Note, that the authors did not consider a visual relay attack, where both phases of the protocol are captured by a malicious camera and then relayed through a display mounted in front of a trustworthy camera[2].

An orthogonal challenge to asserting the user of a trustworthy software configuration on the host platform of the TPM is to establish a *trusted channel* from software to input and output devices. Zhou et al. [375] demonstrate in a first prototype on x86 systems how an ASCII channel can be established from keyboard to the graphics adapter. They however also assume the initial TPM-based attestation of the hypervisor that creates that channel through a hand-held device.

4.2.1 Open Challenges

Based on the presented literature we identify the following additional challenges for the ad-hoc attestation of physically present public computer terminals.

Flexible and Scalable Trust decisions. The display of a public computer must not be trusted to securely show the trust decision as malware display a fake statement. To convey the trust status to a user, a mobile attestation token is a suitable mechanism to provide a direct display and a secure communication channel to the TPM. Several existing implementations not only display the result, but also perform the trust decision on the mobile attestation device. How the collection of meaningful measurement values on the public computer can be achieved is discussed in Section 2.5. Note that the performance of mobile devices may limit the size of the known-good-value repository and the complexity of the state analysis needed for the trust decision. Also, if only a small set of possible configurations is provided, none of them might match the specific security requirements of the user. A priori stored reference values also limit the flexibility in case of system updates, or when encountering terminals from unexpected operators. The same holds true for proprietary servers, which force a user to trust in their operator and his policies.

Direct, Local Channel between User Token and TPM. Practical proposals have so far considered different interfaces such as Bluetooth or USB. However, as outlined in [246], both technologies require an honest software stack to forward their messages to the passive TPM device. A direct, wired physical channel would require extensive changes to the TPM design and new standard

[2]We recognize that performing this hypothetical attack scheme might, in most scenarios be relatively easy to detect and prevent.

plugs, both being expensive and impractical. Flickering displays provide only one-way communications and require users to obtain and trust a special propose token. Displaying and scanning QR-codes or taking images in multiple phases requires lengthy, time-consuming interactions between the user and the device which is challenged. Bluetooth has a long radio range and thus it could also connect to a neighboring kiosk. To prevent this, current proposals introduce stickers that identify TPM keys to link attestation to a physical machine. However, stickers are easy to manipulate [195]: Foremost, it is extremely easy to copy and print them (with the attacker posing as a legitimate user, taking the photo with his mobile phone camera). The manipulated sticker can easily be placed on top of the original one within seconds by a casual attacker and thus fake the identity of another kiosk. This is exactly the setting for the cuckoo attack we wish to prevent.

4.2.2 Scenario: Kiosk Computing

Kiosk computers are often found at shops, in hotel lobbies, transportation terminals or Internet cafés; they are public terminals to provide applications like Web browsers or ticket-vending services. Such kiosks are rarely deployed alone; in most cases several similar devices are operated in close vicinity. We assume that the kiosk are equipped with persistent storage like a hard drive and can therefore store data and programs over power cycles. We also assume that the Kiosk operator offers the infrastructure for attestation, such as the hardware interfaces and the necessary software services.

As the kiosk is in a public location, also attackers can visit the kiosk repeatedly during operation hours and pretend to be legitimate users. In our scenario, Attackers are assumed to potentially have full control over the software running on the kiosk, thus software cannot be trusted at all and key-loggers and fake security tools must be assumed. We further assume that wireless communications can be eavesdropped.

With TPM-based attestation we desire to provide the user with means to establish trust in such a device, but of course we have to consider the limitations of the TCG's architecture: it is not designed to protect against hardware attacks (see Section 2.7). We assume that this is compensated by operational measures, i.e., even if the devices may be unattended, they will be physically protected, i.e., by robust casings fixed to the ground, integrated keyboards and displays that prevent hardware based attacks. Also, we assume that the operator performs hardware and software maintenance on a regular basis, thus making most hardware attack schemes like adding malicious devices to the casing impractical. As a natural consequence, the service technician, much like any computer systems administrator, must be trusted.

Figure 4.1 further illustrates the scenario we envision. It shows the following four situations.

1. When a user encounters several terminals available in a room he has no means to establish the trustworthiness of a single device. Therefore he

might choose a malicious kiosk computer for his task, unintentionally exposing private information to an attacker.

2. In this case we illustrate a successful attestation of a trustworthy kiosk. The user is equipped with a smart phone which serves as attestation token, similar the scheme that has been demonstrated by [120, 215]. Note that the individual kiosks are identified by barcode tags attached to them.

3. In this situation, only one of the kiosks has not been manipulated. On the others, the attacker installed malware that fakes a trusted behavior and the unprotected kiosk identification tags have been manipulated: they were copied from the untampered machine (on the right side). Due to the long range of Bluetooth communications, this attack compromises the scheme of [120]: each time the user identifies a kiosk with his smartphone, he communicates with the same (secure) platform, while he in fact faces a maliciously modified machine.

4. The last situation illustrates our proposal. NFC is used to establish which is the kiosk that actually performs the attestation. With this mechanism in place, the user can recognize that the first, closest machine is not secure and will refrain from performing any critical task on it.

In the next section we will show how this local attestation with machine-in-the-middle resistance can be achieved. The result is a trust decision, which is reported to the user. Which assets a user will choose to expose to the so attested kiosk computer depends on the user alone and his choice of security policies.

4.3 Mobile Attestation Token

We now present two improvements. First, we outline a kiosk attestation architecture which is designed to be user-controlled, flexible and scalable with regard to kiosk state analysis. This first protocol, however, will not be resistant against machine-in-the-middle attacks. Then, in Section 4.3.2, we will detail how NFC can be integrated in attestation, thus providing for direct user token to TPM communications. This effects the desired attack resilience, under the restrictions discussed in Section 4.4.3.

4.3.1 The MAT Protocol

We now present a first version of a protocol that allows the attestation of a kiosk computer using a smartphone. This version does not yet utilize the NFC link.

In our scheme, three parties collaborate to perform a cryptographic protocol. We assume that the *Kiosk* contains a TPM and an operating system that offers a complete chain-of-trust and measurement services that allow a verifier the extraction of meaningful properties and allow to make a sound trust decision. Secondly, the *Mobile Attestation Token* (MAT) is the client the user installs on his mobile phone. Finally, we introduce a trusted third party, the *Verification*

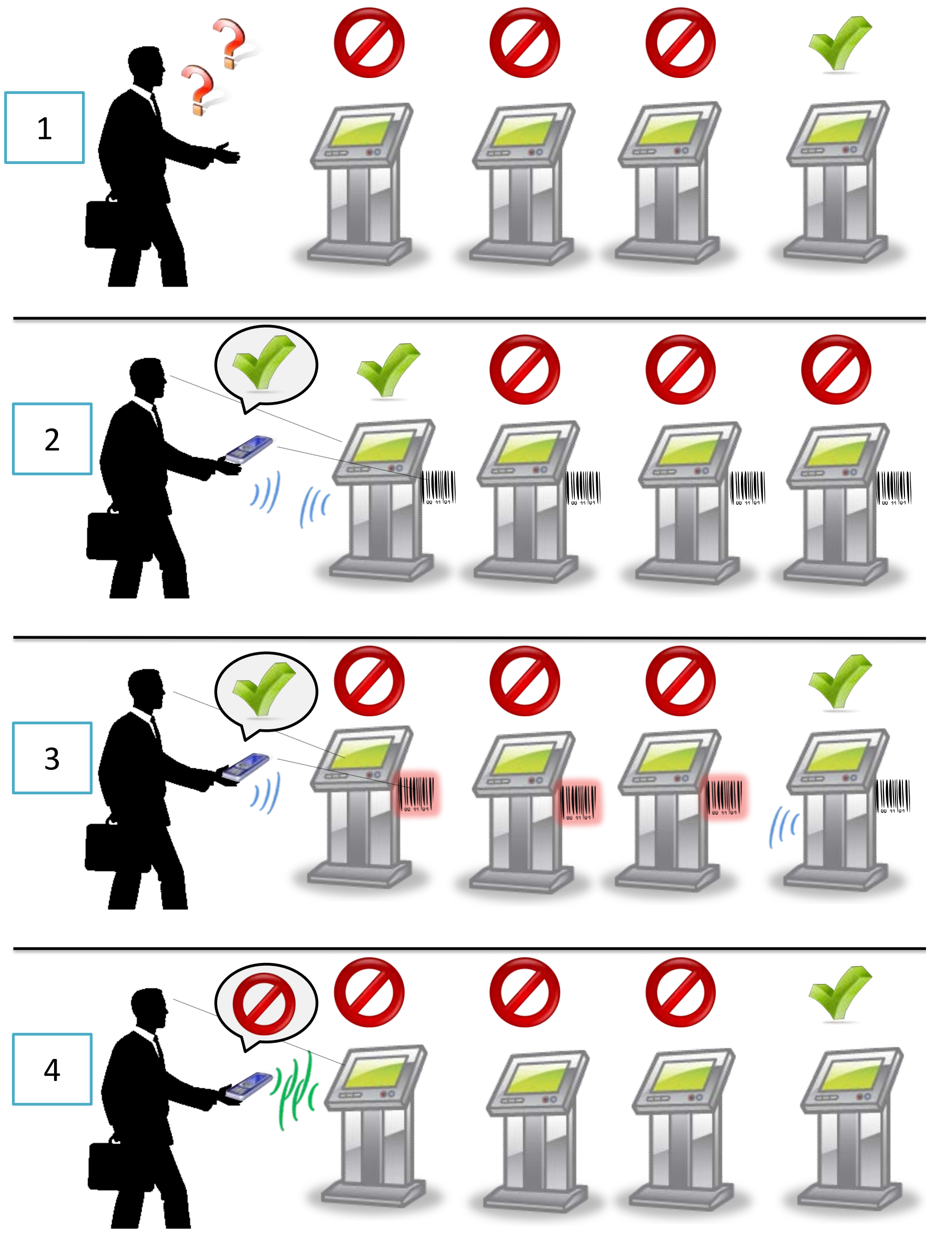


Figure 4.1: Description of the Motivating Usecase (1), Proposed Solutions by [120, 215] (2), a Possible Attack Scenario (3), and our Proposed Solution (4).

Server. For better illustration, the protocol flow is shown in Figure 4.2. The figure also includes additions, which will be explained in Section 4.3.3. These additions, which represent direct NFC communications are shown in dotted lines. Our initial protocol proposal follows. We assume that an appropriate PKI has been set up and that the kiosk operator has provides the necessary certificate chain on each kiosk or on-line.

0. In an initial setup step, the user *chooses* the Verification Server she trusts and configures it with policies according to her preferences and needs. She then transfers the public key certificates and the URL of the Verification Server to the MAT.

1. When, at some later point of time, encountering a public computer the user connects the MAT to the attestation service installed on the kiosk.

2. In the beginning of the attestation protocol, the MAT generates a nonce N_a to provide fresh data for replay protection.

3. With the first message to the kiosk, the user initiates the attestation with her MAT, transmitting the URL of the Verification Server she intends to use and N_a.

4. The kiosk will then ask a quote of its recorded system state from the TPM in the kiosk.

5. The quote PCR $.\mathbf{S}_{\mathrm{AIK}^{-1}}(V_{\mathrm{PCR}}, N_a)$, i.e., the current PCR values and the signature over all PCR registers under an Attestation Identity Key (AIK), is created within the TPM and passed on to the attestation service of the kiosk.

6. Now, the kiosk establishes a secure TLS connection to the Verification Server via the URL provided by the MAT. It transmits the quote together with the AIK certificate and the stored measurement log SML which documents the chain-of-trust of the kiosk in detail.

7. With the so collated information, the Verification Server analyzes the quote and decides the trustworthiness according to the detailed requirements of the user, using its local or other suitable known-good-value services[3] or property extraction methods. It also validates that AIK is indeed a TPM protected key and asserts that the PrivacyCA did not revoke the certificate.

8. Once a trust decision is made, the Verification Server assembles and returns a ticket to the kiosk: $\mathbf{S}_{\mathrm{VS}^{-1}}(\texttt{trusted}.N_a, S_{\mathrm{AIK}^{-1}}(V_{\mathrm{PCR}}.N_a), \texttt{text})$. It contains a binary trust decisions (the `trusted`-bit) and a free `text` for additional messages and is signed by the Verification Server's private key VS^{-1}.

[3]Depending on the market environment of the Kiosk operator, these might exist or not.

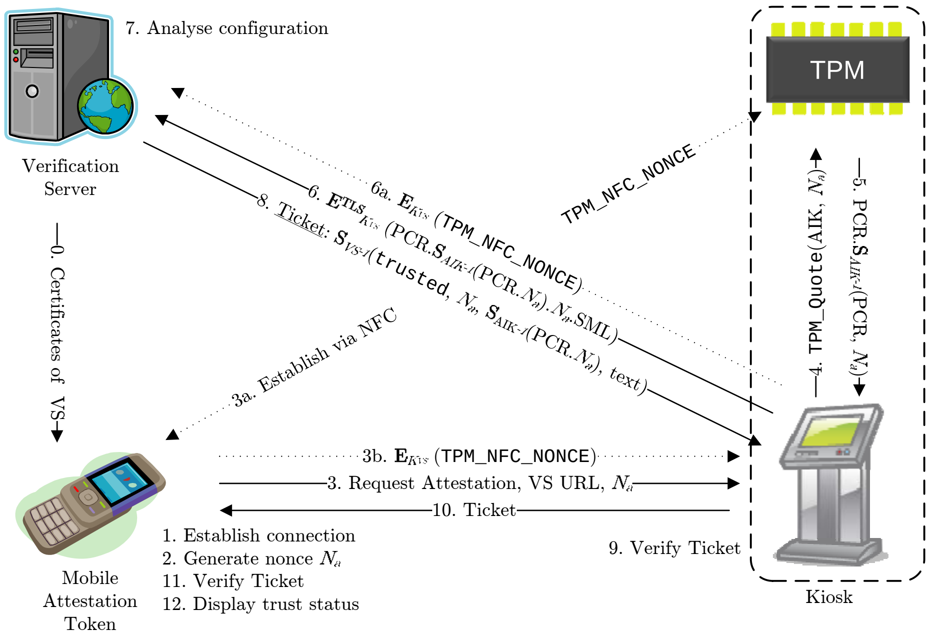


Figure 4.2: Ticket-based Local Attestation Scheme with MAT and Trusted Verification Server. The optional steps 3a, 3b, and 6a are only performed with the NFC extension as described in Section 4.3.3.

9. The ticket is validated on the Verification Server before being passed on to the MAT. This allows a kiosk operator to gather statistics on how much acceptance the offered configuration finds.

10. The unmodified ticket is passed on to the MAT.

11. The MAT verifies the signature using the pre-installed Verification Server certificate and also that N_a is from the same session.

12. If this succeeds, it can then proceed to finally display the result to the user. Here, intelligible icons can be used to illustrate the `trusted`-bit.

The protocol attests the kiosk configuration to the Mobile Attestation Token. The central design decision is to source out the complex state analysis and decision procedure to the Verification Server. Thus we delegate [74] the analysis of the attestation information to a proxy [64] chosen by the user. Thus, the capability to decide on the trustworthiness of the kiosk is not limited by the restricted resources on the mobile device. Therefore the number of kiosk configurations and the complexity of analysis (i.e., hash-based comparisons or property extraction) considered will not influence the local performance of the protocol. The on-line Verification Server will also be able to check the validity of AIK certificates on-line, so that kiosk identities need not be specified by the user beforehand. Indeed, as many attestation identities may be used as a trusted PrivacyCA vouches for. With all this performed remotely, our scheme requires the MAT only to validate the signature of the ticket, the nonce and the trusted bit. The protocol therefore *scales* with the number of kiosks and their configurations.

Also our protocol does not require any vendor specific operations or tokens — it can be implemented on any device capable of the small set of cryptographic functions and the necessary communication interfaces. In addition, there are no limitations on who operates the Verification Server. It could be the kiosk operator as well as the user or any commercial or open institution. The Verification Server might even consult other services to help it decide on a reported system state. This *flexibility* not only allows user to make sound trust decisions, but also provides easy adaption to changing profiles.

Thus, our proposal is not only a scalable and flexible technical solution, but at the same time it also protects the users' right of self-determination in their trust decisions.

4.3.2 An NFC Interface for the TPM

We now outline how the TPM could be extended with an NFC interface to create a direct channel to the MAT. We believe that this will not require extensive changes to the TPM design. NFC has been designed to be integrated in small hardware solutions like smart cards which are very similar to many TPM implementations. Furthermore, many of the challenges that have to be overcome in the design of passive NFC tags are not an issue with the TPM. For instance, the TPM has an active power supply and full cryptographic capabilities. Only a simple, passive Radio Frequency (RF) interface is needed, and the antenna circuit could be printed on the mainboard of the host machine[4]. We believe that this NFC interface is potentially cheaper than a proprietary wired interface, which would require modifications to TPM, board, casing and MAT.

In this way it is possible to establish a direct link from the attestation device to the TPM. Note that in our approach any software on the kiosk is circumvented, thus preventing any kind of *software*-based attacks on the communication link[5].

We now describe how the MAT and the kiosk can agree on a nonce over the NFC and how we integrate this nonce in step 5 of the MAT protocol, thus creating an implicit proof of locality.

New TPM Commands

To introduce NFC-based nonce agreement, changes to the TPM itself should be limited to a minimum. As a special purpose trusted component, the TPM should not provide more features than necessary to perform its tasks and therefore should not operate as a full flexible NFC reader to the host. Also, changes to the TPM API should be minimal and not affect normal operations. The TPM

[4]Assuming a non-shielded casing.
[5]Except for denial-of-service.

Incoming Parameters

Parameter #	[byte]	Type	Name	Description
1	2	`TPM_TAG`	tag	`TPM_TAG_RQU_COMMAND`
2	4	`UINT32`	paramSize	Total number of input bytes including paramSize and tag
3	4	`TPM_COMMAND_CODE`	ordinal	Command ordinal: `TPM_ORD_establishNonce_NFC`

Outgoing Parameters

Parameter #	[byte]	Type	Name	Description
1	2	`TPM_TAG`	tag	`TPM_TAG_RSP_COMMAND`
2	4	`UINT32`	paramSize	Total number of input bytes including paramSize and tag
3	4	`TPM_RESULT`	ret.	Result code of the operation
4	4	`TPM_COMMAND_CODE`	ordinal	Command ordinal: `TPM_ORD_establishNonce_NFC`

Table 4.1: The `TPM_establishNonce_NFC` command establishes a shared nonce between remote NFC reader and TPM. The resulting nonce is not returned to the host machine but retained in the protection of the TPM.

specifications describe the mechanism of creating a quote comprehensively and authoritatively. Here, we only present the proposed changes to the TCG TPM 1.2 specification [345] in this section.

The RF interface is to be activated only in the Enabled-Activated-Owned state of the TPM life-cycle (see Section 2.3.3) and an owner-authorized call to `TPM_SetCapability` is needed to activate the permanent flag `enableNFCInterface` that enables the following operations.

We introduce a new command that allows the NFC reader and the TPM, which have no prior knowledge of each other's identity, nor a shared key, to jointly establish a shared secret over the NFC channel. This secret can then be used as nonce in a *single* subsequent TPM operation. The `TPM_establishNonce_NFC` command is described in Table 4.1. It is important to notice that if the command returns with `TPM_SUCCESS`, the nonce is not returned to the TPM's host machine but retained in a special volatile and protected register `TPM_NFC_NONCE` inside the TPM. This register can be read-accessed as if it were an additional PCR, but with one exception: it is always reset to zero after a read operation. If the protocol fails, or times out, appropriate error codes are returned. `TPM_establishNonce_NFC` does not require authorization, as it only stores the nonce. All commands that use its result must be properly authenticated. The command itself performs a standard Diffie-Hellman-Merkle key-agreement operation which is described in detail in the next section.

Minor changes are now needed for TPM commands that utilize this nonce, for instance, `TPM_Quote`. It is called with a `TPM_PCR_SELECTION` that indicates the PCR registers to consider. The behavior is extended as follows: `TPM_NFC_NONCE` is selected like other PCRs with index: number of normal PCRs + 1. If `TPM_NFC_NONCE` is zero, the command terminates with error code `TPM_NO_NFC_NONCE`, else its value is, as in the unmodified TPM, hashed in the `TPM_PCR_COMPOSITE` together with the PCRs and finally signed with the provided AIK. The values of all used registers are also returned to the host. The `TPM_NFC_NONCE` register is then set to zero.

This way, the quote result depends on the `TPM_NFC_NONCE` that was previously agreed upon by the NFC reader (i.e., the Mobile Attestation Token) and the TPM. As the quote result is signed with an AIK, this links `TPM_NFC_NONCE` to an authentic TPM. Each nonce can only be used once, thus guaranteeing freshness. Other commands which access PCRs can be adapted in a similar way[6], without changing their signature.

The NFC Nonce-Agreement Protocol

Essentially, the security of NFC is based on the physical characteristics of the electromagnetic near field, which limit the operational range to about 10 [cm]. Still, eavesdropping attacks remain a threat in NFC applications that can be performed even at a distance [138].

In order to establish a shared nonce between the TPM and the NFC-enabled reader, we propose to use a classical key-agreement scheme according to Diffie-Hellman-Merkle [92] (DH). This scheme is quite simple to implement on both embedded devices and provides two major advantages. First, a nonce is established within a two-way communication process. The TPM and the NFC reader (e.g. a mobile phone) agree on a shared nonce without the need of installing any a priori secrets on the device. Second, the nonce is never transmitted in plaintext so that a potential attacker cannot extract the nonce by simply eavesdropping the communication.

We base our NFC nonce-agreement protocol, developed together with Michael Hutter, on two NFC standards: the ECMA 385 [98] and ECMA 386 [99]. These standards define security services and protocols for NFC communication. The standards specify several schemes for a secure communication channel and a shared secret service for ECMA 340-enabled [97] devices. These standards define cryptographic mechanisms that use the Elliptic Curves Diffie-Hellman (ECDH) protocol for key agreement and the AES algorithm for data encryption and integrity. However, for the TPM, can we only assume the cryptographic abilities which can be found in TPM version 1.2 [345], i.e., RSA but neither Elliptic-Curve Cryptography (ECC) nor AES. Therefore, we implemented a non-elliptic-curve

[6]The `TPM_NFC_NONCE` *must* be reset to zero after every kind of read.

$$
\begin{array}{lll}
0.1 & \mathrm{MAT}: & Q_A = g^a \bmod p \\
0.2 & \mathrm{TPM}: & Q_B = g^b \bmod p \\
1 & \mathrm{MAT} \rightarrow \mathrm{TPM}: & \texttt{ACT\_REQ}(Q_A) \\
2 & \mathrm{TPM} \rightarrow \mathrm{MAT}: & \texttt{ACT\_REQ}(Q_B) \\
3.1 & \mathrm{MAT}: & Z = Q_B^a \bmod p \\
3.2 & \mathrm{TPM}: & Z = Q_A^b \bmod p
\end{array}
$$

Figure 4.3: NFC Nonce-Agreement Protocol between Mobile Attestation Token and TPM.

version of DH for our NFC demonstrator. Note that the protocol domain parameters such the prime p, and the primitive root g need to be known by both entities.

In Figure 4.3, the NFC nonce-agreement protocol is shown. The TPM and the mobile phone agree on a shared secret key Z as follows. First, each entity calculates a public number (Q_A and Q_B). Second, the mobile phone sends its number Q_A plus the required parameters p and g to the TPM using the `ACT_REQ` command according to the ECMA 386 standard. The data is encoded in a byte structure (`TPM_NFC_DH_PARMS`) which is shown in Figure 4.4. Third, the TPM sends the similarly encoded number Q_B to the mobile phone using the `ACT_RES` command. In the last step of the key agreement protocol, both entities compute the shared secret Z according to the DH primitive as specified in IEEE 1363 [157] (6.2.1 DLSVDP-DH), with SHA-1(Z) being the value of `TPM_NFC_NONCE`.

```
TPM_NFC_DH_PARMS
  TPM_STRUCTURE_TAG tag;
  UNIT32 root;
  UINT32 keyLength;
  BYTE* key;
  UINT32 primeLength;
  BYTE* prime;
```

Figure 4.4: The `TPM_NFC_DH_PARMS` data structure holds the information transmitted in the DH scheme.

4.3.3 Integration of NFC with the MAT Protocol

We now present the modifications necessary to the MAT protocol to achieve resilience against machine-in-the-middle attacks, based on the described TPM modifications.

The protocol can be easily integrated into the MAT protocol described in Section 4.3.1. The following steps have to be added. The complete protocol flow including the additional NFC steps is shown in Figure 4.2. The additions are drawn in dotted lines.

3. (a) When the MAT and the kiosk's TPM touch, the key-agreement protocol as described above is executed and both entities share knowledge of `TPM_NFC_NONCE`.

   (b) The secret nonce is encrypted by the MAT under the public key of Verification Server and forwarded to the kiosk.

4. Note that `TPM_NFC_NONCE` is implicitly included in the quote result by the TPM.

6. (a) The Kiosk passes the encrypted nonce on to Verification Server.

7. The trusted bit is only true, if the same `TPM_NFC_NONCE` was received from the MAT, because it is contained in the TPM quote. Note that the TPM with its AIK implicitly guarantees for the authenticity of the nonce origin.

With these changes, our attestation protocol first performs a secure and eaves-dropping resistant nonce exchange between TPM and MAT. It then uses this nonce in the quote operation. Therefore the physical proof-of-locality is implicitly guaranteed after completion of the protocol.

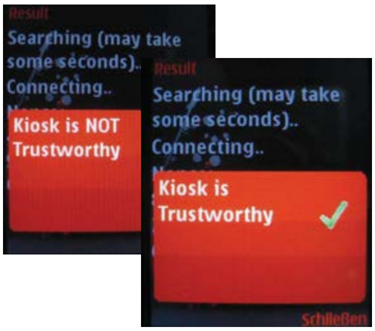
4.4 Implementation and Validation

In this section we report on a series of experiments that validate the concept and protocol of a Mobile Attestation Token that employs a proximity interface.

4.4.1 Mobile Attestation Token and Kiosk Software Platform

An initial software-prototype implementing attestation validation was created by Manuel Schallar and Herwig Guggi under supervision of the author. The MAT prototype implementation is based on commodity hardware and on platform-independent software. On the Verification Server, Java SE is used. The Verification Server stores reference known-good-values in a relational MySQL database, which is accessed using Hibernate. A GUI tool allows the user to collect reference measurements. The Mobile Attestation Token is built as applet for Java ME, MIDP 2.0, extended with JSR 82 (Bluetooth/OBEX support), JSR 75 (PDA profile) and JSR 257 (NFC support). For cryptographic support we use IAIK JCE ME on all hosts. The MAT software is thus compatible to NFC-enabled phones such as the Nokia 6212. Figure 4.5 shows typical screenshots on the MAT. As no NFC-enabled TPM is currently available in hardware, we simulate the high level MAT protocol communications using Bluetooth and separately demonstrate the NFC interface.

On the Kiosk, we currently collect binary measurements, and accept plug-ins for further analysis. The TPM can be accessed from Java SE using IAIK jTSS (see Section 3.2.2).



Searching (may take some seconds)..
Connecting..
Kiosk is NOT Trustworthy

Searching (may take some seconds)..
Connecting..
Kiosk is Trustworthy ✓

Figure 4.5: The Mobile Attestation Token Software Informs the User on the Result of the Attestation Process in a Comprehensible way.

The complexity of a typical Kiosk system can be assumed to be comparable to that of a Linux installation with graphical user interface. An experiment to this end was made by Martin Pirker, on a customized Linux system. Based on November 2009 source files from the Gentoo distribution, a restricted base system, including the `fvwm2` X-Window Manager and the Firefox web browser are covered. The kernel includes the IMA [279] Linux Security Module (LSM) which performs measurements of all accessed files. The platform performs a measured boot and launches the browser afterwards. The chain-of-trust created by this typically consists of around 520 different IMA measurements. Note that the exact order and number of programs and libraries loaded depends on external factors like network services or user interactions.

This result is of comparable magnitude to the complexity assessment by Lyle and Martin [203], who identified 277 different measurements for a basic web service and about five time as many security-relevant updates over a three-year period. Based on this we believe that several thousand of known-good-values will need to be managed for each kiosk software platform and its updates.

4.4.2 NFC Demonstrator

In order to demonstrate an autonomic and NFC-compatible interface for a TPM, Michael Hutter performed experiments based on the IAIK HF DemoTag device[7].

The prototype represents a TPM that is assembled on a Printed Circuit Board (PCB). As TPMs available on the market cannot be freely programmed or extended with additional communication interfaces, a we use an 8-bit micro-controller in our experiments. Moreover, the DemoTag integrates an antenna into the PCB as an easy access point that can be touched with an NFC-enabled mobile phone. The antenna has been designed according to ISO/IEC 7810 [164] and has a size of a conventional smart card (ID-1 format). Besides the micro-controller and the NFC antenna, the prototype consists of an analog front-end, a clock oscillator, a serial interface, a Joint Test Action Group (JTAG) interface,

[7]`http://www.iaik.tugraz.at/content/research/rfid/tag_emulators/`

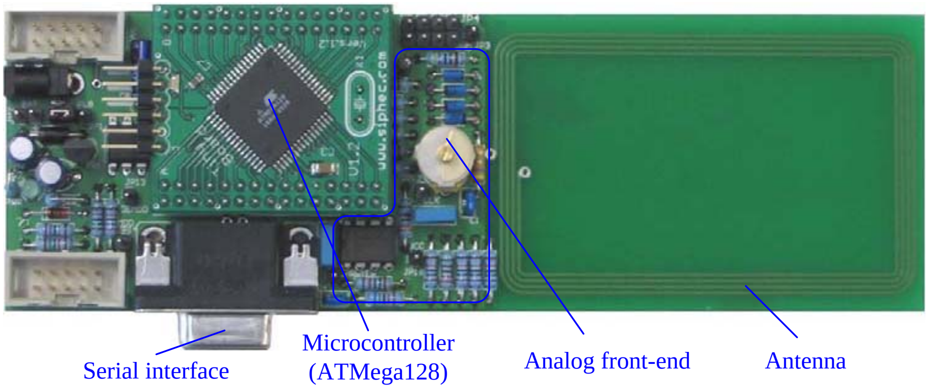


Figure 4.6: NFC Demonstrator Printed Circuit Board.

and a power-supply connector. It operates at 13.56 MHz and has been assembled using discrete components. In Figure 4.6, a picture of our prototype is shown.

As a microcontroller, the ATmega128 [26] has been used, which is responsible for managing NFC-reader requests and target responses by following the specification of the NFC Forum. Next to the air interface, the microcontroller is able to communicate with a PC over a serial interface. The JTAG interface enables debug control and system programming. The NFC prototype is semi-passive which means that the microcontroller is powered by an external power source while the RF communication is done passively without any signal amplification.

The analog front-end is responsible to transform the analog signals of the NFC reader into digital signals used for the microcontroller. Note that in prodution devices, the analog front-end and the digital circuit are typically integrated into one piece of silicon like in passive RFID tags. The analog front-end is mainly composed of an antenna matching circuit, a rectifier, a voltage regulation unit, a demodulation and a modulation circuit.

The NFC prototype can communicate using several protocol standards. It implements several RFID protocols such as ISO/IEC 15693, ISO/IEC 14443 (type A and B), ISO/IEC 14443-4 and also ISO/IEC 18092. The software is written in C while parts have been implemented in assembly language due to timing constraints. Moreover, it implements a user-command interface that allows one easy administration over the serial interface. For our experiments, we have used the ISO/IEC 14443-A [163] protocol standard up to layer 4 using ISO/IEC 7816-4 [162] Application Protocol Data Units (APDU) according to the NFC Forum type 4 tag operation specification [237].

For our experiments, we implemented the key-agreement protocol described in Section 4.3.2 on our NFC prototype and also on an NFC-compatible mobile phone (Nokia 6212). Both devices are capable of transmitting NDEF messages, which enables an automatic key-agreement between the NFC prototype and the mobile phone by simply touching the antenna of the prototype with the mobile phone. After that, the running time of the proposed NFC protocol was measured using an 8-bit oscilloscope. First, the mobile phone sends a request command (REQA) to the NFC prototype which answers with its unique ID (UID) number after an anti-collision and initialization phase. This phase takes about 22.5 [ms] in our experiments. Second, the mobile phone calculates the public key using a private key and prime size of 768 bits. After that, it sends the generated `TPM_NFC_DH_PARAMS` structure as a payload of the `ACT_REQ` command to the NFC prototype. The size of the payload is 1648 bits which take about 141.8 [ms] to transmit. The time between the first bit transmitted and the last bit received is measured. Third, the same computation is performed by the NFC prototype which answers with an `ACT_RES` command using the same payload size in 141.8 [ms]. In the experiments, the entire key-agreement protocol can be performed within one second.

4.4.3 NFC Robustness against Man-in-the-Middle Attacks

The security of NFC is mainly based on the physical characteristics of the electromagnetic near field: the field strength for inductive coupling degrades cubically with the distance, which results in a typical range of about 10 [cm]. It should be noted, especially as the physical layer is typically not encrypted, that eavesdropping might be possible, even at a distance [138]. Classical man-in-the-middle attacks can be prevented due to the characteristics of transmission parameters, which allow the reader to sense manipulations in the field [140]. However, there exist also the threat of relay attacks on RFID and NFC devices using custom built [137] or even commercial-off-the-shelf [115] RF-hardware devices. Relay attacks are related to man-in-the-middle attacks where the data of a trusted device is simply forwarded by an adversary so that the local presence of the trusted device is not ensured anymore.

Alpár and Hoepman [11] point out that this attack would also affect our scheme and propose two generic additions to increase security, by including either i) location-based information, distance-bounding or multi-channel communications or ii) involving the TPM during the whole communication session, or iii) both. The second proposal is not straightforward, as the TPM does not offer support for continuous encryption, such as in TLS, which led us to create the piggyback scheme of adding the nonce as 'virtual PCR' in the first place. Regarding the author's first proposal, the possibility of physical relay attacks was already mentioned in our initial publication [320], together with a possible research direction, distance bounding protocols. These distance bounding protocols are an approach to prevent relay attacks [4,30,49,230,267,347]. However, in NFC the distance itself is very small, making distance bounding protocols a challenge to implement. Achieving effective and efficient distance bounding is still an active research topic [30] and implementations are are not generally[8] available yet.

[8]According to [4], the NXP Mifare Plus product range already offers distance bounding in contact-less tokens.

4.4.4 Validation

Based on the Experiments described above we have shown that

- A Mobile Attestation Token Software application can be implemented on Java-enabled mobile phones, even in the low-performance feature-phone class.
- The expected configuration of Kiosk computers can be described in a few hundreds, or in case of updates, a few thousand known-good-values. This amount of information can be handled on any Desktop-class Verification Server.
- The nonce-exchange introduced in Section 4.3.3, can be physically transmitted using standards-compliant NFC protocols in less than one second.
- NFC effectively prevents man-in-the-middle attacks that happen on the protocol or network layer. Protection against relay attacks on the physical layer is currently an emerging but promising technology.

Thus, the approach can be realized on consumer mobile devices and offers a consumer-friendly reaction time, making it suitable to the scenario outlined previously. Our scheme extends on what TPMs are capable of today and in the near future. While Dmitrienko et al. [93] see it as a disadvantage that our proposed interface is not commercially available in certified TPM implementations, we would like to point out that this would clearly be desirable, but it is also a matter of economical feasibility in the global semiconductor and systems market.

4.5 Extensions and further Experiments

We now outline future research we conducted on the fringes of the work so far presented in this chapter, which had been largely based on our 2009 publication [326]. We now present an optimization of the NFC-based attestation protocols, discuss attestation against even smaller systems and outline advances on platform security in mobile devices.

4.5.1 Touch‘n’ Trust

In two additional publications [152,153] we have explored together with Hutter a second, different point in the design space for NFC-based attestation. The main difference is that the trust decision is moved into the mobile device. This is enabled by substantial optimizations that include an advanced terminal hardware and software platform and a more efficient cryptographic protocol.

The terminal platform is improved by implementing the acTvSM (see Section 5.8.2) approach on it, resulting in a simpler comparison of known-good-values. Consequently, the space of acceptable values is much smaller, as structured file systems allow the booting platform a complete measurement of the overall system configurations within only a few hashes and can therefore be compared easily with reference values in recent, powerful smart phones. This step can be performed with the mobile device alone, without the Verification Server.

The second idea is to omit the Nonce-agreement phase of the NFC-enabled MAT protocol, but instead to make the quote data structure more compact, and thus reduce time needed for the over-the-air communication of the full quote. To achieve this improvement, we propose to use elliptic curve cryptography (ECC) to increase the computation and communication performance.

The first step in the alternative attestation protocol is to generate a nonce N_0 with the NFC-enabled mobile phone which again acts as a reader device (initiator). As soon as the mobile phone touches the RF antenna of the terminal, the nonce is transmitted over the air interface within a configuration challenge (the nonce serves as fresh data to avoid replay attacks). After that, the public terminal (target) responds with the Quote of its currently recorded terminal state. A modified[9] Quote data structure which contains the nonce, the current PCR values, and the signature over the selected PCR registers under an Attestation Identity Key AIK of the TPM, is then returned to the mobile phone.

As the over-the-air transmission time is the limiting factor of using NFC, we propose the use of elliptic curve cryptography (ECC) for signing the PCRs. In contrast to other public-key primitives like RSA, ECC has gathered much attention due to the use of shorter key sizes. The computation time and especially the communication time over the air interface can be significantly improved by providing the same cryptographic strength. For instance, the strength of a 2048-bit RSA key can be compared to that of a 224-bit ECC key. ECC has been widely standardized, for instance by ANSI [13], IEEE [158], NIST [233], and ISO/IEC [165]. Also, there already exist public-key infrastructures that support this algorithm for signing and verifying data and X.509 certificates [166]. In order to sign the PCRs we propose to use Elliptic Curve Digital Signature Algorithm (ECDSA) according to the recommendations of the National Institute of Standards and Technology (NIST) [233]. We use the smallest recommended elliptic curve, that is 192-bits, for prime-field arithmetic.

A number of arithmetic optimizations for ECC, which are detailed in [153] facilitate the efficient implementation on resource-constrained micro controllers while supporting reliance to side-channel attacks [206].

The digital-signature generation of the 8-bit ATmega128 microcontroller that takes the TPM's role in our experiments, takes about 31 million clock cycles. Due to the characteristics of our scenario it is sufficient to consider only the performance of a single session. Running at 13.56 [MHz] this results in a running time of about 2.31 [s] to generate the signature. The verification of the signed message on the mobile phone takes 33 [ms]. The anti-collision and initialization phase of the NFC protocol needs about 22 [ms]. Second, the challenge N_0 and the Quote response are transmitted. For this, we assumed a typical number

[9]Static elements need not be transmitted over the air, and would only incur longer transmission times.

of different PCR values, viz., 6 in our experiments, resulting in a payload of $160 + 6 * 160 + 192 = 1312$ bits. The transmission takes about 140 [ms] using a fixed RFID/NFC data rate of 106 [kbit/s].

Existing 1.2 TPMs already include cryptographic services such as RSA, where significantly more bits would have to be transmitted in contrast to elliptic curve implementations. As a comparison, the transmission of a 1024-bit RSA-based signature (comparable with a 160-bit ECC implementation) alone would need 2192 bits, or roughly 240 [ms] transmission time which is almost twice as high as the elliptic-curve based attestation protocol. This motivated our design decision, as we desire to keep the time the user needs to touch the public terminal as short as possible.

If the signature is validated successfully, the Quote information is compared to a list of known-good PCR values. A quote is only accepted if the state report contains only trusted values.

Thus, this optimized scheme achieves the attestation without the need of an external verification service, but against only a smaller set of possible public computing terminals. Depending on the organizational measures taken beforehand, it can be applied off-line, without Internet connection. Ideally, though, the phone should validate the AIK certificate in the context of a trusted computing PKI [255] and confirm that it has not been revoked. The entire attestation process can be performed within three seconds, without hardware cryptography accelerator. The presentation of this scheme in [152] has been recognized with a “best paper award” to Hutter and Toegl at ICSNC 2010.

Still, which scheme is better suited for a concrete scenario depends on the specific organizational requirements.

4.5.2 Local Attestation with a Dedicated Hardware Device

It may appear difficult to come to a practical repository of acceptable trusted states such that a meaningful trust decision can be made locally inside the mobile phone.

Together with Hein et al., we proved the viability for one specific scenario, the protection of a software framework for disaster relief support, in [141–143] by showing attestation against an even smaller embedded system: a dedicated single-chip token. The *Autonomous Attestation Token (AAT)* is a small, user-owned token to attach to or to plug into any available device, be it a laptop computer or a mobile phone. As a hardware security token the AAT could potentially provide various cryptographic services. Based a the close cooperation of TPM and AAT, we define a new mechanism, which we refer to as *Local Attestation*. Users, in our example emergency service workers, are issued [84] an AAT which identifies them. When it is plugged in an mobile device, the platforms state will be conveyed via the TPM’s standard quote operation, but only to the directly connected token. Of course, the platform software needs to trigger measurements and reporting to the AAT. The AAT will validate the signature and compare the PCR values to a set of known good configurations. If the platform is deemed trustworthy, the AAT signals this to the user and she can authorize an operation. In [142] we describe a basic use case where a network master access key is only revealed by the AAT if the host platform and the user are trusted and authentic. Local attestation implements a true, decentralized trust decision process, which does not rely on on-line third parties such as a PrivacyCA. It can easily accommodate different platforms with varying valid configurations. The AAT, in addition to performing the platform state verification, also serves as a proof of possession in authentication protocols. Therefore, local attestation not only ensures a specific state of the platform, but also provides strong evidence of the identity of the owner. The AAT has been implemented by Daniel Hein on a tamper-resilient Infineon security controller, and demonstrated attestation of a mobile agent framework optimized for disaster response [143]. Again, the attestation was facilitated through structuring the software configuration according to the acTvSM approach that will be explained in more detail in the next chapter (Section 5.8.2).

4.5.3 Isolation and Integrity Protection on the Mobile Device

In the approach we have described so far, we have in essence moved the user interface of the trust decision one device closer to the user, onto her personal mobile phone. Basically, using the phone as attestation respectively authentication device to other machines introduces it as a second authentication factor besides the user, making the process more resilient.

However, modern smart phones have reached a level of richness in features that is comparable to those of the public computing terminals we intend to secure. As a consequence, smart phones can also be attacked on a software level [154], recreating the initial challenge [216]. Still, we would like to argue that the ecosystem of mobile operating systems is better controlled and thus offers a higher chance of warranting a degree of trust in them. This is especially plausible, as there is a very promising line of research on how to increase the security of mobile platforms by circumventing the consumer operating system on them altogether. The solution currently followed by large parts of the industry is to offer a highly reliable, secure computation environments. To us, these environments appear to be very well suited for attestation token implementations.

One of the dominant processor architectures employed in current mobile and embedded devices is the ARM architecture. Current ARM-based processor designs span a wide range of application fields, ranging from tiny embedded devices (e.g. ARM Cortex-M3) to powerful multi-core systems (e.g. ARM Cortex-A9 MPCore). ARM introduced a set of hardware-based security extensions called TrustZone [21] to ARM processor cores and on-chip components. For cost reasons integrated into the CPU core, ARM TrustZone is an emerging technology to provide security features without the need of extra hardware chips.

The key foundation of ARM TrustZone is the introduction of a *secure world* and a *non-secure world* operating mode. This secure world and non-secure world mode split is an orthogonal concept to the privileged/unprivileged modes already found on earlier ARM cores. On a typical ARM TrustZone core, secure world and non-secure world versions of all privileged and unprivileged processor modes co-exist. A number of System Control Coprocessor registers, including all registers relevant to virtual memory, exist in separate banked secure and non-secure world versions. Security critical processor core status bits (interrupt flags) and System Control Coprocessor registers are either totally inaccessible to non-secure world or access permissions are strictly under the control of secure world. For the purpose of interfacing between secure and non-secure world a special Secure Monitor Mode together with a Secure Monitor Call instruction exists. Depending on the register settings of the processor core, interrupts are routed to Secure Monitor Mode handlers. Apart from the extensions to the processor core itself, The AMBA AXI bus in a TrustZone enabled system carries extra signals to indicate the originating world for any bus cycles. Thus, TrustZone aware System-On-Chip (SoC) peripherals can interpret those extra signals to restrict access to secure world only; a secure world executive can closely monitor any non-secure world attempts to access secure world peripherals. In essence, an ARM TrustZone CPU core can be seen as two virtual CPU cores with different privileges and a strictly controlled communication interface.

The realization of the two-worlds paradigm in both security- and consumer-operating-system software is largely vendor specific. Still, there are specifications and standards available. Previously, ARM had published its own TrustZone software API specification [20]. Together with Trusted Logic, ARM has developed a closed-source TrustZone software stack, complementing the TrustZone hardware extensions. ARM has since donated its TrustZone API to the GlobalPlatform industry association and this has developed into the Trusted Execution Environment (TEE) Client API [127]. It allows an application in the "non-secure world", which typically runs a rich-OS such as Google Android or Microsoft Windows Mobile 8, to communicate with the "secure world". ARM has also been working with other companies to develop the TEE Internal API [128] that interfaces between a Trusted OS, running in the secure world, and a Trusted Application.

Today, many modern ARM-based Smartphones (with Cortex-A CPU) include a TEE based on SoCs by manufacturers like Qualcomm, Samsung, Nvidia, and Texas Instruments. Accordingly, TEEs have already been deployed for several years, featuring Trusted OSes currently made by TrustedLogic/Gemalto (Trusted Foundation) or Giesecke & Devrient (Mobicore). Moreover, ARM, Gemalto and Giesecke & Devrient and others have recently created the TRUSTONIC Joint Venture on TEE Trusted OS and its ecosystem of services.

Within a TEE, a Trusted OS is the basis to execute use-case specific Trusted Applications. Several scientific publications deal with proposals for secure mobile and embedded system designs based on the ARM TrustZone security extensions [101, 102, 102, 181] and their possible applications for instance in digital rights management [150], cryptographic protocols [356], mobile ticketing [151] or wireless sensor networks [372]. Aspects of virtualization based on TrustZone are discussed by [116, 188, 361]. A TrustZone protected, next generation TPM is reportedly shipping in Windows 8 RT devices.

Together with Winter et al. [364] we showcase an experimental open-source software environment for experiments with ARM TrustZone. The software framework offers a prototype kernel running within a trusted environment and features a software based Trusted Platform Module hosted in a TrustZone protected runtime environment and an Android operating system accessing it through the high-level, JSR 321 Trusted Computing API (see Chapter 3). The joint work received the "Best Paper Award" at INTRUST 2011 conference.

Overall, TrustZone based TEEs seem suitable for local attestation with physical devices and there are various proposals on how to create a direct visual channel from the TEE to the user [349]. This ongoing development towards higher security in mobile devices thus supports the Mobile Attestation Token approach, as a reliable execution environment and also channel to the user is becoming realizable in real-world, consumer devices.

4.6 Summary

In this chapter we consider challenges that arise in Remote Attestation scenarios with locally present computer systems. We conclude that a trusted mobile device, the Mobile Attestation Token, is a suitable instrument to interact with local computer platforms. It will perform an attestation protocol, report the result to the user, even if the display the user faces cannot be trusted and may be connected to the platform under test.

We extend on previous proposals in this field to provide more scalability considering the limited computational power and memory of mobile devices, and add flexibility by moving the complex state analysis to a trusted third party. Our scheme can be implemented without special purpose hardware and is not restricted to specific operators. While it does not overcome all complexities of attestation, our scheme gives users full control which verification server to use. Thus security requirements and trust policies can be defined individually and to maintain the users' full self-determination in their trust decisions.

Furthermore, we describe how to add a direct, affordable interface to the TPM. With Near Field Communication, a proof of locality is embedded in the attestation process. The presented extension allows us to completely circumvent any malicious software and thus prevent software-based platform-in-the-middle attacks on public available computers and other devices.

The chapter also presents collaborative research results on the fringes of the original protocol, which describe performance improvements, single-chip local attestation and inclusion of TEEs on the mobile device.

5 Rigorous Design of Trusted Services

5.1 Introduction

Trusted Computing mechanisms that enforce binary integrity and runtime isolation do not per se guarantee a secure and correct *behavior* of a service. Yet, all (meaningful) services need to communicate, and they do so through well-defined interfaces. These interfaces expose software services to the network to other hosts, or to the Internet in general and any (software) attack will and must pass through them. These interfaces are security protocols, or, if they allow remote parties more persistent operations, security APIs. These interfaces, however, can be restricted to enforce precise specification compliance and thus prevent malicious behavior.

One way to achieve assurance on a security behavior is to design and implement systems, interfaces and protocols rigorously, proving and verifying their correctness and robustness against attacks. In this chapter, we investigate how formal methods can be applied in the context of two different services that are protected by Trusted Computing mechanisms such as the TPM and Intel TXT.

In the first part, Sections 5.2-5.5, we study a secrets distribution protocol that employs the binding mechanism of the TPM to ensure that only trusted hosts have access to a secret. We apply formal methods to highlight a subtle, yet potentially relevant security issue and propose an improvement.

In the second part, Sections 5.6-5.13, we report on our design of a *virtual* security module that deeply integrates trusted computing mechanisms. Our approach uses commodity personal computer hardware to offer integrity protection and strong isolation to a security module which offers signature services. The module provides only a compact, yet proven secure API.

Declarations
This chapter extensively adapts, cites and reuses previously published material from the author, especially

[325] R. Toegl, G. Hofferek, K. Greimel, A. H. Y. Leung, R.-W. Phan, and R. Bloem. Formal analysis of a TPM-based secrets distribution and storage scheme. In *Proceedings TRUSTCOM 2008, in: Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 2289–2294. IEEE Computer Society, 2008.

[336] 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*, pages 145–160. Springer Berlin Heidelberg, 2013.

[323] R. Toegl. Verification of a trusted virtual security module. In M. Bond, R. Focardi, F. Sibylle, and G. Steel, editors, *Analysis of Security APIs (Dagstuhl Seminar 12482)*, Dagstuhl Reports, page 166. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013. Abstract.

Besides these publications, this chapter also adapts, cites and reuses material from internal deliverables [256, 257, 269, 322, 332, 335] of the acTvSM project, co-authored with Martin Pirker, Georg Lindsberger, Andreas Niederl, Gerhard Fliess, and Florian Reimair.

The vulnerability of the protocol described in 5.4.1 was originally discovered by Adrian Leung and later described in his PhD thesis [190].
The model for the protocol described in 5.2.1 was programmed by Georg Hofferek and Karin Greimel as part of class practicals supervised by the author.
The security module prototype was programmed and its performance measured by Florian Reimair [268] as his master's thesis, supervised by the author.
The acTvSM platform has been designed and implemented by Martin Pirker, Michael Gissing, Andreas Niederl, Michael Gebetsroither and the author and it is detailed in [126, 253, 254, 333].

5.2 Formal Analysis of a Secret Distribution Scheme

As previously outlined in Section 2.6, the design of cryptographic protocols is a demanding task and many vulnerabilities have been discovered in various protocols. With the advent of Trusted Computing came the temptation to rely on the TPM's security mechanisms to achieve goals like authenticity and confidentiality in communication protocols. However, even with hardware security in place, similar logical problems as with conventional protocols may still occur, for instance failure to mutually authenticate two parties. While general purpose cryptographic protocols have been the subject of formal analysis for many years, the field of Trusted Computing has seen only limited use of this.

Specific to Trusted Computing, Bruschi et al. [53] model the authentication session that is performed between the local application and the TPM module in the Spin model checker [147]. Lin [193] formally analyses parts of the TPM API in the theorem prover Otter. Gürgens et al. [135] perform a systematic security evaluation of the TPM specification the SH-verification tool (SHVT) to model several TPM-based protocols and consequently suggested clarifications in the TPM 1.2 specification. Chen and Ryan [66] study the authentication protocols of TPM command discovering a security issue with shared authentication data and propose an improved protocol verified with ProVerif. Coker et al. [74] propose a multi-party attestation protocol and perform an analysis applying the Cryptographic Protocol Shapes Analyzer (CPSA). Recently, Delaune et al. [85] introduce a way to model protocols including the TPM PCRs through a formal Horn-clause-based framework, which allows analysts the application of ProVerif even on such stateful behaviors. Beyond stateful protocols, security APIs are also a target for rigorous analysis, as we will see in Section 5.7.2.

We now describe a formal analysis of a protocol proposed by Sevinç et al. [292] for "securely distributing and storing secrets", "independent of a specific usage-control application". The protocol "ensures that the server only distributes given secret data to trusted clients." It does not, however, ensure that the secret received by the client does indeed come from the server. This latter property is not mentioned in [292], but we argue that it is important in some applications, such as when decisions are based on the contents of the secret. We show the absence of this mutual authentication property using a model checker and suggest an improvement of the protocol which does not suffer from the same drawback.

In the remainder of this section, we introduce the protocol we analyze, and outline the fundamentals of Model Checking. In Section 5.3, we describe our model of the protocol and the attacker in the NuSMV model checker. As presented in Section 5.4, our model unveils a security flaw which illustrates the risks of integrating complex TPM operations such as Binding in network protocols. We also discuss how the design can be improved and how this can be verified with our model. The discussion of the protocol concludes in Section 5.5.

5.2.1 Protocol

Sevinç et al. propose a scheme to securely distribute a secret using trusted computing functionalities [292]. The setting is that the server does not trust the client platform but only the TPM residing at the client's end. It is possible for the server to ascertain with the help of the PCRs that the client platform is in a *trusted state*. Then, the server can expect the client platform to function in the desired manner, rather than (intentionally or unintentionally) running some malicious activity.

Protocol Flow. The table in Figure 5.1 shows the flow of the protocol. We stick to the numbering used in [292], but use a slightly different notation.

1.			C	$\rightarrow$	S	REQ
2.			C	$\leftarrow$	S	$V_{\mathrm{PCR}}.N$
3.	TPM	$\leftarrow$	C			`TPM_CreateWrapKey(`$\mathbf{A}$`,`$\mathrm{h}(P)$`,...)`
4.	TPM					fresh (K, K^{-1}) with policy
						$\mathbf{A} = \{V_{\mathrm{PCR}}, \texttt{non-migratable}, \texttt{binding}\}$
5.	TPM	$\rightarrow$	C			$K.\mathbf{E}_P(\mathbf{A}.K^{-1})$
6.	TPM	$\leftarrow$	C			`TPM_LoadKey2(`$\mathrm{h}(P)$`,`$K.\mathbf{E}_P(\mathbf{A}.K^{-1})$`,...)`
7.	TPM	$\rightarrow$	C			$\mathrm{h}(K)$
8.	TPM	$\leftarrow$	C			`TPM_CertifyKey(`$\mathrm{h}(K), \mathrm{h}(\mathrm{AIK}), N$`)`
9.	TPM	$\rightarrow$	C			$\mathbf{S}_{\mathrm{AIK}^{-1}}(\mathbf{A}, N, K)$
10.			C	$\rightarrow$	S	$\mathbf{S}_{\mathrm{AIK}^{-1}}(\mathbf{A}, N, K).\mathbf{S}_{\mathrm{CA}^{-1}}(\mathrm{AIK}, K_{\mathrm{AIK}})$
11.					S	check certificates and $\mathbf{A}$ of K
12.			C	$\leftarrow$	S	$\mathbf{E}_K$(`SECRET`)
13.	TPM	$\leftarrow$	C			$\mathbf{E}_K$(`SECRET`)
14.	TPM					assert current *trusted* state $V'_{\mathrm{PCR}} \supseteq V_{\mathrm{PCR}} \in \mathbf{A}$
15.	TPM	$\rightarrow$	C			`SECRET`

Figure 5.1: The Key Distribution Protocol of [292].

There are three participants in the protocol, the client's TPM, the client, and the server. The TPM is assumed to provide a non-migratable storage key P, and some certified and readily loaded AIK, both available through their handles $\mathrm{h}(P), \mathrm{h}(\mathrm{AIK})$. First the client sends an initial request (REQ) to the server. The server replies by sending desired values for PCRs (V_{PCR}) which define the *trusted state*, and a nonce N. In the third step, the client asks the TPM to create a key pair (K, K^{-1}) which the TPM restricts with the policy $\mathbf{A}$ of being typed as a non-migratable binding key with its use further restricted to V_{PCR}.

After generating the key, the TPM returns the public key part K; the private part K^{-1} is encrypted together with its policy under P as P is the parent key of K in the TPM key hierarchy of the client. After the encrypted key data structure has been provided to the client, the client requests the TPM in step 6 to load the key pair into a key slot. As a result, in step 7, the TPM returns a handle $\mathrm{h}(K)$ to the key, which can now be used in cryptographic operations. Next, the client asks the TPM to certify key K (step 8). The TPM subsequently signs the public key K together with its policy and the nonce N under an AIK. This certificate is sent to the server (step 10). Now the server verifies the certificate for K and asserts the policy $\mathbf{A}$ that the key K is non-migratable and sealed to the correct PCR values (step 11). The server also needs to verify the AIK's certificate, which we assume given here. Subsequently, in step 12, the server encrypts the secret under K. The encrypted message is then sent to the client which asks the TPM to decrypt it (step 13). The TPM checks if the client is in the *trusted state* defined as the $V_{\text{PCR}} \in \mathbf{A}$. If and only if so, the secret is revealed to the client in plaintext.

Security Targets. We identify the following three security targets:

1. An intruder never learns the secret.
2. The client learns the secret only if it is in a *trusted state.*
3. A client in the trusted state either learns the (real) secret, or discovers that an attack has taken place.

Sevinç et al. state that the main goal of the protocol is to ensure the confidentiality of the secret, i.e., neither an intruder nor a client in an untrusted state may learn the secret. This corresponds to our first two security targets. The third target, not mentioned in [292], ensures that one of the following cases always occurs: (1) An honest client is capable of successfully completing the protocol and thus learns the secret, or (2) the client is able to detect that some malicious activity occurred.

Below, we formalize these security targets and verify them using a model checker.

5.2.2 Model Checking

Model checking is used to formally prove or disprove that certain properties hold for a given model. For example, we prove that given security properties hold for a protocol. A model checker tool takes a finite model and a specification, a set of properties, as input and returns true if and only if the model satisfies the specification. If the model does not satisfy the specification the model checker returns false. Most model checkers are able to give a counterexample showing why the model does not satisfy the specification.

In the following, we will shortly describe the models, the specification language, and the model checker we use.

Models. Our models are Finite State Machines (FSMs) or more formally Kripke structures. Let AP be a set of atomic propositions. A *Kripke structure* is a tuple $K = (S, T, S_0, A, L)$, where S is the finite set of states, $T \subseteq S \times S$ is the complete transition relation, $S_0 \subseteq S$ is the set of initial states, $A = 2^{AP}$ is the alphabet and $L : S \to A$ is the labeling function, which associates with every state the set of propositions that hold in that state. An infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ is a *path* of K if $\forall i. (s_i, s_{i+1}) \in T$. A path is a *run* of K if additionally, it starts with an initial state. The corresponding *word* $\sigma = L(s_0)L(s_1)L(s_2)\ldots$ is an infinite sequence of letters from the alphabet A, defined by the labeling function. In order to reason about properties of the set of words a Kripke structure defines a specification language is needed.

Specifications. The specification language we use is *Computation Tree Logic* (CTL) [70]. The logic is defined over a finite set of propositions, AP, the same set we used to define the labeling of Kripke structures. If we fix a Kripke structure, a CTL formula is satisfied by a set of states. We now define the syntax of CTL inductively and give a brief overview of its semantics. Let s be a state. If p is an atomic proposition, and φ and ψ are CTL formulas, the following are also CTL formulas: p (meaning that $p \in L(s)$), $\mathsf{AX}\,\varphi$ (meaning that φ holds for all successors of s), $\mathsf{AF}\,\varphi$ (meaning that for every path starting in s, φ holds eventually), $\mathsf{AG}\,\varphi$ (meaning that for all paths starting in s, φ holds in all states), and $\mathsf{A}\,\varphi\,\mathsf{U}\,\psi$ (meaning that on all paths starting in s, ψ holds eventually and φ holds in all prior states). We can use Boolean connectives to obtain $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\neg\varphi$, with the usual meaning. We say that a formula holds for a Kripke structure if the formula is satisfied by all initial states.

Model Checker. We use the NuSMV model checker [67] by Cimatti et al. Given a Kripke structure and a CTL specification, it is able to decide if the specification holds for the Kripke structure. The NuSMV modeling language allows for a very easy description of Kripke structures and the CTL model checking functionality is based on a fast symbolic algorithm. For the subset of CTL that interests us, NuSMV produces a counterexample whenever the model violates a formula. For instance, a counterexample for an $\mathsf{AG}\,\varphi$ formula consists of a single run of the model where φ does not always hold. A counterexample for an $\mathsf{AF}\,\varphi$ formula consists of a run of the model where φ never holds.

5.3 Modeling the Protocol

In order to model-check the protocol,we need a model that represents the protocol flow. We model the protocol actors in four FSMs, one for each party of the protocol: client, TPM, server, and intruder. The states of these FSMs represent the steps of the protocol and the content of the messages, modeled as shared variables. Thus, the state of the overall system determines (1) the current step of the protocol for each party, (2) the content of the messages sent so far, and (3) the current knowledge of each party.

As the protocol outcome depends on V_{PCR}, we need to include the TPM's PCR state into our model; this statefulness is not supported in the high-level modeling languages of specialized security protocol analysis tools (see Section 2.6.2). To overcome this, we use a general purpose model checker instead.

In order to describe the model of the intruder, we will first lay down some assumptions on the capabilities of the intruder.

5.3.1 Assumptions

Cryptography. Our assumptions on security and intruder capabilities are based on the considerations of Dolev and Yao [94], which we restate here from Section 2.6.1. We assume that the underlying cryptographic primitives are perfect and that keys and message fields are atomic. Thus, an intruder can read an encrypted message if and only if it knows the correct key. Similarly, an intruder can only create signed messages with signature keys it knows. An intruder is not able to learn partial information of a secret key or message, thus the intruder either knows it completely or not at all. Our model therefore does not cover attacks like statistical analysis or differential cryptanalysis, nor attacks based on mathematical or number-theoretic properties of the underlying cryptographic systems. We further assume that all parties know all public keys.

Model Restrictions. Our model introduces some further restrictions. First, we only model one run of the protocol. Thus, we do not consider replay or interleaving attacks. The model is further restricted in that it allows only for limited fresh data. Such restrictions have been, for *stateless* protocols, overcome by more specialized tools (see Section 2.6.2). Furthermore, since the server trusts only the client's TPM but not the client itself, our model includes some malicious client behavior. We do not model the complex API of the TPM. We thus assume that it is impossible to circumvent the TPM's security policies by abusing the API. We, however, allow the client to pass arbitrary values from its knowledge to the TPM during the protocol run. Due to these restrictions, our model cannot be used to obtain a proof that the protocol under investigation is secure under all circumstances.

Apart from these restrictions, our model of the intruder is quite powerful. We assume the intruder to be a classical man in the middle: the intruder can intercept all messages between the client and the server. Messages between the client and the TPM cannot[1] be intercepted by the intruder. If the intruder knows the correct key, it learns the content of the message and adds it to its knowledge. The intruder can also alter messages or introduce new messages which are composed of items in its knowledge.

5.3.2 Model Details

The models of the client, the server, and the TPM are a straightforward implementation of the protocol. Whenever a party is supposed to send a message, it fills the fields of the corresponding (global) variable and then sets a flag that

[1]Eavesdropping on this channel would require i) physical access to the client's machine or ii) access to the inter-process communication on it. In case i), the TPM cannot be assumed to offer much security (see Section 2.7) anyway. In case ii), the attestation will reveal that the platform is not trusted and the protocol will abort.

the message has been sent. Then the sender remains in an inactive state until the `received` flag of the response message becomes `true`. When the receiver has received the message (possibly after the intruder has changed it), it can perform checks such as whether all expected fields in the message are non-empty, whether the message is signed with the expected key, etc. If all checks pass, the protocol continues until the state `COMPLETED` is reached. If a check fails, the party enters a state `ATTACK_DETECTED` and remains there forever.

We observe that the number of keys and other interesting items such as nonces in the protocol is limited. Thus, we represent them by a finite number of symbolic constants (`NONCE`, `CLIENT_KEY`, `SECRET`, etc.), in addition to the constant `ARBITRARY`, which represents "any other value".

While we do not trust the client in all cases, the purpose of the protocol is to determine whether it can be trusted, even with an attacker present. Therefore, we mode the intruder separately. he knowledge of the intruder is stored as an array of Boolean values, where each entry corresponds to one symbolic constant. The entry is `true` if and only if the intruder knows the information represented by the corresponding constant.

An encrypted message simply contains a field which stores the key with which the message should be decrypted. When the intruder sees the message, it checks whether it knows the key. If not, it cannot perform any actions that require knowledge of the key. For example, without the key it cannot add the content of the message to its knowledge. The same goes for signatures. If an intruder does not know the signature key of a message, it cannot change the content of the message. However, note that the intruder is allowed to create a completely new message, either unsigned or signed with another, known key.

The actions of the intruder are modeled according to the restrictions outlined above. Whenever the `sent` flag of a message is `true`, the intruder can start to operate on the message. It nondeterministically chooses to either leave the message as it is, or to construct a new one based on its current knowledge. When constructing a new message, the actual values of the fields are also chosen nondeterministically from the overall intruder's knowledge. The CTL model checker analyses *all* possible (nondeterministic) combinations. Thus, without explicitly modeling all choices in the state machine, they are all taken into account when checking the specification. After the intruder has dealt with a particular message, it sets its `received` flag to `true`. This indicates that the receiving party may continue its operation. It also prevents the intruder from making any more changes to the message. The intruder still has read access to the message, which allows it to decrypt and use a message if it learns the corresponding key later on. Notice that our model, especially the model of the intruder would be straightforward to adapt to other protocols.

5.3.3 Security Targets

When formalizing the security targets given in Section 5.2.1 we obtain the following CTL properties:

1. $\mathsf{AG}(\neg \mathit{IntruderKnowledge}[\mathsf{SECRET}])$

   The intruder never knows the secret.

2. $\mathsf{AG}((\mathit{Client.state} \neq V_{\mathrm{PCR}}) \rightarrow \neg \mathit{ClientKnowledge}[\mathsf{SECRET}])$

   If the client is not in the *trusted state*, which is described by the PCR values V_{PCR} then it does not know the secret.

3. $\mathsf{AG}((\mathit{Client.state} = V_{\mathrm{PCR}}) \wedge \neg \mathit{AttackDetected}) \rightarrow$
   $\mathsf{AF}(\mathit{ClientKnowledge}[\mathsf{SECRET}]))$

   If the client is in the *trusted state* and no attack has been detected then the client should eventually know the (real) secret.

5.4 Protocol Analysis Results

5.4.1 Model Checking

We feed our model and these properties to the NuSMV model checker. Within just a few seconds NuSMV finds the first two properties `true` and thus satisfies the specification given in [292]. The third property, however, is found to be false. An examination of the counterexample reveals a potential security issue.

Security Issues. The intruder is able to replace the last message from the server to the client. This message only consists of the encrypted secret. Thus the intruder can choose any arbitrary content it knows, encrypt it with the client's public key, and send it to the client. The client has no means of knowing whether the message has been altered or not. It can correctly decrypt the message, but instead of the secret it only learns the arbitrary content chosen by the intruder.

This prevents the two parties from establishing a shared secret. The secret value should not only remain confidential to just the server and the client, but it should also be assured that the client and server share the same secret in the end; yet this is no longer the case due to the aforementioned possibility of the intruder.

In certain scenarios this weakness does not cause a problem. Imagine a scenario in which the `SECRET` is a key to access some intellectual property (like a movie). If the client receives a faked secret, it will be unable to perform the necessary decryption operation, but otherwise no harm is done. This is no more dangerous than a denial-of-service attack.

Consider, however, another setting. Suppose the `SECRET` is a symmetric session key, which client and server want to use for confidential communication. After the protocol has been completed, the client uses the session key to encrypt confidential information (i.e. bulk data) and send it to the server. The client would think that (except for itself) only the server knows the session key. However, the session key is actually a faked one, sent by the intruder. Thus, the intruder is able to decrypt the message and learn the confidential content. The server would detect this attack, because the client's message is not encrypted with the expected session key[2]. However, at this point it is too late: confidentiality has already been compromised.

In the appendix of [292], the potential application of sending confidential documents to the director of a secret service is given as illustrative and illustrious scenario for the protocol. It is true that confidentiality will be protected by the protocol. But, as we have shown, it is also true that any fake intelligence could be sent to the director, possibly a manipulated list of double agents.

5.4.2 Enhancements

The underlying problem is that the scheme only considers the security from the server's point of view. Indeed there is no authentication from the server to the client, neither in message 2 nor message 12. In fact, the message $Enc_K(\texttt{SECRET})$ cannot be linked to the other messages nor to any value representing the current protocol session. The message in step 12 is publicly computable and totally independent of prior messages, except that it can only be unbound by C.

The aforementioned attack can be prevented if the server signs the message in step 12 with its own private key S^{-1}. We also suggest to include the nonce N into message 12, so that it is uniquely linked to a particular protocol session:

12. C $\leftarrow$ S $\mathbf{S}_{S^{-1}}(\mathbf{E}_K(\texttt{SECRET}), N)$

Now, if the intruder changes the content of the message the client can detect this, because the signature would be invalid. After this modification, NuSMV proved all three of our security properties to be `true`. However, we caution that the model checker cannot prove overall correctness of the protocol. It can only prove the correctness of the model of the protocol (with its limitations) with respect to the specified security properties.

5.5 Conclusions on Protocol Analysis

We have formally modeled and analyzed a scheme for the distribution of secrets with the NuSMV model checker. The main design goal of the scheme is to securely distribute a secret. Our rigorous model shows that despite TPM-based encryption of secrets, the lack of mutual authentication leads to a weakness in the cryptographic protocol, allowing an attacker to supply the client with a secret of its choice. We have modified the protocol and prove that the problem does not occur in our enhanced version.

The inclusion of Trusted Computing devices like the TPM leads to the introduction of a notion of state such that push-buttons tools for protocol analysis cannot be applied directly. This required us to create a model and specifications for a general purpose model checker. Later, in 2011, Delaune et al. [85] found a method to create PCR-dependent models for a specialized security protocols verifier.

[2]Actually, an intruder might also deny the connection to the server, preventing detection.

The TPM is no silver bullet for protocols security and it does not magically remove or resolve security issues. Actually, we found that its inclusion does make reasoning and verification of protocols more complicated and requires higher effort. Thus best practices for protocol design must still be adhered to, otherwise vulnerabilities in protocols may be introduced.

5.6 A Trusted Virtual Security Module

In the previous sections we have studied how one specialized security module, the TPM, and the interface it offers can be used correctly in network protocols. We now move on to look into how such a security module can be designed that justifies the trust in it.

In public key infrastructures the secure handling of *private* key material is of critical importance. Current mass-market implementations of secure key stores are either pure software based solutions or dedicated hardware implementations.

In most cases, simple *software key stores* are used due to their flexibility and low costs. Such *software security modules* are, by concept, rather simple software libraries. A typical client application will access them via the industry standard PKCS #11 [275] interface. Software security modules perform cryptographic operations in software on the main CPU and store sensitive data such as private keys protected, typically with a password. Some modules take care to zero sensitive data after usage, some may not. Due to their operation within platform system RAM a malware-prone platform such as the common industry standard PC platform cannot effectively prevent the access to private key material in a software security module [136].

On the other end of the market spectrum, *Hardware Security Modules* (HSMs) (see [16, 96] and Section 2.2.2) offer isolated and protected secure processing environments hardened against a wide range of software and hardware attacks. Naturally, these devices tend to be expensive.

Both types of security modules do not offer generic computing services, but are accessed through a *Security API*. Often, those APIs are HSM-specific implementations of general-purpose cryptographic API standards such as Microsoft Crypto API, PKCS #11 or JCE. Within the HSM, a special security policy must be enforced. It determines which operations are valid considering the authentication of the caller and her authorization to operate on the relevant entities. The minimally required policy is a separation between the access rights of system administrators and those of users. Modern security policies provide support for much more elaborate elements such as key typing. The literature contains a number of API attacks on such interfaces [47, 73]. Often, the critical operations are concerned with the handling of secrets, especially the sharing, backup, and migration of keys. One characteristic of this class of attacks is that they consist of valid statements, but in unexpected combinations and sequences. Put differently, the implemented security policy does not cover all degrees of freedom the security API offers. Accordingly, such vulnerabilities are difficult to discover or to prevent using a best-practices engineering approach, which is still the state-of-the-art in security API design. The design of such a security API is difficult and attacks have been found that allow for the extraction of secret key material not only in theory, but also in practice for commercial HSMs [48].

The technological convergence of Trusted Computing platforms with HSMs, as noted by Smith [300], allows for improved security levels on commodity general-purpose systems and thus offers a compromise between the two contradictory goals of minimizing costs and maximizing security. Building on these emerging technologies, we have studied the design and implementation of such a hybrid security module.

In the remainder of this section we present the design and implementation of a *Trusted Virtual Security Module* (TvSM) that offers a restricted set of security critical operations and manages cryptographic keys. A hardware-supported virtualization platform on commodity PC hardware with Intel Trusted eXecution Technology (TXT) [133] extensions offers protection against insecure applications running on the same system and further ensures the integrity of the module through the Trusted Platform Module (TPM). Within the TvSM, a flexible key hierarchy allows operators to support different use cases. As even for commercially available HSMs logical API flaws have opened the way for attacks [48], we perform a formal security verification of the module's API. The prototype implementation we present suggests that the cryptographic performance is comparable to medium-sized HSMs and a significant speedup when compared to the TPM.

5.7 TvSM Background

In this section we first revisit Trusted Computing platforms and give a more detailed explanation of how Intel TXT implements a DRTM. Second, we discuss the state of the art for API analysis.

5.7.1 Intel TXT as DRTM

We now give more details on one of the DRTM technologies, we first discussed in Section 2.5.3. A *secure boot* process using Intel's TXT be be used to ensure execution of a known-good hypervisor configuration. The `GETSEC[SENTER]` CPU instruction provides a well-defined, trusted system state. During boot, a *chain-of-trust* is established by measuring software components into the TPM's PCRs. A *measurement* m in this context is a 160 bit SHA-1 hash which is stored in PCR_i using the one-way *extend* operation, which we introduced in Section 2.3.3. We now extend the notation and define the behavior on platforms with an Intel TXT DRTM. On version 1.2 TPMs, a PCR with index $i, i \geq 0$ in state t may then be extended with m by setting

$$extend_i(m) : \text{PCR}_i^{t+1} = \text{SHA-1}(\text{PCR}_i^t \,||m). \tag{5.1}$$

Each PCR can hold 20 bytes which reset to either all zero (denoted as $\texttt{0xFF}^{20}$) or the inverse. For the PCRs' initial state ($t = 0$) we write $initial_i$

$$initial_i : \text{PCR}_i^{t_0} = \begin{cases} \texttt{0xFF}^{20} & 17 \leq i \leq 22 \text{ with static boot} \\ \texttt{0x00}^{20} & 17 \leq i \leq 22 \text{ after dynamic (re-)boot} \\ \texttt{0x00}^{20} & \text{else.} \end{cases} \tag{5.2}$$

For several $m_j, j = 1..n$ we simply write $extend_i\{m_j, .., m_n\}$. Note that after platform power-on, this operation and the CPU's DRTM call are the only way to alter the content of a PCR; extend always depends on the previous value. To achieve a certain value in a PCR the same measurements have to be extended in the same order.

The TPM also serves as access-controlled, autonomous storage for two policies which are enforced. First, a *Launch Control Policy* (LCP), which is evaluated by an *Authenticated Code Module* (ACM) provided by Intel, defines which secure boot loader is allowed to run as a so called *Measured Launch Environment* (MLE) [161]. Second, a *Verified Launch Policy* (VLP), which is evaluated by the secure boot loader, defines which subsequent kernel, `initrd` are allowed to be loaded and executed. Furthermore, mechanisms in the Intel TXT chip set offer isolated partitions of main memory and restrict *Direct Memory Accesses* (DMA) I/O to defined areas.

5.7.2 Challenges and Tools for API Analysis

The correct handling of key material through a security API is a complex challenge that has lead to the discovery of several theoretical attacks [47,58,86,371]. Clulow [73] presents the following categorization of attacks on the PKCS#11 key management API. Key conjuring is the unauthorized generation of keys. Attacks on the key integrity can ease cryptanalysis based on parts of the key or allow for the introduction of faults or targeted modification of private keys. Insufficient checks in the import and export mechanisms may allow for the introduction of "Trojan" keys under the control of the attacker, possibly also as key wrapping keys. A key separation attack allows attackers to disassociate a key from its attribute policy. In weaker key attacks, a strong key is wrapped with a cryptographically weaker one. Bit-wise API operations on key material may allow for several other, typically brute-force, attacks.

Even for some commercial HSMs it has been shown that key material can be extracted [48] through the software interfaces concerning the creation, import and export of cryptographic keys. Important issues are the incompleteness of security policies and implementations lacking checks and verifications of *all* preconditions and policies at *all* times and in *all* API functions.

At a first glance, the problem appears to be similar to that of security protocol security, as APIs are often the building blocks of non-trivial protocols. Indeed, it would be convenient to use existing push-button tools to reason on the security of an API. Ideally, such a tool receives a simple-to-write model of the system under test and a number of security specifications (e.g. that secrets should remain confidential from a Dolev-Yao style attacker). The tool would then either prove the security or describe a possible attack (message trace). With protocols, as we have seen in Section 2.6.2, this can be achieved with good performance, since most cryptographic protocols are linear and restarted with every fresh session or run.

Unfortunately, this does not hold for APIs [145], as the following example will illustrate. A simple protocol performed through the API could be to 1) open a session, 2) create a key, 3) use it to encrypt a message and 4) close the session. From the protocol point of view, this is a linear process, where the outcome of one session does not influence any other session at any other time. However, consider for example that the API, much like that of the TPM, allows users to create key pairs which are stored on an external storage for later use[3]. While each session of a protocol creating a key will be similar, it will not be the same. Each newly created key will alter the global state of all players. An intruder could for instance encrypt a message under a key that was created in an earlier previous session. Thus each API call in the protocol will lead to a monotonic growth of the overall system state information. This *mutable global state* leads to a state-space explosion in the models under test and requires a more powerful abstraction that currently goes beyond the capabilities of common protocol analysis toolkits.

However, a first generation of tools that are able to model the mutable global state is approaching. In [117], Fröschle and Steel describe the analysis of a subset of the PKCS#11 [275] API considering a model that allows for an unbounded amount of fresh data (i.e. generated fresh keys). A special focus is on the design and verification of *attribute policies* for keys, which describe the type of operations a key may be used for. Accordingly, well-designed policies can be shown to be secure using the SAT-based security protocol model checker SATMC [23].

Arapinis et al. extend ProVerif to become StatVerif [18], which allows for the expression of stateful processes and translates them to clauses that can be subject to verification.

Mödersheim [227] introduces the AIF language as extension to the backend language of the AVISPA toolkits. It allows for the declaration not only of persistent, but also non-persistent states. AIF can be translated to Horn clauses and has been integrated with ProVerif and the SPASS theorem prover. Experiments show the applicability to API analysis.

5.8 Architecture

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

5.8.1 Virtual Security Module

We now describe the design of a cryptographic software module, which we outline visually in Figure 5.2.

Our TvSM shares many of the typical functionalities found in other cryptographic modules. Connections are managed in sessions and authentication protocols ensure that only authorized roles can perform critical operations. This

[3]Actually, the TPM has no control at all over the external key store [173].

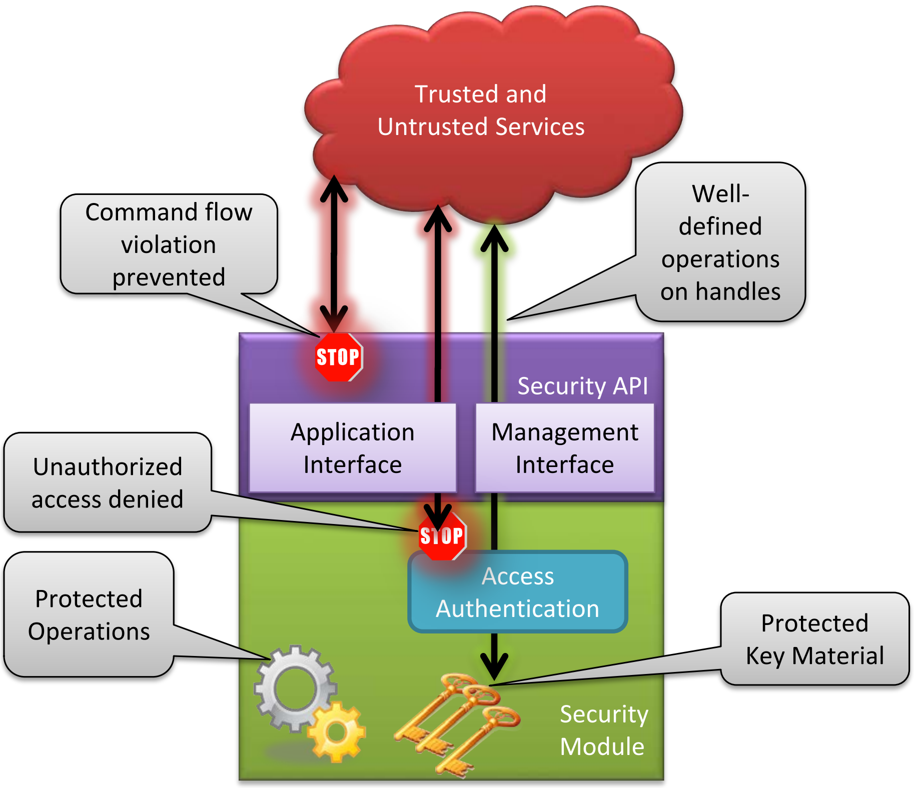


Figure 5.2: The TvSM Protects Keys Through a Security API.

especially applies on initialization, backup and handling of key material. RSA keys can be used to perform cryptographic signatures within the module, but only if their creation-time defined restrictions support these operation. Finally, the module supports the creation of key hierarchies which are rooted in a single, so-called master key. Other keys, or a hierarchy of storage keys, can be wrapped with it. This architecture allows for external storage, back-up of keys and scalability.

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

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

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

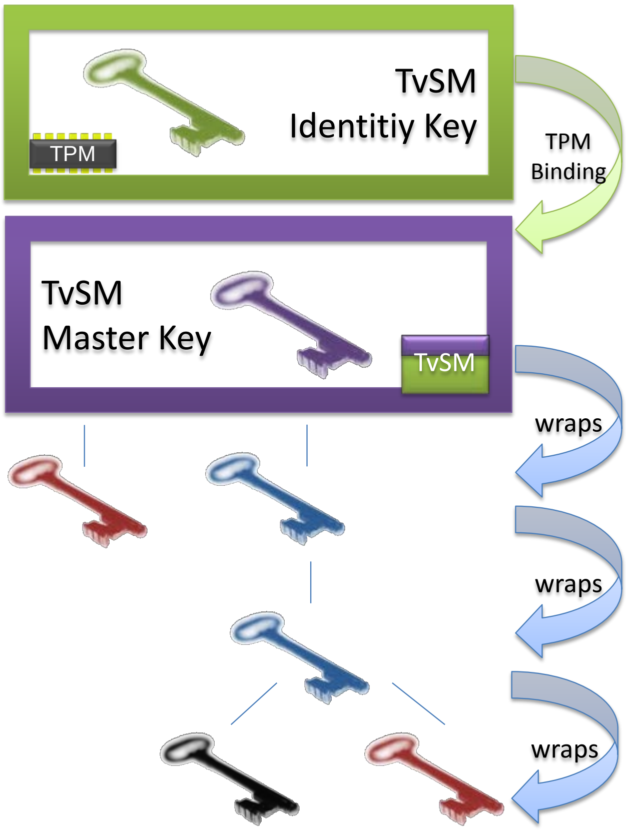


Figure 5.3: The Key Hierarchy.

We also root the key hierarchy of each instance of the TvSM in a unique TPM-key, which we refer to as identity key from now on. The overall key hierarchy structure is illustrated in Figure 5.3. We use the identity key to protect the master key, as well as for TvSM identification. Being non-migratable, the identity key identifies the TvSM uniquely. During setup we associate the TvSM instance to the identity key. The same key also protects the module's master key by binding it to the hardware platform. We choose the binding mechanism, as it does not enforce a trusted state for the key, which would preclude updates of any part of the chain-of-trust after the module is deployed. Instead, the lock to a trusted state is achieved by storing the key blob of the identity key in an encrypted file system that is sealed to a well defined state ($S_{service}$, which we will define in the next Section 5.8.2) but can be re-sealed to future trusted states.

Previous work by Berger et al. [42] on the virtualization of the TPM recommended a similar coupling of hardware-to-virtual module key hierarchies. The integration with our third-generation virtualization platform now allows us to leverage the strong TPM sealing to this end, even in the face of updates of the chain-of-trust.

5.8.2 Trusted Virtualization Platform Integration

The acTvSM platform [126,253,254,333] is a third generation (see Section 2.5.3) Trusted Computing architecture, which has been specifically designed by Martin Pirker and the author to host software security modules. It enforces integrity of the virtualization platform, the applications and services it hosts while applying strong hardware-based memory separation of partitions. The platform takes advantage of the Linux Kernel-based Virtual Machine (KVM) hypervisor, which is operated in a non-trivial configuration, which we call *Base System*. While we discuss the secure boot mechanism, the file system layout [333] and application management and update [126] processes in separate publications, we now show the integration with the security module, for which the platform was designed for in the first place.

An important novel aspect of our architecture is that the measurements are predictable so the PCR values can be calculated *a priori* and data can be sealed to a *future, trusted state* after a planned configuration update.

Based on PCR measurements, access to data can be restricted to a known-good platform state. This can be achieved using the TPM's ability to *seal* data. Remember, that sealing and unsealing means to encrypt — respectively decrypt — data inside the TPM with a key that is only available inside the TPM and where usage of the key is restricted to a defined PCR configuration. Under the assumption that the TPM is a tamper-resistant device this is a robust way to guarantee that sealed data can only be accessed under a predefined platform state.

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

Beginning from power on, the system performs a conventional boot and then calls `GETSEC[SENTER]`. After returning from that call, the system continues to transit between system *trusted* states we refer to as S_{boot}, S_{update}, $S_{application}$, S_{update} and $S_{service}$. The state S_{boot} is reached where ACM and MLE have been measured, the secure boot loader has already launched the kernel, and the code contained in the initial ram-disk executing. In S_{update}, the platform has measured the full code base of the base system and accepts maintenance operations[4] by the platform administrator, who needs to be in possession of the

[4]In [126] we describe an update process that does not require the system to enter an

TPM Owner authentication. $S_{application}$ differs from S_{update} by an arbitrary token that indicates that changes to the persistent state are prohibited and applications can now be offered. Finally, the chain-of-trust includes the security module service by a measurement of the module's software image before it is mounted. More formally,

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

This definition of the expected and trusted system state describes a integrity-guaranteed boot of the platform up to the security module service. Following the so defined steps, the acTvSM platform can compute the expected PCR values from the binary code and file systems installed and consequently seal data to. With it the TPM's sealing mechanism is used to restrict access to the file system access keys for the `security module image` to $S_{service}$ only. Thus we achieve integrity protection and enforcement at boot-time.

At runtime, however, the hardware assisted memory isolation of the chip-set provides runtime isolation. Note that TXT also enforces DMA restrictions and memory zeroization upon boot cycles. Thus the virtualization platform ensures the protected operation of the virtual security module.

5.9 API Design

This section describes the design of the TvSM security API by motivating design choices, presenting a notation and describing the API functions and key policies.

5.9.1 Considerations and Notation

The API has been designed to provide the necessary functionality while at the same time to allow us a rigorous reasoning about its security.

We decided against implementing a full PKCS#11 [275] interface. This would have increased implementation complexity, and would have precluded a concise, full-scope security analysis on the level of detail we desire.

Instead, we decided to create a new design that supports a single use case: the management and use of asymmetric keys for cryptographic signatures. This focus allows us to reduce the attack surface and to design the actually needed functionality with full formal rigor. A number of contributions influenced the

untrusted state.

design. For instance, we avoid arithmetic operations like `XOR` on API data structures [371], clearly group related API functionalities to avoid side effects, use static types for keys according to their use [117, 345], apply the (already) verified authentication protocol SKAP [66], store exported keys and their attributes together [48], and offer an easy to program, typed, and object-oriented interface definition.

To provide an abstract description of individual commands that together compose the API, we extend our syntax for cryptographic protocols. We use the following notation which is derived by the notation and semantics defined for API rules in [117] and which helps us describe the stateful behavior of APIs. Fröschle and Steel [117] define a rewriting system over a typed term algebra which also covers the notion of handles bindings of cryptographic keys through an API and also provide a formal semantic. We use our notation as rather more informal abstraction of the original API [335], which is defined in about 40 pages of JavaDoc. Still, it helps us discuss the elements of the interface in compact definitions and also simplifies the creation of the executable model in the analysis tool. In our notation:

A handle, which identifies an object behind the module's security boundary, is the binding of a nonce n_i to key K_j: $\mathrm{h}(n_i, K_j)$. Keyed integrity is denoted by $\mathrm{HMAC}_{K^{-1}}(m)$. An attribute a is element in attribute set $\mathbf{A}$ belonging to key K is written $a \in \mathbf{A}_K$. We write $\{a, b\} \subseteq \mathbf{A}_K$, if both attributes a, b hold for K, with $|\mathbf{A}_K| \geq 2$.

We describe the signature of API functions from the external Dolev-Yao [94] view, and augment this with parameter conditions checked internally and elements freshly created, *always* assuming successful authorization:

$$\texttt{function} : \textit{preconditions} \xrightarrow[\textit{fresh variables}]{\textit{conditions checked}} \textit{postconditions}.$$

Intuitively, on the left of the arrow are the input parameters of an API function, and the return values are on the right side. The function's body must check the conditions on top of the arrow, and create fresh data under the arrow. Only fresh data which is visible to the intruder is given, but no implementation details specified.

Slightly more formal, a function (or API rule, or API step) can only be triggered in our model if the *precondition* and the *conditions checked* both hold. Then the *postconditions* hold for the next *state* the model will transit to. Such functions may generate new objects, which will extend the state of the security module. An intruder will only learn the facts *input* and *output*, but not the internal state changes. The *conditions checked* is an addition introduced in our notation to allow a compact description of important security preconditions; in implementations, these checks can easily be coded as `IF-THEN-ELSE` statements in the beginning of the functions that implement the API. It is important to notice that for our abstraction we do not define all changes to the internal state - as the internal state is clearly implementation dependent. We further assume that API functions can be called in arbitrary order.

$$\texttt{createMasterKey} : \mathbf{A} \xrightarrow[\text{new } n_M, K_M, K_M^{-1}]{\mathbf{A} \supseteq \{\text{MST,STO}\}} \mathrm{h}(n_M, K_M) \tag{5.3}$$

$$\texttt{migrateMasterKey} : K_{\text{ID2}} \xrightarrow{\mathbf{A}_{K_M} \supseteq \{\text{MIG,MST}\}} \texttt{TPM\_BOUND\_DATA}_{K_{\text{ID2}}}(K_M, \mathbf{A}_{K_M}) \tag{5.4}$$

$$\texttt{importMasterKey} : \texttt{TPM\_BOUND\_DATA}_{K_{\text{ID2}}}(K_M, \mathbf{A}_{K_M}) \xrightarrow[\text{new } n_M]{\mathbf{A}_{K_M} \supseteq \{\text{MIG,MST}\}} \mathrm{h}(n_M, K_M) \tag{5.5}$$

$$\texttt{getMasterKey} : \rightarrow K_M \tag{5.6}$$

Figure 5.4: Abstract TvSM Maintenance API.

Cryptographic keys are typed with a set of attributes from the set $\{\text{MST}, \text{MIG}, \text{SIG}, \text{STO}, \text{EXT}\}$ and size and algorithm information for the respective key type descriptors master, migratable, signature, storage and exportable. Intuitively, the MST key attribute defines the master key, the root of the key hierarchy; storage keys (STO) keys may wrap other keys, and thus allow the creation of branches in the hierarchy; keys with the attribute MIG may transfer from one storage key to another (defined at creation time), even across TvSM instances. Only signature keys may sign messages. Finally, a key is flagged EXT, if it was created outside the TvSM. Attributes cannot be changed at any time after creation.

5.9.2 Abstract Presentation of the API

Overall, the API consists of two Java interfaces, `MaintenanceSession` and `OperatingSession`.The `MaintenanceSession` is concerned with initialization of a fresh module with a user-specific key hierarchy beneath the unique Identity Key pair K_{ID}. K_{ID} is created when installing the module's software image and non-migratably bound to the TPM. A Client in the `MaintenanceSession` uses K_{ID} in the SKAP protocol to authenticate the session.

Within the session, a singleton master key K_M can be freshly created calling API function 5.3. Here, the desired set of attributes $\mathbf{A}$ is passed on to the API method, and parsed to fulfill the necessary policy, i.e. a master key needs to carry the MST flag and be a storage (STO) key[5]. The method will then

[5]As we will later see in the discussion of Table 5.1, there is no security rule against having as MST key which is not STO in general, but it would be rather useless which is why we restrict it here in the generation method. This example also demonstrates how widely distributed

$$\texttt{createKey}: \mathrm{h}(n_1, K_1), \mathbf{A} \xrightarrow[\text{new } K_2, K_2^{-1}]{A_{K_1} \supseteq \{\mathrm{STO}\} \oplus \mathbf{A}_{K_1} \supseteq \{\mathrm{SIG}\}} \mathrm{KBLOB}(K_2, K_1) \tag{5.7}$$

$\texttt{loadKey}$:

$$\mathrm{KBLOB}(K_2, K_1), \mathrm{h}(n_1, K_1) \xrightarrow[\text{new } n_2]{\mathbf{A}_{K_1} \supseteq \{\mathrm{STO}\} \& \text{ HMAC valid}} \mathrm{h}(n_2, K_2) \tag{5.8}$$

$\texttt{migrateKey}$:

$$\mathrm{h}(n_1, K_1), \mathrm{h}(n_2, K_2) \xrightarrow{\mathbf{A}_{K2} \supseteq \{\mathrm{MIG}\} \& \mathbf{A}_{K_1} \supseteq \{\mathrm{STO}\}} \mathrm{KBLOB}(K_2, K_1) \tag{5.9}$$

$\texttt{importKey}$:

$$K_2, \mathrm{h}(n_1, K_1), \mathbf{A}_{K_2} \xrightarrow[\text{new } n_2]{\mathbf{A}_{K_1} \supseteq \{\mathrm{STO}\} \& \mathbf{A}_{K_2} = \{\mathrm{SIG,EXT}\}} \mathrm{KBLOB}(K_2, K_1) \tag{5.10}$$

$$\textit{texttexportKey}: \mathrm{h}(n, K) \xrightarrow{\mathbf{A}_K \supseteq \{\mathrm{EXT}\}} K, K^{-1} \tag{5.11}$$

$$\texttt{sign}: m, \mathrm{h}(n, K) \xrightarrow{\mathbf{A}_K \supseteq \{\mathrm{SIG}\}} \mathbf{S}_{K^{-1}}(m) \tag{5.12}$$

$$\texttt{getMasterKeyHandle}: \rightarrow \mathrm{h}(n_M, K_M) \tag{5.13}$$

$$\texttt{unloadKey}: \mathrm{h}(n, K) \rightarrow \perp \tag{5.14}$$

Figure 5.5: Abstract TvSM Operating API.

generate a fresh key pair, assigns the attribute set to it, store and use the new private master key K_M^{-1} internally. Finally a nonce n_M is generated, stored as handle value $h(n_M, K_M)$ and returned to the caller.

If a master key is created with attribute MIG set and if the necessary authorization is provided, it may be migrated (5.4) to another identity key K_{ID2} by using the TPM binding mechanism. Inversely, it can be imported (5.5) to allow for scalability by having several parallel instances of the TvSM with the same key hierarchy. This scheme can also serve to establish a back-up process. K_M is world readable (5.6) and a TvSM instance can also be instructed to delete the hierarchy root key.

Once a master key is established, it can be used to authenticate an `OperatingSession`. Here, the master key is accessible by its handle (5.13). The key hierarchy is built by creating (5.7) a fresh key pair (K_2, K_2^{-1}) under an existing storage key K_1 or K_M. The return value of this creation process is a data structure that holds the private key and the policy assigned to it.

security policy-relevant aspects often are in security APIs.

Confidentiality is protected by encryption with parent key K_1 and integrity by incorporating a keyed cryptographic hash sum:

$$\begin{aligned}\mathrm{KBLOB}(K_2, K_1) = \{K_1.K_2.\mathbf{E}_{K_1}(K_2^{-1}).\mathbf{E}_{K_1}(\mathrm{AUTH}_{K_2}).\mathbf{A}_{K_2}.\\ \mathrm{HMAC}_{K_1^{-1}}(K_2 \| K_2^{-1} \| \,\mathrm{AUTH}_{K_2} \| \mathbf{A}_{K_2} \| ..)\}\end{aligned} \tag{5.15}$$

The TvSM is actually oblivious to the number of keys it can protect by leaving the actual storage location of key blobs at the discretion of its clients. Thus keys need to be loaded into a key slot before they can be used. Keys can be loaded (5.8) only, if the parent key has been previously loaded for decryption, as this operation first unwraps the key blob. Slots may also be freed (5.14). Thus a hierarchy of keys is created. Application keys can be migrated (5.9) to other storage keys, if the attribute MIG and a migration authorization secret was set at creation. Migration essentially means the re-wrapping of a key previously wrapped under K_1 to the new parent K_2. A different mechanisms is the import (5.10) of externally created keys. This allows the TvSM to hold legacy keys and enforce that they are only used if the correct credentials, i.e. authentication passwords are provided. Since the private part of the key has been created externally, this has to be noted in the attribute set. Only for this type of key, an export of the private key is allowed (5.11). Finally, SIG keys can perform RSA signatures (5.12).

We consider the selection of key policies as the most important part of the design of an API, because most practical attacks have been found in this context. We therefore choose the following security rules 5.16, 5.17 and 5.18 for the key policies, which supplement the abstract API operations given in figures 5.4 and 5.5.

$$\text{No MST key is EXT.} \tag{5.16}$$

$$\text{STO keys are never EXT.} \tag{5.17}$$

$$\text{A key must not have the attributes SIG and STO at the same time.} \tag{5.18}$$

From these statements we derive the truth table presented in Table 5.1 of valid key policies, which are the only ones allowed to be created within the TvSM. Note that these valid policies are not sensible or secure per se, but must be considered together with the full API definition.

5.10 API Model and Practical Verification

In this section we first describe the tools and methods used to analyze the API and then describe the executable model in detail, at source code level.

5.10.1 The Verification Tool

We have verified the key security policies of the TvSM against a formal model of the API presented above. We created a machine-readable model based on

Key Policies									
#	Key Attributes					Rules			Valid
	MST	MIG	SIG	STO	EXT	(5.16)	(5.17)	(5.18)	
0	1	1	1	1	1	0	0	0	**0**
1	0	1	1	1	1	1	0	0	**0**
2	1	0	1	1	1	0	0	0	**0**
3	0	0	1	1	1	1	0	0	**0**
4	1	1	0	1	1	0	0	1	**0**
5	0	1	0	1	1	1	0	1	**0**
6	1	0	0	1	1	0	0	1	**0**
7	0	0	0	1	1	1	0	1	**0**
8	1	1	1	0	1	0	1	1	**0**
9	0	1	1	0	1	1	1	1	**1**
10	1	0	1	0	1	0	1	1	**0**
11	0	0	1	0	1	1	1	1	**1**
12	1	1	0	0	1	0	1	1	**0**
13	0	1	0	0	1	1	1	1	**1**
14	1	0	0	0	1	0	1	1	**0**
15	0	0	0	0	1	1	1	1	**1**
16	1	1	1	1	0	1	1	0	**0**
17	0	1	1	1	0	1	1	0	**0**
18	1	0	1	1	0	1	1	0	**0**
19	0	0	1	1	0	1	1	0	**0**
20	1	1	0	1	0	1	1	1	**1**
21	0	1	0	1	0	1	1	1	**1**
22	1	0	0	1	0	1	1	1	**1**
23	0	0	0	1	0	1	1	1	**1**
24	1	1	1	0	0	1	1	1	**1**
25	0	1	1	0	0	1	1	1	**1**
26	1	0	1	0	0	1	1	1	**1**
27	0	0	1	0	0	1	1	1	**1**
28	1	1	0	0	0	1	1	1	**1**
29	0	1	0	0	0	1	1	1	**1**
30	1	0	0	0	0	1	1	1	**1**
31	0	0	0	0	0	1	1	1	**1**

Table 5.1: Valid Key Policies in the TvSM.

the API analysis method presented by [117] and which was later refined for the results of Bortolozzo et al. [48] on commercial HSMs.

For our TvSM, the behavior of the API is modeled in the AVISPA intermediate format (IF), a language originally designed as machine-generated input format to the different back-ends of the AVISPA [28] protocol analysis push-button tool kit. Of the different AVISPA back-ends, we use SATMC [23] as engine for our experiments. SATMC decides the security of a model against specifications by converting it through multi-set rewriting, lineralization and various logical optimizations into a finite satisfiability problem which is then fed into a SAT solver. In our case, the MINISAT [100] solver is used for this task. As with other model checkers, SATMC will either report a concrete attack trace or conclude that no attack could be found. SATMC is used because it can consider different system states in its analysis, a feature not available in most other protocol-oriented verification tools. However, this capability of SATMC is not available through the user-friendly AVISPA High Level Protocol Specification Language, which extends the Alice and Bob notation.

5.10.2 The Executable Model

The AVISPA Intermediate Format (IF) model of the TvSM API is structured in several sections, which we will now review. It is worth discussing the *source code* [322] of our model in some detail, as the input and output language was not designed for human programmers; therefore interpreting the results requires some background knowledge and also helps understanding the applicability of the approach.

Sections Signature and Type The initial section Signature describes functions that are specific to this model. Here, a function `state_generation()` is declared that represents the transition from one system model state to another, based on sets of input variables. The Type section declares variables' and constants (without concrete value). For understanding the source code, it is important to note that variables' first letters are always in upper case, while constants start with a lower case. The basic and abstract data types used are either defined in the IF language definition or in a prelude file. For our model, we define two agents the intruder `i` and the security module `theVSM`, key pairs `kp1, kp2, kpi, Kp1` etc., nonces `n0,n1,n2,`..., key attributes and various helping elements such as sets that store the state information.

Section Inits Here, constants are assigned to variables to define the initial (starting) state of the model. The SATMC engine will later perform set rewriting of these terms in the course of automatic analysis. Note that the dot `.` represents the logical AND operation.

Initially, the intruder knows its public and also its private key (denoted as the inverse `inv()` of the public key). It also knows the public key of `migrationkey` which serves as abstraction of all keys that may be specified as keys to migrate

TvSM keys to. Therefore, we assume that keys which are specified as migration keys at key creation are trusted and not accessible to the intruder by definining in the AVISPA IF language:

```
initial_state init1 :=
iknows(kpi).
iknows(inv(kpi)).
iknows(migrationkey).
```

The following commented-out line would allow us to test what happens if the intruder had learned this key by an out-of-band/model attack.

```
%iknows(inv(migrationkey)).
```

We now define which elements are under the protection of the TvSM, i.e. the goal is that they are not accessed by the intruder at any time.

```
contains(theVMS, protected_data).
secret(inv(kp1),new_asym_Key,protected_data).
```

Finally, we define the initial state of the module by assigning the concrete instances to the variable definitions in the signature of the state generation function.

```
state_generation(theVSM,keyAttributes,handleMapping,
                 privatekeys,pubkeys,nonces,pnonces,0,12)
```

Section Rules This code section defines all state transitions specific to our model. For example, a generic transition would be defined as:

```
step step_name (Variables) :=
  state_generation(Variables).
  preconditions.
=[optional fresh Variables]=>
  state_generation(Variables).
  preconditions.
  new facts
```

Every step (a rule with a given name) for a set of variables has to fulfill the preconditions before it can be triggered. The preconditions are defined as predicates such as `contains(element, set)` or `iknow(...)` which represents the intruder knowledge, i.e. all information that is transmitted in clear through the API. To create a stateful model of our module, we always require the previously declared `state_generation()` with its sets of data. Only if all preconditions are met, the step may be performed. The transition may include the creation of a fresh variable; infinitely many fresh variable instances may be created. The preconditions should always be included in the results part of the statement so that existing information is copied to the new state. The effect of the step is then again expressed as concatenation of predicates.

The most important task of the TvSM is to manage the key life-cycle correctly. As [48, 86, 117] have shown for PKCS#11, a wrong configuration of key policies can directly lead to practical attacks. Therefore, key policies must be permanently linked to the RSA key pairs. To this end, the TvSM attaches an integrity-protected attribute policy to the swapped KBLOB (5.15). The module implementation must then check the integrity with every swap-in (i.e. `loadKey()` (5.8)) operation. However, for our model, this level of detail is not needed; instead we just store the policy for each public key in the set `KeyAttributes`.

```
step step_generate_asym_Key_TFFTF (VSM,KeyAttributes,
                                   HandleMapping,PrivateKeys,
                                   PubKeys,Nonces,Pnonces,Kp1,
                                   N1,SID)
:=
  state_generation(VSM,..).
  contains(N1,Pnonces).
  contains(Kp1,PubKeys).
  contains(pair(pair(storage,true),Kp1),KeyAttributes).
  iknows(hand(N1,inv(Kp1)))
=[exists Kp2]=>
  state_generation(VSM,..).
  contains(Kp1,PubKeys).
  contains(pair(pair(storage,true),Kp1),KeyAttributes).
  iknows(hand(N1,inv(Kp1))).
  secret(inv(Kp2),new_asym_Key,protected_data).
  contains(pair(pair(storage,true),Kp2),KeyAttributes).
  contains(pair(pair(signing,false),Kp2),KeyAttributes).
  contains(pair(pair(master,false),Kp2),KeyAttributes).
  contains(pair(pair(migratable,true),Kp2),KeyAttributes).
  contains(pair(pair(exportable,false),Kp2),KeyAttributes).
  contains(Kp2,PubKeys).
  contains(inv(Kp2),PrivateKeys).
  iknows(crypt(Kp1,inv(Kp2))).
  iknows(Kp2)
```

Figure 5.6: Sample Key Generation Rule.

As previously explained, there are, according to the TvSM specifications, 16 valid combinations of key attribute policies. The allowed key policies are used to define the key generation functions in the model so that we create only valid policies. For each other operation in the API, the appropriate pre-conditions with regard to these key policies are modeled. We will now present the implementation of one of the key generation rules (Figure 5.6). Remember that every key (except for Identity and MasterKey) must have a storage key as parent. This precondition is modeled by requiring that parent key `Kp1` has been created by the TvSM (N1, Kp1 are known), so that there exists a handle and also that Kp1 has been assigned the policy attribute STO. If and only if so, a fresh key pair Kp2 is created. For this fresh key, Kp2 is stored in the list of created keys, and a policy is defined and stored in the model state. Furthermore, the private part of the key is declared as secret (i.e. a potential goal for the attacker). Finally, the intruder learns the public part (for simplicity we also include the public key getter functionality here), and the private part encrypted under the parent. Together with the globally stored policy, this last piece of knowledge represents the KBLOB.

Let us consider another example for a model of a typical and critical function of the TvSM. In Figure 5.7, the export key function (5.11) is given. It is straightforward to read: the rule is only fired for the given input, if the following facts are true: The key Kp1 must be loaded and a handle available. Its attribute policy must ensure that the EXT flag was set at key creation, respectively key import. If these preconditions hold, the existing state is copied and the action performed: releasing the private key inv(Kp1) over the network channel. Note that checking the policy in common programming languages can typically be achieved by a simple IF-THEN-ELSE construct using Boolean statements which are trivially analogous to the statements encoded in the model.

```
step step_exportKey(VSM,..):=
  state_generation(VSM,..).
  contains(pair(hand(N1,inv(Kp1)),Kp1),HandleMapping).
  contains(pair(pair(exportable,true),Kp1),KeyAttributes).
  iknows(hand(N1,inv(Kp1)))
=>
  state_generation(VSM,..).
  contains(pair(hand(N1,inv(Kp1)),Kp1),HandleMapping).
  contains(pair(pair(exportable,true),Kp1),KeyAttributes).
  iknows(hand(N1,inv(Kp1))).
  iknows(inv(Kp1))
```

Figure 5.7: Model Rules for Key Export.

Section Properties and Attack States As with any other model checker tool, the model is verified against a set of specifications. In the case of a security API, the specification should be security properties regarding the handling of private keys.

In a formalization, these security properties are the goals of an attack on the API and describe what knowledge a Dolev-Yao intruder [94] will attempt to gain. Again, we assume that the underlying cryptographic primitives are perfect and that keys and message fields are atomic. The intruder can encrypt and decrypt, given that she knows the necessary keys. We further assume that the intruder starts with her public/private key pair and can easily attain a handle to the

Master-key through the API. When a fresh key pair is created, all parties learn its public key. The private key however should stay confidential. For SATMC this can be expressed by declaring new private keys as `secret` and defining that the attacker must not learn them. Thus, it is the attacker's goal to learn one of the secret elements. Our security specification becomes:

$$\text{The intruder must not learn any further private keys.} \tag{5.19}$$

This is achieved in the model by declaring that a successful attack state has been found, if the attacker learns a private key that has been declared as `secret` based on the property `secrecy_of_new_asym_Key`.

5.11 Validation of Method and Verification Results

We will now illustrate the verification undertaken with two example results from our experiments. To this end we introduce subtle errors in the API specification and observe if the model checker will detect attacks.

5.11.1 Example Attacks

In this section we introduce two deliberate mistakes in the API and discuss the results and give performance data on the analysis.

Example Attack 1 Let us assume that due to some oversight in implementing the security policy of the module the following mistake has been introduced: The export key function (5.11), when it is applied to a key handle, does not check whether the key policy of the key has the EXT flag set. This is the equivalent of removing line 4 of Figure 5.7. The SATMC tool automatically detects this case and provides an easy-to-read trace containing the API calls so that we can observe in Figure 5.8 how an attack could be performed. In the beginning a master key must be present; it is created (5.3) as `kp10` and a handle to it requested (5.13). In the next step, some other key K (i.e. `kp20`) with its security policy configuration is created (5.7) under K_M. Note that SATMC chooses one of several possible configurations; for this example it only matters that K is not exportable, i.e. EXT set to `false`. Next in the trace, this key is loaded (5.8). Finally, the compromised operation of exporting a key (5.11) can be called on K. This should never have occurred, as the key is not EXT. This last step breaks the security goal, as the intruder learns the private key. The attack trace is found by SatMC 3.4 in just 0.57 [s] on a HP DC7800 PC with a 2.33 [GHz] Intel E6550 CPU running Ubuntu Linux 12.04 LTS.

When we insert the missing check of the EXT attribute in the such corrected model, no attacks are found.

```
GOALS:   [secrecy_of_new_asym_Key(inv(pk(fpk(kp20,mr(theVSM),12))),
          protected_data)]

Step 0:  [sc_step_createMasterKey_1(theVSM,keyAttributes,
          handleMapping,privatekeys,pubkeys,nonces,pnonces,12)]
Step 1:  [sc_step_getMasterKeyHandle_1(theVSM,keyAttributes,
          handleMapping,privatekeys,pubkeys,nonces,pnonces,
          kp10(theVSM,12),12)]
Step 2:  [sc_step_generate_asym_Key_FFFFF_1(theVSM,keyAttributes,
          handleMapping,privatekeys,pubkeys,nonces,pnonces,
          kp10(theVSM,12),n10(theVSM,12),12)]
Step 3:  [sc_step_loadKey_1(theVSM,keyAttributes,handleMapping,
          privatekeys,pubkeys,nonces,pnonces,kp10(theVSM,12),
          kp20(theVSM,12),12)]
Step 4:  [sc_step_exportKey_1(theVSM,keyAttributes,handleMapping,
          privatekeys,pubkeys,nonces,pnonces,n20(theVSM,12),
          kp20(theVSM,12),12)]
```

Figure 5.8: API Attack Example 1.

Example Attack 2 As a second example, consider a wrong definition of the key import functionality (5.10) allowing for an illegal key policy:

`importKey'` :

$$K_2, \mathrm{h}(n_1, K_1), \mathbf{A}_{K_2} \xrightarrow[\text{new } n_2]{\mathbf{A}_{K_1} \supseteq \{\text{STO}\} \& \mathbf{A}_{K_2} \supseteq \{\text{SIG,EXT}\}} \text{KBLOB}(K_2, K_1)$$

Here, an attribute set $\mathbf{A}_{K_2} = \{SIG, EXT, STO\} \supseteq \{\text{SIG, EXT}\}$ is valid, thus allowing for the usage of an imported key as storage key. The effect of this allowed key policy is illustrated in Figure 5.9. Again a master key is created (5.3) as `kp10` and a handle to it requested (5.13). The intruder then imports 5.10 a key pair `kpi` as storage key beneath the master key (`pk10`). Now, the imported key can be loaded (5.8) as any other key and used to wrap some other freshly generated key (`kp20`). As the attacker knows the private key `kpi`, he can, under the Dolev-Yao assumptions, unwrap the key and learn the private `kp20`. This clearly breaks the intended security goal, namely that keys protected through the TvSM must not be accessible to an attacker. This attack trace takes marginally longer to find, namely 0.72 [s] on the some configuration as above. With the definition for as provided in the original `importKey` (see Equation 5.10), no attack is found.

Exploiting this weakness in practice might not be straightforward, but potentially very harmful. One possible approach would be to uproot future key hierarchies by confusing the key handles or KBLOBS within an application. As an application that accesses the TvSM cannot expected to be of TPM-enforced binary integrity, it could be manipulated persistently while apparently still creating seemingly secure and well-protected keys using the TvSM.

```
GOALS: [secrecy_of_new_asym_Key(inv(pk(fpk(kp20,mr(theVSM),12))),
        protected_data)]
Step 0:   [sc_step_createMasterKey_1(theVSM,keyAttributes,
           handleMapping,privatekeys,pubkeys,nonces,pnonces,12)]
Step 1:   [sc_step_getMasterKeyHandle_1(theVSM,keyAttributes,
           handleMapping,privatekeys,pubkeys,nonces,pnonces,
           kp10(theVSM,12),12)]
Step 2:   [sc_step_importKey_1(theVSM,keyAttributes,
           handleMapping,privatekeys,pubkeys,nonces,pnonces,
           n10(theVSM,12),kp10(theVSM,12),kpi,12),

           sc_step_loadKey_2(theVSM,keyAttributes,handleMapping,
           privatekeys,pubkeys,nonces,pnonces,kp10(theVSM,12),
           kpi,12)]
Step 3:   [sc_step_generate_asym_Key_FFFFF_1(theVSM,keyAttributes,
           handleMapping,privatekeys,pubkeys,nonces,pnonces,kpi,
           n20(theVSM,12),12)]
Step 4:   [decrypt_public_key_2(kpi,kp20(theVSM,12))]
```

Figure 5.9: API Attack Example 2.

5.11.2 Conclusions from Analysis and Validation

For the TvSM model we presented, without deliberately introduced errors, no attack traces can be found given the described method. Note that the model checker tests all possible command flows under the restrictions of the model. Therefore, for the presented API this *verifies*, under those restrictions, that the API is correct and secure.

Attack traces are easy to ready and clearly show malicious command flows. It therefore becomes easy to single out the elements of the API that need to be fixed in oder to prevent each attack. We especially note that the structure of the model is very close to imperative programming languages like C or Java. The model therefore clearly indicates which security checks, especially on key policies, need to be made in witch function of the API, both in the model and in possible implementations. The model can therefore serve as direct guideline for implementors of the API definition. This is a significant improvement, since security modules are notoriously hard to implement as security relevant operations are typically widely scattered in the source code.

We conclude that our approach can be used to analyze the key policy of the TvSM and that it also allows us to verify that the *right checks* are made and even that they are made *in the right places* of the code. These results can therefore be directly incorporated in the API specification and the TvSM implementation.

5.12 Implementation

5.12.1 Software Design

The general architecture of the TvSM prototypical implementation and its integration in the acTvSM virtualization platform (see Section 5.8.2) is illustrated in Figure 5.10.

On the left hand side, we see the example of a Java-based client application to the TvSM. In the acTvSM research project, this application has been the commercial XiTrust Business Server solution[6] that provides advanced signature services in the sense of EU Directive 1999/93/EC for electronic processes in large enterprises. This business application can access the the TvSM through the a `KeyStore` interface, which builds upon the Jave JCE and connects to different proprietary security modules, as for instance the TvSM. A client application can use the TvSM API through the TvSM client library which hides the communication overhead. In general, a client may run on the same platform but confined in a different compartment or on a remote host platform.

The TvSM server on the right hand side of the figure is a boot-able binary image which includes the core executable, libraries, a Java Virtual Machine (JVM) and a Linux host operating system. The image can be loaded into an isolated compartment on the acTvSM platform. Compartments are installed, managed and updated through the acTvSM libraries and tools. The platform provides the TvSM with secure boot, resource isolation, off-line image encryption and TPM forwarding. It is based on Debian Linux with KVM and several specialized Trusted Computing libraries, such as IAIK jTSS and jTT (see Section 3.2.2). The DRTM late launch is performed at start-up as described in Section 5.8.2 and enables a secure boot that insures software integrity. At runtime, the Intel-VT mechanism is employed by KVM ensuring isolation of memory areas. The API of our TvSM is accessible via the network interface using the Java Remote Method Invocation protocol.

5.12.2 Performance and Results

In this section, we briefly summarize the performance results reported by Reimair in his Master's thesis [268] on the fully functional TvSM prototype. The TvSM server as well as the client application benchmarks ran on a TXT-compatible Intel VPro platform with a Intel Core 2 Duo P9500 CPU at 2.54 [GHz], with Linux kernel 2.6.35-22 in 64-bit mode and Java JRE 1.6.0.22. Cryptographic operations are performed by the Cryptography Provider of the SIC Crypto Toolkit [155].

The results for the TvSM are listed in the table in Table 5.2 and measured from a Java environment, thus include all software initialization and communication overheads, with means calculated over 100 repetitions. The session opening in the SKAP protocol takes about 750 [ms], including a random number generation procedure for session secret creation. Creating a 1024-bit RSA key pair takes a mean time of about 60 [ms]. It takes about 15 [ms] to load a key into

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

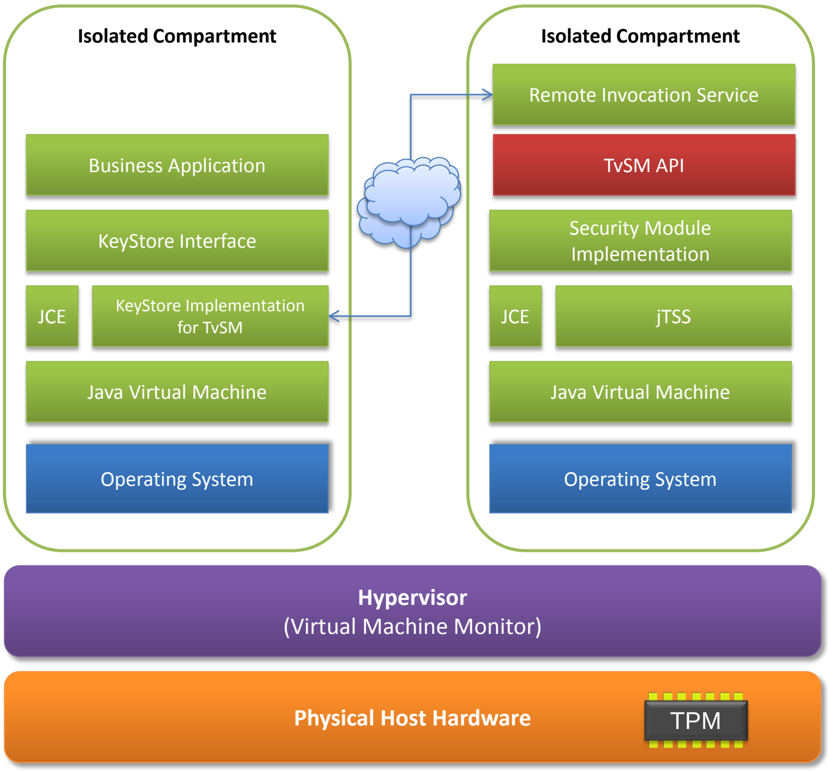


Figure 5.10: Outline of the TvSM Implementation.

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

To put this in context with the performance that the built-in HSM of the hardware platform, the TPM, is able to offer we took a number of measurements on the actual performance of different TPM implementations. Again we include all overhead introduced from a Java environment and use the jTSS library for TPM access. We give a brief comparison of a selection of available TPMs in the table in Table 5.3. A typical TPM chip is capable of creating about one key in three seconds and calculating about four to five signatures per second. However, the hardware random number generators limit the number of fresh keys that can be created as entropy needs to be gathered, literally, over time. Creating a 1024-bit key can take up to 15 [s]. The mean durations for this test series are calculated over 15 repetitions.

In single threaded operation, our TvSM prototype implementation is capable of creating continuously about 17 1024-bit RSA keys per second and calculating about 150 signatures per second; in the signature creation task it equals about

TvSM Operation	Duration [ms]		
	Minimum	Maximum	Mean
Session initialization	708	814	766
RSA Key Generation	21	378	62
Key Loading	12	21	13
Signing	6	9	7
Close Session	1	11	2

Table 5.2: TvSM Implementation Performance.

TPM Operation		Duration [ms]		
		Key Generation		Signature Creation
Manufacturer	Version	Mean	Max	Mean
Atmel	v1.2.13.9	2756	6700	145
Broadcom	v1.2.6.77	1698	3538	300
Infineon	v1.2.3.16	3342	7773	200
Intel	v1.2.5.2	4564	15058	158
STM	v1.2.8.16	2228	8308	288

Table 5.3: TPM Performance on RSA 1024 bit Keys.

30 times the throughput of a TPM, while in the key creation task this equals an speedup of 50 on average and up to 700 in extremis. A higher speedup can be expected with parallel sessions on multiple CPU cores. Reimair has compared [268] the reported performance results to several available and industrial HSMs. The processing speed of our TvSM prototype seems to be in league with medium-sized HSMs, which typically cost around €5000-€15000 per piece.

5.12.3 Security Discussion

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

As discussed in Section 2.7, some selected TPM implementations have been based on resilient SmartCard architectures and provide a relatively robust security against all but the most sophisticated hardware attacks. Yet, our platform cannot store all critical data in the TPM at all times, except for the Identity Key. All other data and key material is processed on the host platform and stored on the hard-drive, albeit in encrypted file systems.

We leverage TPM and TXT to provide a certain level of *modification resistance*, required by many applications and also by the FIPS 140 Security Level 2 standard [235], for the software executed. Here, we are restricted to the security level that can be achieved by these chip-set features. Commodity devices can only be assumed to protect against very simple hardware attacks [133]. Hardware security therefore depends on physical access control and diligent operational procedures in data-centers and other deployment scenarios. We need to caution that not much effort, i.e. less than 100 €, is needed to break the security guarantees of TXT if an attacker has physical access [362].

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

5.13 Conclusions on the TvSM

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

This Trusted virtual Security Module offers improved software attack resilience when compared to software key stores and better performance when compared to the TPM. Thus, while our approach does not achieve the same high level security as dedicated, tamper-resilient hardware modules, it offers an attractive cost-security-performance balance. Note that similar, specialized modules could in parallel implement other, totally isolated sets of functionality on the same platform.

5.14 Summary

In this section, we studied two challenges in practically applying formal methods to Trusted Computing scenarios.

First, we analyzed a TPM-based security protocol and where able to prove a potential vulnerability, i.e. the delivery of unauthenticated data to a client. As no convenient formal software tool was available at the time, we used a general purpose model checker to express the different behavior of the protocol in case that a client is in a trusted state or not.

In the second part, we presented the creation of a Trusted virtual Security Module. It is based on a virtualization platform that was specifically designed to protect a software security module. The software security module is designed to avoid logical API flaws that are a present danger in many current commercial products. Together this demonstrates a significant reduction of the attack surface and thus suggests a higher attack tolerance than conventional software security modules.

6 Conclusions

6.1 A Look Back

This thesis set out to study the interfaces found in Trusted Computing platforms based on the Trusted Platform Module (TPM). Trusted Computing is an influential attempt by key members of the IT industry to improve system security and offering assurance that a computing platform may be trusted for a specific purpose. As a technology focused on operating system security, there has been no initial support for managed software environments, few considerations of how to communicate results securely to users and a large gap between code identity and promised behavior.

This study has sought to improve different kind of interfaces, directed at different parties: application programmers, users and the network. The first result is a programming interface for the TPM, which is suitable for the Java language. Secondly, we proposed a wireless communication channel for the TPM that allows users a more direct interaction. Finally, we studied how verifiable correct network interfaces for services can be designed, either for cryptographic protocols that include a TPM or for purpose-built security APIs in general.

We will now present a synthesis of our main contributions, propose future research directions and finally formulate a conclusion.

6.2 Contribution

The main findings are specific to Chapters 3, 4 and 5 and we now summarize and synthesize them, starting, however, with Chapter 2.

In the background provided through Chapter 2, we began with a short history of computer security to see that perfect security is hard to achieve, but also that, for many purposes, computing platforms need to be trusted. Assurance can be provided by the Trusted Platform Module, which is a compact, specialized, but affordable Hardware Security Module. The TPM relies on a set of system software, the TCG Software Stack. Together with a platform's root of trust for measurement, the TPM can be used to protect private keys, to collect measurements of the software configuration, and to attest of doing so according to its specifications to other hosts. However, for robust measurements hardware virtualization and a Dynamic Root of Trust for Measurement (DRTM) should be employed. Cryptographic protocols enable the communication between hosts to reach defined security goals. Unfortunately, protocol design is a challenging task; a fact which has caused automated tools, based on formal methods, to emerge. We have noted, that protocol analysis tools are often restricted in their abilities, especially when including non-trivial, or stateful security mechanisms as offered by the TPM. Also, trusted platforms are in many cases vulnerable to hardware attacks.

The first main contribution is presented in Chapter 3. It is JSR 321, an API targeted at developers who wish to use the TPM in their Java applications. Java is one of the most important development environments, and used in mobile devices as well as in data-centers. Several proposals for TPM-libraries have been made, but previously to our design, no standardization was attempted. Our design is based on an analysis of existing proposals, defined goals and clear assumptions on the expected developers. It can be implemented in different software architectures as is shown through a reference implementation with accompanying test suite. Besides achieving the design and the implementation, we took an innovative, collaborative approach on how to realize the standardization in the Java Community Process, which was awarded with the "Outstanding Spec-Lead" award. The open design facilitates even skeptical developers to build persistent trust in the specifications. Also, the API has been tried and tested by industrial and academic reviews, a third party implementation, teaching, and experiments with the Cloud and mobile systems. The demonstration of JSR 321 on an Android platform with a TrustZone-protected TPM emulator has been rewarded the "best paper award" at INTRUST 2011.

Chapter 4 presents our second contribution, the proposal of a NFC-interface for the TPM. The challenge we respond to with this proposal is to inform the user of the security state of a public computing platform she faces physically. We believe a conventional smart phone to be a user-friendly way to perform such an attestation. There has been a number of other proposals to achieve this, but we argue that our choice of interface would be better suited to achieve resistance against platform-in-the-middle attacks. We propose to include an NFC interface in the TPM; the characteristics of the electromagnetic near field inherently reduce the working range of this radio scheme, thus implicitly assuring the proximity of the attesting device. We describe the necessary changes to the TPM's software API, and combine this with a protocol for remote attestation. In our scheme, the decision on an attestation report is delegated to a third party of the user's choice. Experiments on implementing the public computing platform, the mobile phone application which serves as attestation token, and the performance of a physical NFC interface support the viability our proposal. Furthermore, a number of variations can be applied to the scheme. More compact cryptographic protocols and DRTM-based platforms lead to potential optimizations, which have been recognized with a "best paper" award at ICSNC 2010. The chapter also outlines further related results on local attestation using a specialized token and outlines platform security for mobile devices.

The third main contribution is presented in Chapter 5. The motivation is to design the interfaces of a service in such a way that no malicious behavior can be triggered remotely. Here, we first present an analysis of a TPM-based cryptographic protocol. For the analysis we use a general purpose model checker, as more specialized tools do not support the TPM's API. Through a formal model and a set of specifications, we can prove the existence of a potential security issue, the lack of sender authentication in a secrets distribution system. An improvement is proposed and shown to be secure in the model. Thus, we have shown how a protocol can be analyzed that contains complex mechanisms from a security module. The second contribution on service behavior is the design of a virtual security module. We introduce two novelties: the co-design with a DRTM-based platform and the verification of the full API. For the verification, we extended on recent advanced in API analysis to cover the complete security API in our model for the SATMC model checker. We validate the verification by introducing bugs in our design which are promptly uncovered by the tool. Our trusted virtual security module solution provides effective, fully controlled integrity protection, runtime isolation and strong assurance on the secure handling of key material. Implementation and benchmarks results suggest that this contribution fills a gap in today's available solutions for security modules.

6.3 A Look Forward

In systems as complex and quickly developing like today's computing platforms, countless research directions can be suggested. Still, starting from the results of this thesis, a number of future challenges can be identified.

First, new hardware mechanisms for Trusted Computing are appearing. For instance, TPM version 2.0 will come with much more powerful policies that can be assigned to keys and other objects; this needs to be made available in protocols, or APIs and also warrants a scrutinizing analysis.

Second, we have assumed that NFC implies the immediate, physical proximity, while noting that relay attacks might be possible with some effort. Amongst other possible approaches, an effective and efficient distance bounding protocol for NFC-enabled TPMs would prevent that.

Third, the features of high-level Trusted Computing APIs would also be useful on systems, where no actual hardware TPM is available, for instance in mobile phones. Implementing JSR 321 in a TrustZone environment with a TPM emulator was only the first step in this direction; more powerful and flexible mechanisms can be imagined with TrustZone. Besides mobile systems, also embedded system developers can profit from high level security APIs[1].

Fourth, for the automated analysis of security APIs, more powerful and easier to use tools and the according theories are needed. An interesting starting point would be a common language for the description of security APIs, much in the way the "Alice and Bob" notation is commonly understood in the security protocol community.

6.4 Conclusions

This thesis has studied and improved interfaces in Trusted Computing. The contributions include the verified correct design of a virtual security module and the study of a Trusted Computing protocol. Furthermore we propose to include NFC-based proximity services in Remote Attestation. The design of a Java API for Trusted Computing that has been released as an official standard. This thesis presents a selection of the author's contributions to some 30 scientific publications.

To conclude, the novel and improved interfaces described can help advance the technology of trusted computing platforms and trusted services, which will ultimately lead to more secure systems.

[1]We are currently researching this in the EC-funded STREP STANCE.

A
Appendix 1 - The JSR 321 API

A.1 The API

In this section we present the full Application Programming Interface Specifications of JSR 321 in a very short notation without comments. For a full reference and interactive outline of the API, the reader is referred to the original standard [327] and the JavaDoc contained therein.

```

package javax.trustedcomputing;

public abstract class TrustedComputingException extends Exception{
    public static final long HIGH_LEVEL_API_LAYER_ERROR;
    public TrustedComputingException();
    public abstract Throwable getCause();
    public abstract long getLowLevelErrorCode();
    public abstract String getMessage();
    public abstract String getShortMessage();
}

public class tpm.PCRsNotAccessibleException extends
     java.lang.RuntimeException{
    public tpm.PCRsNotAccessibleException(java.lang.String string);
}

public interface tpm.TPM{
    public static final java.lang.String PROPERTY_TPM_MANUFACTURER;
    public static final java.lang.String PROPERTY_TPM_VERSION;
    public static final java.lang.String PROPERTY_TPM_FIRMWARE_VERSION;
    public static final java.lang.String PROPERTY_TSS_VENDOR;
    public static final java.lang.String PROPERTY_TSS_VERSION;
    public static final java.lang.String PROPERTY_JSR_REVISION;
    public static final java.lang.String PROPERTY_JSR_VERSION;
    public abstract byte[] getRandom(int length)      throws
        TrustedComputingException;
    public abstract boolean isActivated()      throws
        TrustedComputingException;
    public abstract boolean isEnabled()      throws
        TrustedComputingException;
    public abstract boolean isOwned()      throws
        TrustedComputingException;
    public abstract int getNumberPCR()      throws
        TrustedComputingException;
    public abstract void extendPCR(int PCRindex, tpm.structures.Digest
        data)      throws TrustedComputingException,
        tpm.PCRsNotAccessibleException;
    public abstract void extendPCR(int PCRindex,
        tpm.structures.PCREvent event)      throws
        TrustedComputingException;
    public abstract tpm.structures.PCRInfo readPCR(int[] PCRindices)
              throws TrustedComputingException;
    public abstract void stirRandom(byte[] entropy)      throws
        TrustedComputingException;
    public abstract java.lang.Object getProperty(java.lang.String
        property)      throws TrustedComputingException,
        java.lang.IllegalArgumentException;
}

public abstract class tpm.TPMContext extends java.lang.Object{
    public abstract void close()      throws TrustedComputingException;
    public abstract void connect(java.net.URL remoteAdress)
        throws TrustedComputingException;
    protected void finalize()      throws java.lang.Throwable;
    public static tpm.TPMContext getInstance()      throws
        java.lang.ClassCastException, java.lang.ClassNotFoundException,
        java.lang.InstantiationException,
        java.lang.IllegalAccessException;
    public static tpm.TPMContext getInstance(java.lang.String)
        throws java.lang.ClassCastException,
        java.lang.ClassNotFoundException,
        java.lang.InstantiationException,
        java.lang.IllegalAccessException;
    protected tpm.TPMContext();
    public abstract tpm.TPM getTPMInstance()      throws
        TrustedComputingException;
    public abstract boolean isConnected()      throws
        TrustedComputingException;
    public abstract tpm.keys.KeyManager getKeyManager()      throws
        TrustedComputingException;
    public abstract tpm.tools.Sealer getSealer()      throws
        TrustedComputingException;
    public abstract tpm.tools.Binder getBinder()      throws
        TrustedComputingException;
    public abstract tpm.tools.Signer getSigner()      throws
        TrustedComputingException;
    public abstract tpm.structures.Digest getDigest(byte[] digest);
    public abstract tpm.structures.PCRInfo getPCRInfo();
    public abstract tpm.structures.PCRInfo getPCRInfo(int numberOfPCRs);
    public abstract tpm.structures.Secret
        getSecret(tpm.structures.Digest hashedSecret);
    public abstract tpm.structures.Secret getSecret(char[] password);
    public abstract tpm.structures.Secret getSecret(char[] password,
        boolean addNullTermination, java.nio.charset.Charset encoding)
              throws java.nio.charset.CharacterCodingException;
    public abstract tpm.structures.PCREvent getPCREvent(long eventType,
        tpm.structures.Digest data, java.lang.String eventDescription);
    public abstract tpm.tools.Initializer getInitializer()      throws
        TrustedComputingException,
        java.lang.UnsupportedOperationException;
    public abstract tpm.tools.Attestor getAttestor()      throws
        TrustedComputingException;
    public abstract tpm.tools.Certifier getCertifier()      throws
        TrustedComputingException,
        java.lang.UnsupportedOperationException;
    public abstract tpm.tools.remote.RemoteAttestor getRemoteAttestor();
    public abstract tpm.tools.remote.RemoteBinder getRemoteBinder();
    public abstract tpm.tools.remote.RemoteCertifier
        getRemoteCertifier();
    public abstract tpm.tools.remote.RemoteSigner getRemoteSigner();
}

public interface tpm.keys.BindingKey extends
    tpm.keys.TPMKey,tpm.keys.TPMRSAKey{
}

public interface tpm.keys.IdentityKey extends
    tpm.keys.TPMKey,tpm.keys.TPMRSAKey{
}

public abstract class tpm.keys.KeyManager extends java.lang.Object{
    protected tpm.keys.KeyManager();
    public abstract tpm.keys.BindingKey
        createBindingKey(tpm.keys.StorageKey parent,
        tpm.structures.Secret usageSecret, tpm.structures.Secret
        migrationSecret, boolean isMigratable, boolean isVolatile,
        boolean needsAuthorization, int RSAKeyLength,
        tpm.structures.PCRInfo pcrInfo)      throws
        TrustedComputingException;
    public abstract tpm.keys.SigningKey
        createSigningKey(tpm.keys.StorageKey parent,
        tpm.structures.Secret usageSecret, tpm.structures.Secret
        migrationSecret, boolean isMigratable, boolean isVolatile,
        boolean needsAuthorization, int RSAKeyLength,
        tpm.structures.PCRInfo pcrInfo)      throws
        TrustedComputingException;
    public abstract tpm.keys.StorageKey
        createStorageKey(tpm.keys.StorageKey parent,
        tpm.structures.Secret usageSecret, tpm.structures.Secret
        migrationSecret, boolean isMigratable, boolean isVolatile,
        boolean needsAuthorization,  tpm.structures.PCRInfo pcrInfo)
              throws TrustedComputingException;
    public abstract void deleteTPMKey(java.util.UUID identifier)
        throws TrustedComputingException;
    public abstract void deleteTPMSystemKey(java.util.UUID identifier)
              throws TrustedComputingException;
    public abstract tpm.keys.LegacyKey
        importLegacyKey(tpm.keys.StorageKey parent,
        java.security.KeyPair keyPair, tpm.structures.Secret keySecret)
              throws TrustedComputingException;
    public abstract tpm.keys.StorageRootKey
        loadStorageRootKey(tpm.structures.Secret srkSecret)
        throws TrustedComputingException;
    public abstract tpm.keys.TPMKey
        loadTPMSystemKey(tpm.keys.StorageKey parent,
        java.security.interfaces.RSAPublicKey pubKey,
        tpm.structures.Secret usageSecret)      throws
        TrustedComputingException;
    public abstract tpm.keys.TPMKey loadTPMKey(tpm.keys.StorageKey
        parent, java.security.interfaces.RSAPublicKey pubKey,
        tpm.structures.Secret usageSecret)      throws
        TrustedComputingException;
    public abstract tpm.keys.TPMKey loadTPMKey(tpm.keys.StorageKey
        parent, java.util.UUID identifier, tpm.structures.Secret
        usageSecret)      throws TrustedComputingException;
    public abstract tpm.keys.TPMKey
        loadTPMSystemKey(tpm.keys.StorageKey parent, java.util.UUID
        identifier, tpm.structures.Secret usageSecret)      throws
        TrustedComputingException;
    public abstract void storeTPMKey(tpm.keys.StorageKey parent,
        tpm.keys.TPMKey key, java.util.UUID identifier)      throws
        TrustedComputingException;
    public abstract void storeTPMSystemKey(tpm.keys.StorageKey parent,
        tpm.keys.TPMKey key, java.util.UUID identifier)      throws
        TrustedComputingException;
    public abstract java.util.UUID[] getStoredTPMKeys()      throws
        TrustedComputingException;
    public abstract java.util.UUID[] getStoredTPMSystemKeys()
        throws TrustedComputingException;
}

public class tpm.keys.KeyNotMigratableException extends
    java.lang.RuntimeException{
    public tpm.keys.KeyNotMigratableException(java.lang.String string);
}

public interface tpm.keys.LegacyKey extends
    tpm.keys.TPMKey,tpm.keys.TPMRSAKey{
}

public interface tpm.keys.SigningKey extends
    tpm.keys.TPMKey,tpm.keys.TPMRSAKey{
}

public interface tpm.keys.StorageKey extends
    tpm.keys.TPMKey,tpm.keys.TPMRSAKey{
}

public interface tpm.keys.StorageRootKey extends tpm.keys.StorageKey{
    public static final java.util.UUID SRK_UUID;
    public abstract void changeUsageSecret(tpm.structures.Secret
        ownerSecret, tpm.structures.Secret newSecret)      throws
        TrustedComputingException;
    static {};
}

public interface tpm.keys.TPMKey{
    public abstract void changeMigrationSecret(tpm.keys.StorageKey
        parent, tpm.structures.Secret oldSecret, tpm.structures.Secret
        newSecret)       throws TrustedComputingException,
        tpm.keys.KeyNotMigratableException;
    public abstract void changeUsageSecret(tpm.keys.StorageKey
        parent,tpm.structures.Secret oldSecret, tpm.structures.Secret
        newSecret)       throws TrustedComputingException;
    public abstract void unload()       throws
        TrustedComputingException;
    public abstract void setUUID(java.util.UUID keyIdentifier);
    public abstract java.util.UUID getUUID();
}

public interface tpm.keys.TPMRSAKey extends
    java.security.interfaces.RSAKey{
    public abstract java.math.BigInteger getModulus();
    public abstract java.security.interfaces.RSAPublicKey
        getPublicKey()       throws TrustedComputingException;
}

public abstract class tpm.structures.Digest extends java.lang.Object{
    protected tpm.structures.Digest();
    public tpm.structures.Digest(byte[] digest);
    public abstract byte[] getBytes();
    public abstract boolean equals(java.lang.Object other);
    public abstract int hashCode();
}

public class tpm.structures.PCREvent extends java.lang.Object{
    protected final tpm.structures.Digest dataDigest_;
    protected final java.lang.String eventDescription_;
    protected final long eventType_;
    public tpm.structures.PCREvent(long eventType,
        tpm.structures.Digest data, java.lang.String eventDescription);
    public tpm.structures.Digest getDataDigest();
    public java.lang.String getEventDescription();
    public long getEventType();
}

public abstract class tpm.structures.PCRInfo extends java.lang.Object{
    public tpm.structures.PCRInfo();
    public tpm.structures.PCRInfo(int numberOfPCRs)       throws
        java.lang.IllegalArgumentException;
    public abstract tpm.structures.Digest getPCRValue(int index);
    public abstract int[] getValueIndices();
    public abstract void setPCRValue(int index, tpm.structures.Digest
        value);
    public abstract int getNumberOfPCRs();
}

public abstract class tpm.structures.Secret extends java.lang.Object{
    public static tpm.structures.Secret WELL_KNOWN_SECRET;
    protected tpm.structures.Secret();
    public tpm.structures.Secret(tpm.structures.Digest hashedSecret);
    public tpm.structures.Secret(char[] password);
    public tpm.structures.Secret(char[] password, boolean
        addNullTermination, java.nio.charset.Charset encoding) ;
    public abstract void flushSecret();
    public abstract byte[] getBytes();
}

public class tpm.structures.ValidationData extends java.lang.Object
     implements java.io.Serializable{
    protected final byte[] data_;
    protected final byte[] nonce_;
    protected final byte[] validationData_;
    public tpm.structures.ValidationData(byte[] nonce, byte[] data,
        byte[] validationData);
    public byte[] getData();
    public byte[] getNonce();
    public byte[] getValidationData();
}

public abstract class tpm.tools.Attestor extends
     tpm.tools.remote.RemoteAttestor{
    public tpm.tools.Attestor(tpm.TPMContext context);
    public abstract tpm.structures.ValidationData quote(int[]
        PCRindices, tpm.keys.IdentityKey key, tpm.structures.Digest
        nonce)       throws TrustedComputingException,
        tpm.PCRsNotAccessibleException;
    public abstract tpm.structures.ValidationData quote(int[]
        PCRindices, tpm.keys.IdentityKey key, tpm.structures.Digest
        nonce)       throws TrustedComputingException,
        tpm.PCRsNotAccessibleException;
}

public abstract class tpm.tools.Binder extends
     tpm.tools.remote.RemoteBinder{
    public tpm.tools.Binder(tpm.TPMContext context);
    public abstract byte[] unbind(byte[] encryptedData,
        tpm.keys.BindingKey key)       throws TrustedComputingException;
}

public abstract class tpm.tools.Certifier extends
     tpm.tools.remote.RemoteCertifier{
    public tpm.tools.Certifier(tpm.TPMContext context);
    public abstract tpm.structures.ValidationData
        certifyKey(tpm.keys.TPMKey toBeCertified, tpm.keys.IdentityKey
        certifyingKey, tpm.structures.Digest nonce)       throws
        TrustedComputingException, java.lang.IllegalArgumentException;
    public abstract tpm.structures.ValidationData
        certifyKey(tpm.keys.TPMKey toBeCertified, tpm.keys.SigningKey
        certifyingKey, tpm.structures.Digest nonce)       throws
        TrustedComputingException;
    public abstract tpm.structures.ValidationData
        certifyKey(tpm.keys.TPMKey toBeCertified certifyingKey,
        tpm.keys.LegacyKey, tpm.structures.Digest nonce)       throws
        TrustedComputingException;
}

```

```
public abstract class tpm.tools.Initializer extends java.lang.Object{
    public tpm.tools.Initializer(tpm.TPMContext context);
    public abstract void takeOwnership(tpm.structures.Secret
        ownerSecret, tpm.structures.Secret srkSecret)       throws
        TrustedComputingException;
    public abstract void clearOwnership(tpm.structures.Secret
        ownerSecret)       throws TrustedComputingException;
}

public abstract class tpm.tools.Sealer extends java.lang.Object{
    public tpm.tools.Sealer(tpm.TPMContext context);
    public abstract byte[] seal(byte[] plainData,
        tpm.structures.PCRInfo targetState, tpm.keys.StorageKey
        storageKey, tpm.structures.Secret dataSecret)       throws
        TrustedComputingException, tpm.PCRsNotAccessibleException;
    public abstract byte[] unseal(byte[] encrytedData,
        tpm.keys.StorageKey key, tpm.structures.Secret dataSecret)
              throws TrustedComputingException,
        tpm.PCRsNotAccessibleException;
}

public abstract class tpm.tools.Signer extends
     tpm.tools.remote.RemoteSigner{
    public tpm.tools.Signer(tpm.TPMContext context);
    public abstract byte[] sign(byte[] plainData, tpm.keys.LegacyKey
        key)       throws TrustedComputingException;
    public abstract byte[] sign(byte[] plainData, tpm.keys.SigningKey
        key)       throws TrustedComputingException;
}

public abstract class tpm.tools.remote.RemoteAttestor extends
     java.lang.Object{
    public tpm.tools.remote.RemoteAttestor();
    public abstract boolean validateQuote(tpm.structures.ValidationData
        dataToValidate, java.security.interfaces.RSAPublicKey
        identityKey, tpm.structures.Digest nonce,
        tpm.structures.PCRInfo expectedValues)       throws
        java.security.GeneralSecurityException,
        TrustedComputingException;
}

public abstract class tpm.tools.remote.RemoteBinder extends
     java.lang.Object{
    public tpm.tools.remote.RemoteBinder();
    public abstract byte[] bind(byte[] plainData,
        java.security.interfaces.RSAPublicKey bindingKey)       throws
        TrustedComputingException;
}

public abstract class tpm.tools.remote.RemoteCertifier extends
     java.lang.Object{
    public tpm.tools.remote.RemoteCertifier();
    public abstract boolean validate(tpm.structures.ValidationData
        dataToValidate, java.security.interfaces.RSAPublicKey
        certifiedKey, java.security.interfaces.RSAPublicKey
        certifyingKey, tpm.structures.Digest nonce)       throws
        java.security.GeneralSecurityException;
    public abstract boolean
        containsPCRInfo(tpm.structures.ValidationData certifiedKeyInfo,
        tpm.structures.PCRInfo desiredPCRInfo);
    public abstract boolean isBindingKey(tpm.structures.ValidationData
        certifiedKeyInfo);
    public abstract boolean isKeyOfLength(tpm.structures.ValidationData
        certifiedKeyInfo, int desiredKeyLength);
    public abstract boolean isLegacyKey(tpm.structures.ValidationData
        certifiedKeyInfo);
    public abstract boolean isMigratable(tpm.structures.ValidationData
        certifiedKeyInfo);
    public abstract boolean isSigningKey(tpm.structures.ValidationData
        certifiedKeyInfo);
    public abstract boolean isStorageKey(tpm.structures.ValidationData
        certifiedKeyInfo);
    public abstract boolean isVolatile(tpm.structures.ValidationData
        certifiedKeyInfo);
    public abstract boolean
        needsAuthorization(tpm.structures.ValidationData
        certifiedKeyInfo);
}

public abstract class tpm.tools.remote.RemoteSigner extends
    java.lang.Object{
    public tpm.tools.remote.RemoteSigner();
    public abstract boolean validate(byte[] signature, byte[] data,
        java.security.interfaces.RSAPublicKey key)      throws
        java.security.GeneralSecurityException;
}
```

B Appendix - List of Publications

This appendix lists, in accordance with the statutes of the doctoral school of Computer Science at Graz University of Technology, the publications of the author.

B.1 Journals

This section presents journal publications by the author in reversed chronological order.

[122] E. Gatial, Z. Balogh, D. Hein, L. Hluchý, M. Pirker, and R. Toegl. Securing agents using secure docking module. *Techn. Sc.*, 15(1), 2012.

[339] R. Toegl, T. Winkler, M. Nauman, and T. W. Hong. Specification and Standardization of a Java Trusted Computing API. *Softw. Pract. Exper.*, 42(8):945–965, 2012.

[326] R. Toegl and M. Hutter. An approach to introducing locality in remote attestation using near field communications. *The Journal of Supercomputing*, 55(2):207–227, 2011.

[153] M. Hutter and R. Toegl. Touch‘n’ Trust: An NFC-enabled trusted platform module. *The International Journal on Advances in Security*, 4(1 & 2):131–141, 2011.

[253] M. Pirker and R. Toegl. Towards a virtual trusted platform. *Jour-*

nal of Universal Computer Science, 16(4):531–542, 2010. `http://www.jucs.org/jucs_16_4/towards_a_virtual_trusted`.

[142] D. Hein, R. Toegl, and S. Kraxberger. An autonomous attestation token to secure mobile agents in disaster response. *Security and Communication Networks*, 3(5):421–438, 2010.

[91] K. Dietrich, T. Vejda, R. Toegl, M. Pirker, and P. Lipp. Can you Really Trust your Computer Today? — Emerging architectures for Trusted Computing. *ENISA Quarterly*, 3(3):8–9, Jul–Sep 2007.

B.2 Conference and Workshop Proceedings

This section presents publications in workshop and conference proceedings by the author in reversed chronological order.

[184] S. Kraxberger, R. Toegl, M. Pirker, E. P. Guijarro, and G. G. Millan. Trusted identity management for overlay networks. In R. H. Deng and T. Feng, editors, *Information Security Practice and Experience. 9th International Conference, ISPEC 2013, Lanzhou, China, May 12-14, 2013. Proceedings*, volume 7863 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2013.

[323] R. Toegl. Verification of a trusted virtual security module. In M. Bond, R. Focardi, F. Sibylle, and G. Steel, editors, *Analysis of Security APIs (Dagstuhl Seminar 12482)*, Dagstuhl Reports, page 166. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013. Abstract.

[341] R. Toegl, J. Winter, and M. Pirker. A path towards ubiquitous protection of media. In J. Lyle, S. Faily, and M. Winandy, editors, *Proceedings of the Workshop on Web Applications and Secure Hardware (WASH), Co-located with the 6th International Conference on Trust and Trustworthy Computing (TRUST 2013)*, volume 1011 of *CEUR Workshop Proceedings*, pages 32–38, London, United Kingdom, 6 2013. Sun SITE Central Europe, RWTH Aachen University. Position Paper.

[259] M. Pirker, J. Winter, and R. Toegl. Lightweight distributed attestation for the cloud. In F. Leymann, I. Ivanov, M. van Sinderen, and T. Shan, editors, *CLOSER*, pages 580–585. SciTePress, 2012.

[364] 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*, pages 1–15, Beijing, China,

2012. Springer-Verlag.

[260] M. Pirker, J. Winter, and R. Toegl. Lightweight distributed attestation for the cloud. In F. Leymann, I. Ivanov, M. van Sinderen, and T. Shan, editors, *CLOSER*, pages 580–585. SciTePress, 2012.

[143] D. Hein, R. Toegl, M. Pirker, E. Gatial, Z. Balogh, H. Brandl, and L. Hluchý. Securing mobile agents for crisis management support. In *Proceedings of the seventh ACM workshop on Scalable Trusted Computing*, STC '12, pages 85–90, New York, NY, USA, 2012. ACM.

[336] 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*, pages 145–160. Springer Berlin Heidelberg, 2013.

[262] 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, pages 45–53. Springer Berlin Heidelberg, 2011.

[333] 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*, volume 6802 of *Lecture Notes in Computer Science*, pages 326–345. Springer Berlin / Heidelberg, 2011.

[126] 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*, pages 138–145. Springer Berlin Heidelberg, 2011.

[152] M. Hutter and R. Toegl. A trusted platform module for near field communication. In *Systems and Networks Communications (ICSNC), 2010 Fifth International Conference on*, pages 136–141. IEEE, 2010.

[254] 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 '10: Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, volume 6101 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010.

[320] R. Toegl. Tagging the turtle: Local attestation for kiosk computing.

In J. H. Park, H.-H. Chen, M. Atiquzzaman, C. Lee, T. hoon Kim, and S.-S. Yeo, editors, *Advances in Information Security and Assurance*, volume 5576 of *Lecture Notes in Computer Science*, pages 60–69. Springer Berlin / Heidelberg, 2009.

[255]M. Pirker, R. Toegl, D. Hein, and P. Danner. A PrivacyCA for anonymity and trust. In L. Chen, C. J. Mitchell, and A. Martin, editors, *TRUST 2009: Proceedings of the 2nd International Conference on Trusted Computing*, volume 5471 of *Lecture Notes in Computer Science*, pages 101–119. Springer Berlin / Heidelberg, 2009. [330] 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*, pages 60–67.
Vieweg+Teubner, 2009.

[338] R. Toegl, T. Winkler, M. Nauman, and T. Hong. Towards platform-independent trusted computing. In *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing*, pages 61–66, Chicago, Illinois, USA, 2009. ACM.

[141] D. Hein and R. Toegl. An autonomous attestation token to secure mobile agents in disaster response. In A. Schmidt and S. Lian, editors, *Security and Privacy in Mobile Information and Communication Systems (MobiSec 2009)*, volume 17 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 46–57, Turin, Italy, June 2009. Springer Berlin/Heidelberg.

[252] M. Pirker and R. Toegl. Sichere Softwaremodule durch Einsatz von Virtualisierung und Trusted Computing. In W. Seböck and E. Huber, editors, *Tagungsband der 7. Information Security Konferenz.* Österreichische Computer Gesellschaft, 2009. ISBN 978-3-85403-257-1. In German.

[90] K. Dietrich, M. Pirker, T. Vejda, R. Toegl, T. Winkler, and P. Lipp. A practical approach for establishing trust relationships between remote platforms using trusted computing. In G. Barthe and C. Fournet, editors, *Trustworthy Global Computing, Proceedings*, volume 4912 of *Lecture Notes in Computer Science*, pages 156–168. Springer Verlag, 2008. ISBN 978-3-540-78662-7.

[353] T. Vejda, R. Toegl, M. Pirker, and T. Winkler. Towards trust services for language-based virtual machines for grid computing. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Trusted Computing âĂŞ Challenges and Applications First International Conference on Trusted Computing and Trust in Information Technologies, TRUST 2008 Villach, Austria, March 11-12, 2008 Proceedings*, volume 4968 of *Lecture Notes in Computer Science.* Springer Verlag, 2008.

[325] R. Toegl, G. Hofferek, K. Greimel, A. H. Y. Leung, R.-W. Phan, and

R. Bloem. Formal analysis of a TPM-based secrets distribution and storage scheme. In *Proceedings TRUSTCOM 2008, in: Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 2289–2294. IEEE Computer Society, 2008.

[328] R. Toegl, C. Parraga Niebla, and U. Birnbacher. Framing efficiency optimization for DVB-S2 systems. In *Global Telecommunications Conference, 2006. GLOBECOM '06. IEEE*, 2006.

[329] R. Toegl, C. Parraga Niebla, and U. Birnbacher. Framing efficiency optimization for DVB-S2 systems with QoS guarantees. In *Ka-band Conference 2006. Ka and Broadband Communications Conference, Proceedings*, Naples, Italy, 2006.

[324] R. Toegl, U. Birnbacher, and O. Koudelka. Deploying IP telephony over satellite links. In *Wireless Communication Systems, 2005. 2nd International Symposium on*, pages 624–628, 2005.

Bibliography

[1] M. Abadi. Security protocols: principles and calculi tutorial notes. In A. Aldini and R. Gorrieri, editors, *Foundations of security analysis and design IV*, pages 1–23. Springer-Verlag, 2007.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: the Spi calculus. In *Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47, Zurich, Switzerland, 1997. ACM.

[3] K. Ables. An alleged attack on key delegation in the trusted platform module. MSc Advanced Computer Science First semester mini-project, University of Birmingham, `http://www.computer-science.birmingham.ac.uk/~mdr/research/papers/pdf/09-ables-3.pdf`, 2009. Website accessed November 15, 2012.

[4] A. Abu-Mahfouz and G. Hancke. Distance bounding: A practical security solution for real-time location systems. *Industrial Informatics, IEEE Transactions on*, 9(1):16–27, 2013.

[5] M. Achemlal, S. Gharout, and C. Gaber. Trusted platform module as an enabler for security in cloud computing. In *Network and Information Systems Security (SAR-SSI), 2011 Conference on*, pages 1–6, 2011.

[6] C. Adams and S. Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations.* Addison-Wesley Professional, 2 edition, 2002. ISBN-13: 978-0321743091.

[7] 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.

[8] Advanced Micro Devices. *AMD64 Virtualization: Secure Virtual Machine Architecture Reference Manual*, May 2005. Publication No. 33047; Rev. 3.01.

[9] M. Aigner, S. Dominikus, and M. Feldhofer. A system of secure virtual coupons using NFC technology. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops '07. Fifth Annual IEEE International Conference on*, pages 362–366, 2007.

[10] M. Alam, X. Zhang, M. Nauman, and T. Ali. Behavioral attestation for web services (ba4ws). In *Proceedings of the 2008 ACM workshop on Secure web services*, pages 21–28, Alexandria, Virginia, USA, 2008. ACM.

[11] G. Alpár and J.-H. Hoepman. Avoiding man-in-the-middle attacks when verifying public terminals. In J. Camenisch, B. Crispo, S. Fischer-Hübner, R. Leenes, and G. Russello, editors, *IFIP Advances in Information and Communication Technology*, volume 375, pages 261–273. Springer Berlin Heidelberg, 2012.

[12] S. Alsouri, O. Dagdelen, and S. Katzenbeisser. Group-based attestation: Enhancing privacy and management in remote attestation. In A. Acquisti, S. Smith, and A.-R. Sadeghi, editors, *Lecture Notes in Computer Science*, volume 6101, pages 63–77. Springer Berlin Heidelberg, 2010.

[13] American National Standards Institute (ANSI). AMERICAN NATIONAL STANDARD X9.62-2005. Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA), 2005.

[14] R. Anderson. 'Trusted Computing' Frequently Asked Questions - TC / TCG / LaGrande / NGSCB / Longhorn / Palladium / TCPA. `http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html`, 8 2003. Website visited October 30, 2012.

[15] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2nd edition, April 2008. ISBN: 978-0-470-06852-6.

[16] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors-a survey. *Proceedings of the IEEE DOI - 10.1109/JPROC.2005.862423*, 94(2):357–369, 2006.

[17] R. Anderson and R. Needham. Programming satan's computer. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 1995.

[18] M. Arapinis, E. Ritter, and M. Ryan. Statverif: Verification of stateful processes. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 33–47, 2011.

[19] 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*, pages 65–71. IEEE Computer Society, 1997.

[20] ARM Limited. TrustZone API Specification v2.0, June 2006. PRD29-USGC-000089.

[21] ARM Limited. ARM Security Technology Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf Website accessed July 24, 2013, 2009. PRD29-GENC-009492C.

[22] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P.-C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proceedings of CAV'2005*, number 3576 in Lecture Notes in Computer Science, pages 281–285. Springer-Verlag, 2005.

[23] A. Armando and L. Compagna. SAT-based model-checking for security protocols analysis. *Int. J. Inf. Secur.*, 7(1):3–32, 2008.

[24] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[25] T. W. Arnold and L. van Doorn. The IBM PCIXCC: a new cryptographic coprocessor for the IBM eServer. *IBM J. Res. Dev.*, 48(3-4):475–487, 2004.

[26] Atmel Corporation. 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash. Available online at http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf, August 2007.

[27] AVANTSSAR Consortium. http://www.avantssar.eu/. Website accessed August 27, 2013.

[28] AVISPA Consortium. The AVISPA project. http://www.avispa-project.org/, 2002-2005. Website accessed August 22, 2012.

[29] AVISPA Consortium. The AVISPA library. http://www.avispa-project.org/library/avispa-library-index.html, 2005. Website accessed August 22, 2012.

[30] G. Avoine, M. A. Bingöl, S. Kardaş, C. Lauradoux, and B. Martin. A Framework for Analyzing RFID Distance Bounding Protocols. *Journal of Computer Security – Special Issue on RFID System Security*, 19(2):289–317, March 2011.

[31] D. Balaban. NFC smartphone chip shipments in 2012 surge past projections. http://nfctimes.com/news/nfc-smartphone-chip-shipments-2012-surge-past-projections Website accessed October 22, 2013, 3 2013. NFC Times.

[32] B. Balacheff, L. Chen, D. Plaquin, and G. Proudler. *Trusted Computing Platforms: TCPA Technology in Context.* Prentice Hall, 2002. ISBN-13: 978-0-13-009220-5.

[33] A. Baldwin, C. Dalton, S. Shiu, K. Kostienko, and Q. Rajpoot. Providing secure services for a virtual infrastructure. *SIGOPS Oper. Syst. Rev.*, 43(1):44–51, 2009.

[34] S. Balfe, E. Gallery, C. Mitchell, and K. Paterson. Challenges for trusted computing. *Security & Privacy, IEEE*, 6(6):60–66, 2008.

[35] E. Bangerter, M. Djackov, and A.-R. Sadeghi. A demonstrative ad hoc attestation system. In T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, editors, *Lecture Notes in Computer Science*, volume 5222, pages 17–30. Springer Berlin Heidelberg, 2008.

[36] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[37] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, June 2005. Published online December 2004.

[38] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Longman, 1st edition, 1999. ISBN-13: 978-0201616415.

[39] K. Beck. *JUnit Pocket Guide.* O'Reilly Media, 1st edition, 2004. ISBN 978-0-596-00743-0.

[40] M. Bellare and P. Rogaway. Optimal asymmetric encryption – How to encrypt with RSA. In A. D. Santis, editor, *Eurocrypt 94 Proceedings*, volume 950 of *Lecture Notes in Computer Science.* Springer Verlag, 1995.

[41] S. Berger, R. Cáceres, K. Goldman, D. Pendarakis, R. Perez, J. R. Rao, E. Rom, R. Sailer, W. Schildhauer, D. Srinivasan, S. Tal, and E. Valdez. Security for the cloud infrastructure: trusted virtual data center implementation. *IBM J. Res. Dev.*, 53(4):560–571, 2009.

[42] S. Berger, R. Cáceres, K. 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. USENIX, 2006.

[43] 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. *SIGOPS Oper. Syst. Rev.*, 42(1):40–47, 2008.

[44] M. Bishop. *Computer Security: Art and Science.* Addison-Wesley Professional, Dec. 2002.

[45] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[46] B. Blanchet, V. Cheval, X. Allamigeon, and B. Smyth. Proverif: Cryptographic protocol verifier in the formal model. `http://proverif.inria.fr/`. Website accessed October 30, 2012.

[47] M. Bond and R. Anderson. API-level attacks on embedded systems. *Computer*, 34(10):67–75, 2001. IEEE.

[48] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM conference on Computer and Communications Security*, pages 260–269, Chicago, Illinois, USA, 2010. ACM.

[49] S. Brands and D. Chaum. Distance-bounding protocols. In *Workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 344–359, Lofthus, Norway, 1994. Springer-Verlag New York, Inc.

[50] A. Brett, N. Kuntze, and A. Schmidt. Trusted watermarks. In *Broadband Multimedia Systems and Broadcasting, 2009. BMSB '09. IEEE International Symposium on*, pages 1–7, 2009.

[51] A. Brett and A. Leicher. Ethemba trusted host environment mainly based on attestation. `http://ethemba.novalyst.de/wordpress/wp-content/uploads/2009/11/ethemba1.pdf`, 3 2009. Website accessed November 15, 2012.

[52] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, Washington DC, USA, 2004. ACM.

[53] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Replay attack in TCG specification and solution. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 127–137. IEEE Computer Society, 2005.

[54] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[55] G. Cabiddu, E. Cesena, R. Sassu, D. Vernizzi, G. Ramunno, and A. Lioy. The trusted platform agent. *IEEE Software*, 28:35–41, 2011.

[56] S. Cabuk, L. Chen, D. Plaquin, and M. Ryan. Trusted integrity measurement and reporting for virtualized platforms. In L. Chen and M. Yung, editors, *INTRUST 2009*, volume 6163 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2009.

[57] 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*, pages 65–78, Seattle, Washington, 2005. ACM.

[58] C. Cachin and N. Chandran. A secure cryptographic token interface. In *Computer Security Foundations Symposium, 2009. CSF '09. 22nd IEEE*, pages 141–153, 2009.

[59] D. Catteddu and G. Hogben. Cloud Computing benefits, risks and recommendations for information security. Technical report, European Network and Information Security Agency (ENISA), 2009.

[60] L. Catuogno, A. Dmitrienko, K. Eriksson, D. Kuhlmann, G. Ramunno, A.-R. Sadeghi, S. Schulz, M. Schunter, M. Winandy, and J. Zhan. Trusted virtual domains - design, implementation and lessons learned. In L. Chen and M. Yung, editors, *INTRUST 2009*, volume 6163 of *Lecture Notes in Computer Science*, pages 156–179. Springer, 2010.

[61] A. Celesti, A. Salici, M. Villari, and A. Puliafito. A remote attestation approach for a secure virtual machine migration in federated cloud environments. In *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on*, pages 99–106, 2011.

[62] D. W. Chadwick, G. Zhao, S. Otenko, R. Laborde, L. Su, and T. A. Nguyen. Permis a modular authorization infrastructure. *Concurrency and Computation: Practice and Experience*, 20(11):1341–1357, 2008.

[63] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. van Doorn. *A Practical Guide to Trusted Computing.* IBM Press, 1st edition, 2008. ISBN-13: 978-0132398428.

[64] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.-R. Sadeghi, and C. Stüble. A protocol for property-based attestation. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing*, STC '06, pages 7–16, New York, NY, USA, 2006. ACM.

[65] L. Chen and M. Ryan. Offline dictionary attack on TCG TPM weak authorisation data, and solution. In D. Gawrock, H. Reimer, A.-R. Sadeghi, and C. Vishik, editors, *Future of Trust in Computing*, pages 193–196. Vieweg+Teubner, 2009.

[66] L. Chen and M. Ryan. Attack, solution and verification for shared authorisation data in TCG TPM. In P. Degano and J. D. Guttman, editors,

Proceedings of Sixth International Workshop on Formal Aspects in Security and Trust (FAST'09), volume 5983 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, 2010. Springer.

[67] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In E. Brinksma and K. Larsen, editors, *Lecture Notes in Computer Science*, volume 2404, pages 359–364. Springer Berlin Heidelberg, 2002.

[68] L. S. Clair, J. Schiffman, T. Jaeger, and P. McDaniel. Establishing and sustaining system integrity via root of trust installation. *Computer Security Applications Conference, Annual*, 0:19–29, 2007.

[69] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. `http://web.cs.wpi.edu/~guttman/cs559_website/clarkjacob97survey.pdf` Website accessed June 25, 2013, 1997.

[70] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999. ISBN: 9780262032704.

[71] S. Clarke. Measuring API usability. *Dr. Dobbs Journal*, 2004.

[72] Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing V2.1. `https://cloudsecurityalliance.org/csaguide.pdf` Website accessed June 19, 2013, 12 2009.

[73] J. Clulow. On the security of PKCS#11. In C. D. Walter, C. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 411–425. Springer Berlin Heidelberg, 2003.

[74] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of remote attestation. *Int. J. Inf. Secur.*, 10(2):63–81, 2011.

[75] G. Coker, J. Guttman, P. Loscocco, J. Sheehy, and B. Sniffen. Attestation: Evidence and trust. In L. Chen, M. Ryan, and G. Wang, editors, *Lecture Notes in Computer Science*, volume 5308, pages 1–18. Springer Berlin Heidelberg, 2008.

[76] Common Criteria Recognition Arrangement. Common criteria for information technology security evaluation. `http://www.commoncriteriaportal.org/cc/`, 2009. Website visited March 14, 2012.

[77] Computer Security Center. *Trusted Computer System Evaluation Criteria*. Department of Defense, 1983. CSC-STD-00l-83 AKA 'Orange Book'.

[78] Computer Security Center. *Trusted Computer System Evaluation Criteria*. Department of Defense, 1985. DoD 5200.28-STD.

[79] A. Cooper and A. Martin. Towards a secure, tamper-proof grid platform. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, 2006. DOI - 10.1109/CCGRID.2006.103.

[80] L. Coppolino, M. Jäger, N. Kuntze, and R. Rieke. A trusted information agent for security information and event management. In *Proc. ICONS 2012, The Seventh International Conference on Systems.* Think MInd, 2012.

[81] C. Cremers. The Scyther tool: Verification, falsification, and analysis of security protocols - tool paper. In A. Gupta and S. Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 414–418. Springer Berlin / Heidelberg, 2008.

[82] C. Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 119–128, Alexandria, Virginia, USA, 2008. ACM.

[83] C. Cremers, P. Lafourcade, and P. Nadeau. Comparing state spaces in automatic security protocol analysis. In V. Cortier, C. Kirchner, M. Okada, and H. Sakurada, editors, *Formal to Practical Security*, volume 5458 of *Lecture Notes in Computer Science*, pages 70–94. Springer Berlin / Heidelberg, 2009.

[84] P. Danner and D. Hein. A trusted computing identity collation protocol to simplify deployment of new disaster response devices. *Journal of Universal Computer Science*, 16(9):1139–1151, may 2010.

[85] S. Delaune, S. Kremer, M. Ryan, and G. Steel. Formal analysis of protocols based on tpm state registers. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 66–80, 2011.

[86] S. Delaune, S. Kremer, and G. Steel. Formal security analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, Jan. 2010.

[87] Y. Desmedt, C. Goutier, and S. Bengio. Special uses and abuses of the Fiat-Shamir passport protocol (extended abstract). In C. Pomerance, editor, *Lecture Notes in Computer Science*, volume 293, pages 21–39–. Springer Berlin Heidelberg, 1988.

[88] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[89] K. Dietrich. Anonymous client authentication for transport layer security. In B. De Decker and I. Schaumüller-Bichl, editors, *Communications and*

Multimedia Security, volume 6109 of *Lecture Notes in Computer Science*, pages 268–280. Springer Berlin / Heidelberg, 2010.

[90] K. Dietrich, M. Pirker, T. Vejda, R. Toegl, T. Winkler, and P. Lipp. A practical approach for establishing trust relationships between remote platforms using trusted computing. In G. Barthe and C. Fournet, editors, *Trustworthy Global Computing, Proceedings*, volume 4912 of *Lecture Notes in Computer Science*, pages 156–168. Springer Verlag, 2008. ISBN 978-3-540-78662-7.

[91] K. Dietrich, T. Vejda, R. Toegl, M. Pirker, and P. Lipp. Can you Really Trust your Computer Today? — Emerging architectures for Trusted Computing. *ENISA Quarterly*, 3(3):8–9, Jul–Sep 2007.

[92] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[93] A. Dmitrienko, A.-R. Sadeghi, S. Tamrakar, and C. Wachsmann. Smart-tokens: Delegable access control with NFC-enabled smartphones. In S. Katzenbeisser, E. Weippl, L. Camp, M. Volkamer, M. Reiter, and X. Zhang, editors, *Lecture Notes in Computer Science*, volume 7344, pages 219–238. Springer Berlin Heidelberg, 2012.

[94] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[95] L. Dong and K. Chen. *Cryptographic Protocol - Security Analysis Based on Trusted Freshness.* Springer, 2011. ISBN 978-3-642-24072-0.

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

[97] ECMA. *ECMA-340: Near Field Communication — Interface and Protocol (NFCIP-1).* ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, Dec. 2004.

[98] ECMA International. ECMA Standard 385-2008: NFC-SEC: NFCIP-1 Security Services and Protocol, December 2008.

[99] ECMA International. ECMA Standard 386-2008: NFC-SEC-01: NFC-SEC Cryptography Standard using ECDH and AES, December 2008.

[100] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Lecture Notes in Computer Science*, volume 2919, pages 502–518–. Springer Berlin Heidelberg, 2004.

[101] J.-E. Ekberg, N. Asokan, K. Kostiainen, and A. Rantala. Scheduling execution of credentials in constrained secure environments. In *Proceedings of the 3rd ACM workshop on Scalable Trusted Computing (STC)*, ACM CCS Workshop Proceedings, pages 61–70, New York, NY, USA, 2008. ACM.

[102] J.-E. Ekberg and S. Bugiel. Trust in a small package: minimized MRTM software implementation for mobile secure environments. In *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC)*, ACM CCS Workshop Proceedings, pages 9–18, New York, NY, USA, 2009. ACM.

[103] J. Ellis. The story of non-secret encryption. Published on GCHQ Website in 1997. Currently available only through `http://web.archive.org/web/20030610193721/http://jya.com/ellisdoc.htm`, 1987. Website accessed October 30, 2012.

[104] EMSCB Project Consortium. The European Multilaterally Secure Computing Base (EMSCB) project. `http://www.emscb.com/`, 2004–2007. Website accessed January 29, 2013.

[105] P. England. Practical techniques for operating system attestation. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Lecture Notes in Computer Science*, volume 4968, pages 1–13. Springer Berlin Heidelberg, 2008.

[106] P. England, B. Lampson, J. Manferdelli, and B. Willman. A trusted open platform. *Computer*, 36(7):55–62, July 2003.

[107] P. England and J. Loeser. Para-virtualized TPM sharing. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Lecture Notes in Computer Science*, volume 4968, pages 119–132. Springer Berlin Heidelberg, 2008.

[108] ENISA. ATM Crime: Overview of the European situation and golden rules on how to avoid it. Technical Report ISBN-13 978-92-9204-023-9, European Network and Information Security Agency (ENISA), 9 2009. `http://www.enisa.europa.eu/activities/cert/security-month/deliverables/2009/atmcrime` Website accessed July 30, 2013.

[109] F. Fabbri. Progetto e realizzazione di un protocollo di verifica dell'affidabilita' di un terminale remoto. Tesi di laurea specialistica, Università di Pisa, 2007. In Italian.

[110] F. Fabrega, J. Herzog, and J. Guttman. Strand spaces: why is a security protocol correct? In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 160–171, 1998.

[111] U. Farooq and D. Zirkler. API peer reviews: a method for evaluating usability of application programming interfaces. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 207–210, Savannah, Georgia, USA, 2010. ACM.

[112] K. Finkenzeller. *RFID Handbuch.* Carl Hanser Verlag, 6th edition, 2012. ISBN: 978-3-446-42992-5. In German.

[113] M. Fossi et al. Symantec internet security threat report - trends for 2010. Technical Report 16, Symantec Corporation, April 2011.

[114] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.

[115] L. Francis, G. Hancke, K. Mayes, and K. Markantonakis. Practical NFC peer-to-peer relay attack using mobile phones. In S. Ors Yalcin, editor, *Lecture Notes in Computer Science*, volume 6370, pages 35–49. Springer Berlin Heidelberg, 2010.

[116] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig. ARM TrustZone as a virtualization technique in embedded systems. In *Proceedigns of the Twelfth Real-Time Linux Workshop*, October 2010. `http://os.inf.tu-dresden.de/papers_ps/rtlws2010_armtrustzone.pdf` Website accessed July 23,2013.

[117] S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In P. Degano and L. ViganÃš, editors, *Lecture Notes in Computer Science*, volume 5511, pages 92–106. Springer Berlin Heidelberg, 2009.

[118] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1994.

[119] 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 Symposium on Operating System Principles(SOSP 2003)*, pages 193–206. ACM New York, NY, USA, October 2003.

[120] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. In D. Grunwald, R. Han, E. de Lara, and C. S. Ellis, editors, *MobiSys*, pages 199–210. ACM, 2008.

[121] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proc. 12th National Computer Security Conf.*, pages 305–319, Baltimore, 1989. NIST/NCSC.

[122] E. Gatial, Z. Balogh, D. Hein, L. Hluchý, M. Pirker, and R. Toegl. Securing agents using secure docking module. *Techn. Sc.*, 15(1), 2012.

[123] C. Gebhardt and C. Dalton. Lala: a late launch application. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 1–8, Chicago, Illinois, USA, 2009. ACM.

[124] C. Gebhardt and A. Tomlinson. Secure Virtual Disk Images for Grid Computing. In *3rd Asia-Pacific Trusted Infrastructure Technologies Conference (APTC 2008).* IEEE Computer Society, October 2008.

[125] V. Getov, G. von Laszewski, M. Philippsen, and I. Foster. Multiparadigm communications in Java for grid computing. *Commun. ACM*, 44(10):118–125, 2001.

[126] 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*, pages 138–145. Springer Berlin Heidelberg, 2011.

[127] GlobalPlatform. TEE Client API Specification v1.0. `http://www.globalplatform.org/specificationsdevice.asp` Website accessed Jula 23, 2013, July 2011.

[128] GlobalPlatform. TEE Internal API Specification v1.0. `http://www.globalplatform.org/specificationsdevice.asp` Website accessed Jula 23, 2013, December 2011.

[129] K. Goldman and S. Potter. SHA-1 uses in TPM v1.2. `http://www.trustedcomputinggroup.org/files/resource_files/72563193-1A4B-B294-D07815857DD45716/SHA1-Impact_V2.0.pdf` Website accessed September 25, 2013, 4 2010.

[130] L. Gong, M. Mueller, and H. Prafullch. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *In Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997.

[131] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java Language Specification Java SE 7 Edition. JSR 901, 7 2011. `http://docs.oracle.com/javase/specs/index.html` Website accessed November 2, 2012.

[132] D. Grawrock. *The Intel Safer Computing Initiative.* Intel Press, 1 edition, 2006. ISBN 0-9764832-6-2.

[133] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach.* Intel Press, February 2009. ISBN 978-1934053171.

[134] H. Guggi and M. Schallar. Trusted computing - mobile attestation token. Class Project Report, Graz University of Technology, 2008.

[135] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga. Security evaluation of scenarios based on the TCG's TPM specification. In *Proceedings of the 12th European conference on Research in Computer Security*, pages 438–453, Dresden, Germany, 2007. Springer-Verlag.

[136] P. Gutmann. An open-source cryptographic coprocessor. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, pages 8–8, Denver, Colorado, 2000. USENIX Association.

[137] G. Hancke. A practical relay attack on ISO 14443 proximity card. Technical report, University of Cambridge, 2005.

[138] G. Hancke. Eavesdropping Attacks on High-Frequency RFID Tokens. In *Workshop on RFID Security 2008 (RFIDSec08), July 9-11, Budapest, Hungary*, volume RFIDsec 2008, pages 100–113, July 2008.

[139] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[140] E. Haselsteiner and K. Breitfuss. Security in near field communication (NFC). In *Workshop on RFID Security*, 2006.

[141] D. Hein and R. Toegl. An autonomous attestation token to secure mobile agents in disaster response. In A. Schmidt and S. Lian, editors, *Security and Privacy in Mobile Information and Communication Systems (MobiSec 2009)*, volume 17 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 46–57, Turin, Italy, June 2009. Springer Berlin/Heidelberg.

[142] D. Hein, R. Toegl, and S. Kraxberger. An autonomous attestation token to secure mobile agents in disaster response. *Security and Communication Networks*, 3(5):421–438, 2010.

[143] D. Hein, R. Toegl, M. Pirker, E. Gatial, Z. Balogh, H. Brandl, and L. Hluchý. Securing mobile agents for crisis management support. In *Proceedings of the seventh ACM workshop on Scalable Trusted Computing*, STC '12, pages 85–90, New York, NY, USA, 2012. ACM.

[144] M. Hermanowski and E. Tews. TPM4JAVA. Currently only available through `http://web.archive.org/web/20090510093615/http://tpm4java.datenzone.de/trac`, 2009. Website accessed November 6, 2012.

[145] J. Herzog. Applying protocol analysis to security device interfaces. *Security & Privacy, IEEE*, 4(4):84–87, 2006.

[146] L. Hoffman and J. Gibbons. JT Harness. `http://java.net/downloads/jtharness/jt_whitepaper.pdf`, 11 2006. Website accessed November 9, 2012.

[147] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[148] J. H. Huh. *Trustworthy Logging for Virtual Organisations.* PhD thesis, University of Oxford, 2009.

[149] J. H. Huh, H. Kim, J. Lyle, and A. Martin. Achieving attestation with less effort: an indirect and configurable approach to integrity reporting. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pages 31–36, Chicago, Illinois, USA, 2011. ACM.

[150] W. H. Hussin. E-pass using DRM in Symbian v8 OS and TrustZone: Securing vital data on mobile devices. In R. Edwards and P. Coulton, editors, *Mobile Business, International Conference on*, volume 0, pages 14–14, June 2006.

[151] W. H. Hussin, P. Coulton, and R. Edwards. Mobile ticketing system employing trustzone technology. In *Mobile Business, 2005. ICMB 2005. International Conference on*, pages 651–654, 2005.

[152] M. Hutter and R. Toegl. A trusted platform module for near field communication. In *Systems and Networks Communications (ICSNC), 2010 Fifth International Conference on*, pages 136–141. IEEE, 2010.

[153] M. Hutter and R. Toegl. Touch'n' Trust: An NFC-enabled trusted platform module. *The International Journal on Advances in Security*, 4(1 & 2):131–141, 2011.

[154] M. Hypponen. Malware Goes Mobile. *Scientific American*, 295:70–77, 2006.

[155] IAIK/Stiftung SIC. IAIK Provider for the Java Cryptography Extension (IAIK-JCE). `http://jce.iaik.tugraz.at/sic/Products/Core-Crypto-Toolkits/JCA-JCE` (23 February 2011), 2011. Website accessed February 18, 2013.

[156] IBM Corp. TrouSerS - an open-source TCG software stack implementation. `http://trousers.sourceforge.net/`. Website accessed October 30, 2012.

[157] IEEE. IEEE Standard 1363-2000: IEEE Standard Specifications for Public-Key Cryptography. Available online at `http://ieeexplore.ieee.org/servlet/opac?punumber=7168`, 2000.

[158] IEEE. IEEE Standard 1363a-2004: IEEE Standard Specifications for Public-Key Cryptography, Amendment 1: Additional Techniques. Available online at `http://ieeexplore.ieee.org/servlet/opac?punumber=9276`, September 2004.

[159] Intel Corporation. Intel low pin count (LPC) interface specification - revision 1.1. `http://www.intel.com/design/chipsets/industry/lpc.htm` Website accessed August 22,2013, 8 2002.

[160] Intel Corporation. Trusted Boot. `http://sourceforge.net/projects/tboot/`, 2008. Website accessed January 29, 2013.

[161] Intel Corporation. Intel Trusted Execution Technology Software Development Guide. `http://download.intel.com/technology/security/downloads/315168.pdf`, March 2011. Website accessed February 18, 2013.

[162] International Organization for Standardization (ISO). ISO/IEC 7816-4: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 4: Interindustry commands for interchange. Available online at `http://www.iso.org`, 1995.

[163] International Organization for Standardization (ISO). ISO/IEC 14443: Identification Cards - Contactless Integrated Circuit(s) Cards - Proximity Cards, 2000.

[164] International Organization for Standardization (ISO). ISO/IEC 7810: Identification cards – Physical characteristics, 2003.

[165] International Organization for Standardization (ISO). ISO/IEC 14888-3: Information technology – Security techniques – Digital signatures with appendix – Part 3: Discrete logarithm based mechanisms, 2006.

[166] International Organization for Standardization (ISO). ISO/IEC 9594-8:2008: Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks, 2008.

[167] International Organization for Standardization (ISO). ISO/IEC 9899:2011 Information technology — Programming languages — C, December 2011.

[168] N. Ivanov and D. Setrakyan. GridGain. `http://www.gridgain.com` Website accessed June 20, 2013, 2010.

[169] J. Jang, S. Nepal, and J. Zic. A trust enhanced email application using trusted computing. In *Ubiquitous, Autonomic and Trusted Computing, 2009. UIC-ATC '09. Symposia and Workshops on*, pages 502–507, 2009.

[170] Java Community Process. JCP procedures overview. `http://jcp.org/en/procedures/overview`. (N.B. For JSR 321, version 2.6 applied.) Website accessed November 12, 2012.

[171] Y. Jianhong and P. Xinguang. Protocol for dynamic component-property attestation in trusted computing. In *Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on*, volume 2, pages 369–372, 2010.

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

[173] S. Katzenbeisser, K. Kursawe, and F. Stumpf. Revocation of TPM keys. In L. Chen, C. Mitchell, and A. Martin, editors, *Lecture Notes in Computer Science*, volume 5471, pages 120–132. Springer Berlin Heidelberg, 2009.

[174] B. Kauer. OSLO: improving the security of trusted computing. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–9, Berkeley, CA, USA, 2007. USENIX Association.

[175] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), Dec. 2005. Updated by RFC 6040.

[176] A. Kerckhoffs. La cryptographie militaire. In *Journal des sciences militaires*, volume IX, 1883.

[177] I. Khan, H. Rehman, and Z. Anwar. Design and deployment of a trusted eucalyptus cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 380–387, 2011.

[178] Z. Khattak, S. Sulaiman, and J. Manan. Security, trust and privacy (STP) framework for federated single sign-on environment. In *Information Technology and Multimedia (ICIM), 2011 International Conference on*, pages 1–6, 2011.

[179] J. King-Lacroix and A. Martin. Bottlecap: a credential manager for capability systems. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 45–54, Raleigh, North Carolina, USA, 2012. ACM.

[180] R. Korn, N. Kuntze, and J. Repp. Performance evaluation in trust enhanced decentralised content distribution networks. In *Communications Quality and Reliability (CQR), 2011 IEEE International Workshop Technical Committee on*, pages 1–6, 2011.

[181] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS ’09, pages 104–115, New York, NY, USA, 2009. ACM.

[182] F. J. Krautheim. Private virtual infrastructure for cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 5–5. USENIX Association, 2009.

[183] F. J. Krautheim, D. Phatak, and A. Sherman. Introducing the trusted virtual environment module: A new mechanism for rooting trust in cloud computing. In A. Acquisti, S. Smith, and A.-R. Sadeghi, editors, *Lecture Notes in Computer Science*, volume 6101, pages 211–227. Springer Berlin Heidelberg, 2010.

[184] S. Kraxberger, R. Toegl, M. Pirker, E. P. Guijarro, and G. G. Millan. Trusted identity management for overlay networks. In R. H. Deng and T. Feng, editors, *Information Security Practice and Experience. 9th International Conference, ISPEC 2013, Lanzhou, China, May 12-14, 2013. Proceedings*, volume 7863 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2013.

[185] 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, pages 50–57, Alexandria, Virginia, USA, 2007. ACM.

[186] K. Kursawe, D. Schellekens, and B. Preneel. Analyzing trusted platform communication. In *In: ECRYPT Workshop, CRASH âĂŞ CRyptographic Advances in Secure Hardware*, page 8, 2005. http://www.cosic.esat.kuleuven.be/publications/article-591.pdf Website accessed October 18, 2013.

[187] Laboratoire Spécification et Vérification. SPORE - Security Protocols Open Repository. http://www.lsv.ens-cachan.fr/Software/spore/index.html, 2003. Website accessed August 22, 2012.

[188] A. Lackorzynski, T. Frenzel, and M. Roitzsch. D2.6 first initial proof of concept for trust-enhanced virtualisation system. TECOM Project, 23 June 2009.

[189] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[190] A. H. Y. Leung. *Securing Mobile Ubiquitous Services using Trusted Computing.* PhD thesis, also published as technical report RHUL-MA-2009-17, Royal Holloway, University of London, 2009. http://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-17.pdf.

[191] F. Li, W. Wang, J. Ma, and Z. Ding. Enhanced architecture of TPM. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 1532–1537, 2008.

[192] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, 1995.

[193] A. Lin. Automated analysis of security APIs. Master's thesis, Massachusetts Institute of Technology, 2005. http://groups.csail.mit.edu/cis/theses/amerson-masters.ps.

[194] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java Virtual Machine Specification Java SE 7 Edition. JSR 924, 7 2011. http://docs.oracle.com/javase/specs/index.html Website accessed November 2, 2012.

[195] F. Lindner. Toying with barcodes. 24th Chaos Communication Congress, 2007. http://events.ccc.de/congress/2007/Fahrplan/events/2273.en.html Website accessed October 18, 2013.

[196] P. Lipp, J. Farmer, D. Bratko, W. Platzer, and A. Sterbenz. *Sicherheit und Kryptographie in Java.* Addison-Wesley Verlag, 2000. ISBN 3827315670. In German.

[197] H. Löhr, H. Ramasamy, A.-R. Sadeghi, S. Schulz, M. Schunter, and C. Stüble. Enhancing grid security using trusted virtualization. In *In 4th International Conference on Autonomic and Trusted Computing (ATC 2007), Hong Kong, China, July 11-13, 2007, Proceedings*, volume 4610 of *Lecture Notes in Computer Science*, pages 372–384. Springer, 2007.

[198] J. C. López Pimentel and R. Monroy. Formal support to security protocol development: A survey. *ComputaciÃşn y Sistemas*, 12:89–108, 2008.

[199] G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.

[200] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.

[201] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, Jan. 1998.

[202] J. Lyle. *Trustworthy Services Through Attestation.* PhD thesis, University of Oxford, 2009.

[203] 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*, pages 283–288. IEEE Computer Society, 2009.

[204] J. Lyle and A. Martin. Trusted computing and provenance: Better together. In *Proceedings of the 2nd conference on Theory and practice of provenance.* USENIX Association, 2010.

[205] R. MacDonald, S. Smith, J. Marchesini, and O. Wild. Bear: An Open-Source Virtual Secure Coprocessor based on TCPA. Technical Report TR2003-471, Dartmouth College, 2003.

[206] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards.* Springer, 2007. ISBN 978-0-387-30857-9.

[207] W. Mao. *Modern Cryptography - Theory and Practice.* Prentice Hall, 1 edition, 2004. ISBN 0-13-066943-1.

[208] W. Mao, A. Martin, H. Jin, and H. Zhang. Innovations for grid security from trusted computing. In B. Christianson, B. Crispo, J. A. Malcolm, and M. Roe, editors, *Security Protocols. 14th International Workshop, Cambridge, UK, March 27-29, 2006, Revised Selected Papers*, volume 5087 of *Lecture Notes in Computer Science*, pages 132–149. Springer-Verlag Berlin, Heidelberg, 2009.

[209] 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, Department of Computer Science/Dartmouth PKI Lab, Dartmouth College, 2003.

[210] P. Marks. Dot-dash-diss: The gentleman hacker's 1903 lulz. *NewScientist*, 2844, 2011.

[211] A. Martin. The ten page introduction to trusted computing. Technical Report RR-08-11, OUCL, December 2008.

[212] T. Mather, S. Kumaraswamy, and S. Latif. *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance.* O'Reilly, 2009. ISBN: 978-0-596-80276-9.

[213] 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*, May 2010.

[214] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCBminimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, Glasgow, Scotland UK, 2008. ACM.

[215] J. McCune, A. Perrig, and M. Reiter. Seeing-is-believing: using camera phones for human-verifiable authentication. In *Security and Privacy, 2005 IEEE Symposium on*, pages 110–124, 2005.

[216] J. McCune, A. Perrig, A. Seshadri, and L. van Doorn. Turtles all the way down: Research challenges in user-based attestation. In *Proceedings of the Workshop on Hot Topics in Security (HotSec)*, August 2007.

[217] C. Meadows. Formal verification of cryptographic protocols: A survey. In *Proceedings of the 4th International Conference on the Theory and Applications of Cryptology: Advances in Cryptology*, pages 135–150. Springer-Verlag, 1995.

[218] C. Meadows. The NRL protocol analyzer: An overview. *The Journal of Logic Programming*, 26(2):113–131, Feb. 1996.

[219] C. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *Selected Areas in Communications, IEEE Journal on*, 21(1):44–54, 2003.

[220] A. J. Menezes, S. A. Vanstone, and P. C. van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., 1996.

[221] Merriam-Webster. On-line dictionary. http://www.merriam-webster.com/. Website accessed March 8, 2012.

[222] Microsoft Developer Network. Overview of the .NET framework. http://msdn.microsoft.com/en-us/library/zw4w595w.aspx. Website accessed November 1, 2012.

[223] Microsoft Developer Network. TPM Base Services. http://msdn.microsoft.com/en-us/library/aa446796(VS.85).aspx. Website accessed October 30, 2012.

[224] J. Millen, S. C. Clark, and S. B. Freeman. The interrogator: Protocol security analysis. *IEEE Trans. Softw. Eng.*, 13(2):274–288, 1987.

[225] Miniwatts Marketing Group. Internet world stats. http://www.internetworldstats.com/stats.htm, 2012. Website accessed March 8, 2012.

[226] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murphi. *Security and Privacy, 1997. Proc., 1997 IEEE Symposium on*, pages 141–151, May 1997.

[227] S. Mödersheim. Abstraction by set-membership: verifying security protocols and web services with databases. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 351–360, Chicago, Illinois, USA, 2010. ACM.

[228] S. Mödersheim and L. Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, Lecture Notes in Computer Science, pages 166–194. Springer-Verlag, 2009.

[229] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, Apr. 1965.

[230] J. Munilla and A. Peinado. Distance bounding protocols for rfid enhanced by using void-challenges and analysis in noisy channels. *Wirel. Commun. Mob. Comput.*, 8(9):1227–1232, 2008.

[231] National Bureau of Standards. Data Encryption Standard. FIPS-Pub.46., 1 1977.

[232] National Institute of Standards and Technology (NIST). Secure hash standard. FIPS PUB 180-1, 4 1995.

[233] National Institute of Standards and Technology (NIST). FIPS-186-2: Digital Signature Standard (DSS), January 2000.

[234] National Institute of Standards and Technology (NIST). FIPS-197: ADVANCED ENCRYPTION STANDARD (AES), November 2001.

[235] National Institute of Standards and Technology (NIST). Security requirements for cryptographic modules. FIPS PUB 140-3, 9 2009. Draft.

[236] R. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[237] NFC Forum. NFC Forum Type 4 Tag Operation - Technical Specification, March 2007.

[238] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[239] Open_TC Consortium. The Open Trusted Computing project (Open_TC). Currently available only through `http://web.archive.org/web/20110723233118/http://www.opentc.net/`, 2005-2009. Archived website accessed October 30, 2012.

[240] A. Oprea, D. Balfanz, G. Durfee, and D. K. Smetters. Securing a remote terminal application with a mobile trusted device. In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 438–447. IEEE Computer Society, 2004.

[241] OPSWAT. Security industry market share analysis. Technical report, OPSWAT Inc., June 2011.

[242] Oracle. Java Native Interface Specification. `http://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html` Website accessed August 23, 2013, 2011.

[243] Oracle. About Java. `http://www.java.com/en/about/`, 2012. Website accessed November 14, 2012.

[244] OSI. Information Technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. ISO/IEC 7498-1:1994, ISO, Geneva, Switzerland, Nov. 1994.

[245] A. T. Othman, S. Khan, M. Nauman, and S. Musa. Towards a high-level trusted computing API for android software stack. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, pages 1–9, Kota Kinabalu, Malaysia, 2013. ACM.

[246] B. Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd conference on Hot topics in security*, pages 1–6, San Jose, CA, 2008. USENIX Association.

[247] B. Parno, J. Lorch, J. Douceur, J. Mickens, and J. McCune. Memoir: Practical state continuity for protected modules. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 379–394, 2011.

[248] B. Parno, J. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers.* Springer Publishing Company, Incorporated, 2011.

[249] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Comput. Secur.*, 6(1-2):85–128, 1998.

[250] S. Pearson, M. Mont, and S. Crane. Persistent and dynamic trust: Analysis and the related impact of trusted platforms: Trust management. In P. Herrmann, V. Issarny, and S. Shiu, editors, *Trust Management, Third International Conference, iTrust 2005, Paris, France, May 23-26, 2005. Proceedings*, volume 3477 of *Lecture Notes in Computer Science*, pages 407–412. Springer Berlin / Heidelberg, 2005.

[251] B. Pfitzmann, J. Riordan, C. Stueble, M. Waidner, and A. Weber. The perseus system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, 2001.

[252] M. Pirker and R. Toegl. Sichere Softwaremodule durch Einsatz von Virtualisierung und Trusted Computing. In W. Seböck and E. Huber, editors, *Tagungsband der 7. Information Security Konferenz*. Österreichische Computer Gesellschaft, 2009. ISBN 978-3-85403-257-1. In German.

[253] M. Pirker and R. Toegl. Towards a virtual trusted platform. *Journal of Universal Computer Science*, 16(4):531–542, 2010. `http://www.jucs.org/jucs_16_4/towards_a_virtual_trusted`.

[254] M. Pirker, R. Toegl, and M. Gissing. Dynamic enforcement of platform integrity (a short paper). In A. Acquisti, S. Smith, and A.-R. Sadeghi, editors, *Trust '10: Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, volume 6101 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010.

[255] M. Pirker, R. Toegl, D. Hein, and P. Danner. A PrivacyCA for anonymity and trust. In L. Chen, C. J. Mitchell, and A. Martin, editors, *TRUST 2009: Proceedings of the 2nd International Conference on Trusted Computing*, volume 5471 of *Lecture Notes in Computer Science*, pages 101–119. Springer Berlin / Heidelberg, 2009.

[256] M. Pirker, R. Toegl, and G. Lindsberger. acTvSM Deliverable 2.1: Requirements Specification Report. Technical report, Graz University of Technology, 2010.

[257] M. Pirker, R. Toegl, and A. Niederl. acTvSM Deliverable 3.1: Virtual Platform Prototype. Technical report, Graz University of Technology, 2010.

[258] M. Pirker, R. Toegl, T. Winkler, and T. Vejda. Trusted computing for the Java^{TM} platform, 2009. Website accessed January 29, 2013.

[259] M. Pirker, J. Winter, and R. Toegl. Lightweight distributed attestation for the cloud. In F. Leymann, I. Ivanov, M. van Sinderen, and T. Shan, editors, *CLOSER*, pages 580–585. SciTePress, 2012.

[260] M. Pirker, J. Winter, and R. Toegl. Lightweight distributed heterogeneous attested android clouds. In S. Katzenbeisser, E. Weippl, L. Camp, M. Volkamer, M. Reiter, and X. Zhang, editors, *Lecture Notes in Computer Science*, volume 7344, pages 122–141. Springer Berlin Heidelberg, 2012.

[261] S. Podesser. Trust in distributed networks. Master's thesis, Graz University of Technology, 2012.

[262] 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, pages 45–53. Springer Berlin Heidelberg, 2011.

[263] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[264] J. Poritz. Trust[ed in] computing, signed code and the heat death of the internet. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1855–1859, Dijon, France, 2006. ACM.

[265] J. Poritz, M. Schunter, E. Herreweghen, and M. Waidner. Property attestation: Scalable and privacy-friendly security assessment of peer computers. Technical report, IBM Research, 2004.

[266] U. W. Ravi Sahita and P. Dewan. Dynamic software application protection. Technical report, Intel Corporation, 2009.

[267] J. Reid, J. M. G. Nieto, T. Tang, and B. Senadji. Detecting relay attacks with timing-based protocols. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 204–213, Singapore, 2007. ACM.

[268] F. Reimair. Trusted virtual security module - design and implementation. Master's thesis, Graz, University of Technology, 2011.

[269] F. Reimair and R. Toegl. acTvSM Deliverable 5.1: TvSM Prototype. Technical report, Graz University of Technology, 2011.

[270] A. Reiter, G. Neubauer, M. Kapfenberger, J. Winter, and K. Dietrich. Seamless integration of trusted computing into standard cryptographic frameworks. In *Proceedings of the Second international conference on Trusted Systems*, pages 1–25, Beijing, China, 2011. Springer-Verlag.

[271] H. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[272] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[273] V. Roubtsov. *EMMA User Guide*, 2006. http://emma.sourceforge.net/userguide/userguide.htmlWebsite accessed November 9, 2012.

[274] D. Rousseau, S. Sitkin, R. Burt, and C. Camerer. Not so different after all: a cross-discipline view of trust. *Academy of Management Review*, 23(3):393–404, July 1998.

[275] RSA Laboratories. PKCS #11 v2.20: Cryptographic Token Interface Standard. RSA Security Inc. Public-Key Cryptography Standards (PKCS), June 2004. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf Website accessed January 29, 2013.

[276] J. Rushby. Design and verification of secure systems. *ACM Operating Systems Review*, 15(5):12–21, 12 1981.

[277] M. Ryan. Cloud computing privacy concerns on our doorstep. *Commun. ACM*, 54(1):36–38, 2011.

[278] 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, pages 67–77, New York, NY, USA, 2004. ACM.

[279] 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*, pages 16–16, San Diego, CA, 2004. USENIX Association.

[280] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, 1975.

[281] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 3–3. USENIX, 2009. https://www.usenix.org/legacy/event/hotcloud09/tech/ Website accessed June 24, 2013.

[282] L. Sarmenta, J. Rhodes, and T. Müller. TPM/J Java-based API for the Trusted Platform Module. http://projects.csail.mit.edu/tc/tpmj/, 4 2007. Website accessed October 30, 2012.

[283] L. Sarmenta, M. van Dijk, C. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 27–42. ACM, 2006.

[284] 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.

[285] R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE DOI - 10.1109/6.591665*, 34(6):52–59, 1997.

[286] J. Schiffman, T. Moyer, C. Shal, T. Jaeger, and P. McDaniel. Justifying integrity using a virtual machine verifier. In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference*, pages 83–92, Washington, DC, USA, 2009. IEEE Computer Society.

[287] 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*, pages 43–46, Chicago, Illinois, USA, 2010. ACM.

[288] M. Schlüter. Realisierung einer mobilen, vertrauenswürdigen Geschäftsplattform auf Basis von Trusted Computing zur gesicherten Datenerfassung. Master's thesis, Technischen Hochschule Mittelhessen, 2012. In German.

[289] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 78–94, 2012.

[290] I. Schnepp, S. Panenka, and M. Richard-Foy. JSR 321 feed-back from TECOM-FP7âĂŹs implementation. Technical report, Atego, 2010. Rev. 2.1.

[291] M. Selhorst, C. Stueble, and F. Teerkorn. TSS Study. Study on behalf of the german federal office for information security (BSI), Sirrix AG security technologies, May 2008. `http://www.sirrix.com/media/downloads/57653.pdf,download` Website accessed November 1, 2012.

[292] P. E. Sevinç, M. Strasser, and D. Basin. Securing the distribution and storage of secrets with trusted platform modules. In *Proceedings of the 1st IFIP TC6 /WG8.8 /WG11.2 international conference on Information security theory and practices: smart cards, mobile and ubiquitous computing systems*, pages 53–66, Heraklion, Crete, Greece, 2007. Springer-Verlag.

[293] R. Sharp, J. Scott, and A. R. Beresford. Secure mobile computing via public terminals. In *Proceedings of the 4th international conference on Pervasive Computing*, pages 238–253, Dublin, Ireland, 2006. Springer-Verlag.

[294] Z. Shen, L. Li, F. Yan, and X. Wu. Cloud computing system based on trusted computing platform. In *Intelligent Computation Technology and Automation (ICICTA), 2010 International Conference on*, volume 1, pages 942–945, 2010.

[295] E. Shi, A. Perrig, and L. van Doorn. BIND: a fine-grained attestation service for secure distributed systems. In *2005 IEEE Symposium on Security and Privacy*, pages 154–168, 2005.

[296] R. Shim, T. Mainelli, B. O'Donnell, C. Chute, F. Pulskamp, and S. Rau. Worldwide interfaces and technologies embedded in PCs 2010-âĂŞ2014 forecast. Technical report, IDC, 2010.

[297] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *EuroSys '06: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 161–174, New York, NY, USA, 2006. ACM.

[298] N. Smart. *Cryptography, An Introduction.* Published by the author at `http://www.cs.bris.ac.uk/~nigel/Crypto_Book/` Website accessed August 21, 2013., 3rd edition, 2013.

[299] M. Smith, T. Friese, M. Engel, and B. Freisleben. Countering security threats in service-oriented on-demand grid computing using sandboxing and trusted computing techniques. *J. Parallel Distrib. Comput.*, 66(9):1189–1204, 2006.

[300] S. Smith. *Trusted Computing Platforms: Design and Applications.* Springer Verlag, 2005.

[301] S. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Comput. Netw.*, 31:831–860, April 1999.

[302] D. X. Song. Athena: a new efficient automatic checker for security protocol analysis. In *Computer Security Foundations Workshop, 1999. Proceedings of the 12th IEEE*, pages 192–202, 1999.

[303] E. Sparks. TPM reset attack. `http://www.cs.dartmouth.edu/~pkilab/sparks/` Website accessed August 23, 2013, 2007.

[304] D. A. Stainforth, T. Aina, C. Christensen, M. Collins, N. Faull, D. J. Frame, J. A. Kettleborough, S. Knight, A. Martin, J. M. Murphy, C. Piani, D. Sexton, L. A. Smith, R. A. Spicer, A. J. Thorpe, and M. R. Allen. Uncertainty in predictions of the climate response to rising levels of greenhouse gases. *Nature*, 433(7024):403–406, Jan. 2005.

[305] L. Sterling and E. Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques.* MIT Press, 1994.

[306] C. Strachey. Time sharing in large, fast computers. In *IFIP Congress*, 1959.

[307] M. Strasser and H. Stamer. A software-based trusted platform module emulator. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Lecture Notes in Computer Science*, volume 4968, pages 33–47. Springer Berlin Heidelberg, 2008.

[308] C. Stueble and A. Zaerin. µTSS - a simplified trusted software stack. In *Proc. 3rd International Conference on Trust and Trustworthy Computing (TRUST 2010)*, number 6101 in LNCS. Springer Verlag, 2010.

[309] C. Stueble and A. Zaerin. µTSS - a simplified trusted software stack. Technical report, Sirrix AG, 2010.

[310] F. Stumpf, M. Benz, M. Hermanowski, and C. Eckert. An approach to a trustworthy system architecture using virtualization. In B. Xiao, L. Yang, J. Ma, C. Muller-Schloer, and Y. Hua, editors, *Lecture Notes in Computer Science*, volume 4610, pages 191–202–. Springer Berlin Heidelberg, 2007.

[311] F. Stumpf, O. Tafreschi, P. Röder, and C. Eckert. A robust integrity reporting protocol for remote attestation. In *Proceedings of the Second Workshop on Advances in Trusted Computing*, 2006.

[312] Sun Microsystems. TCK Project Planning and Development Guide. `http://docs.oracle.com/javame/test-tools/jctt/tck_project_planning_guide.pdf`, 8 2003. Website accessed November 9, 2012.

[313] P. Syverson. A taxonomy of replay attacks [cryptographic protocols]. In *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*, pages 187–191, 1994.

[314] T. Tanveer, M. Alam, and M. Nauman. Scalable remote attestation with privacy protection. In L. Chen and M. Yung, editors, *Lecture Notes in Computer Science*, volume 6163, pages 73–87. Springer Berlin Heidelberg, 2010.

[315] C. Tarnovsky. Hacking the smartcard chip. `http://www.blackhat.com/html/bh-dc-10/bh-dc-10-archives.html`, 2010.

[316] TECOM Consortium. Trusted Embedded Computing project (TECOM). Currently available only through `http://web.archive.org/web/20100625044259/http://www.tecom-project.eu/`, 2008-2010. Website accessed November 9, 2012.

[317] The Trusted Computing Platform Alliance. Building a foundation of trust in the pc. TCPA, 1 2000. `http://perso.telecom-paristech.fr/~guilley/enseignement/projets/crypto_ethique/tcpa/TCPA_first_WP.pdf` Website accessed October 18, 2013.

[318] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27:761–763, 1984.

[319] R. Toegl. OpenTC WP3 Report: Java-API Standardization. Technical Report D03.d7, Graz University of Technology, 2009. `www.opentc.net` Website accessed March 13, 2013.

[320] R. Toegl. Tagging the turtle: Local attestation for kiosk computing. In J. H. Park, H.-H. Chen, M. Atiquzzaman, C. Lee, T.-H. Kim, and S.-S. Yeo, editors, *Advances in Information Security and Assurance*, volume 5576 of *Lecture Notes in Computer Science*, pages 60–69. Springer Berlin / Heidelberg, 2009.

[321] R. Toegl. acTvSM Deliverable 4.1.1: API Analysis Scope. Technical report, Graz University of Technology, 2010.

[322] R. Toegl. acTvSM Deliverable 4.3: Verification Report. Technical report, Graz University of Technology, 2011.

[323] R. Toegl. Verification of a trusted virtual security module. In M. Bond, R. Focardi, F. Sibylle, and G. Steel, editors, *Analysis of Security APIs (Dagstuhl Seminar 12482)*, Dagstuhl Reports, page 166. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013. Abstract.

[324] R. Toegl, U. Birnbacher, and O. Koudelka. Deploying IP telephony over satellite links. In *Wireless Communication Systems, 2005. 2nd International Symposium on*, pages 624–628, 2005.

[325] R. Toegl, G. Hofferek, K. Greimel, A. H. Y. Leung, R.-W. Phan, and R. Bloem. Formal analysis of a TPM-based secrets distribution and storage scheme. In *Proceedings TRUSTCOM 2008, in: Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 2289–2294. IEEE Computer Society, 2008.

[326] R. Toegl and M. Hutter. An approach to introducing locality in remote attestation using near field communications. *The Journal of Supercomputing*, 55(2):207–227, 2011.

[327] R. Toegl, P. Lipp, J. Nisewanger, D. D. Rao, T. Winkler, W. Keil, T. Hong, M. Nauman, B. Gungoren, and K. M. Graf. JSR 321 Trusted Computing API for Java. Java Community Process Specification Final Release `http://jcp.org/en/jsr/detail?id=321`, 12 2011. Java Specification Request # 321. Website accessed October 31, 2012.

[328] R. Toegl, C. Parraga Niebla, and U. Birnbacher. Framing efficiency optimization for DVB-S2 systems. In *Global Telecommunications Conference, 2006. GLOBECOM '06. IEEE*, 2006.

[329] R. Toegl, C. Parraga Niebla, and U. Birnbacher. Framing efficiency optimization for DVB-S2 systems with QoS guarantees. In *Ka-band Conference 2006. Ka and Broadband Communications Conference, Proceedings*, Naples, Italy, 2006.

[330] 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*, pages 60–67. Vieweg+Teubner, 2009.

[331] R. Toegl, M. Pirker, R. Bloem, G. Lindsberger, and S. Posch. acTvSM Deliverable 6.1: Business Model and Product Roadmap. Technical report, Graz University of Technology, 2011.

[332] R. Toegl, M. Pirker, and G. Fliess. acTvSM Deliverable 4.1: Draft API Design. Technical report, Graz University of Technology, 2010.

[333] 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*, volume 6802 of *Lecture Notes in Computer Science*, pages 326–345. Springer Berlin / Heidelberg, 2011.

[334] R. Toegl, M. Pirker, A. Niederl, and M. Gissing. acTvSM Deliverable 3.2: Virtual Trusted Platform Prototype. Technical report, Graz University of Technology, 2010.

[335] R. Toegl and F. Reimair. acTvSM Deliverable 4.2: Revised API Design. Technical report, Graz University of Technology, 2011.

[336] 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*, pages 145–160. Springer Berlin Heidelberg, 2013.

[337] R. Toegl and M. Steurer. Open_TC WP3 report: Java API and library implementation. Technical Report D03.d5, Graz University of Technology, 2008. `www.opentc.net` Website accessed March 13, 2013.

[338] R. Toegl, T. Winkler, M. Nauman, and T. Hong. Towards platform-independent trusted computing. In *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing*, pages 61–66, Chicago, Illinois, USA, 2009. ACM.

[339] R. Toegl, T. Winkler, M. Nauman, and T. W. Hong. Specification and Standardization of a Java Trusted Computing API. *Softw. Pract. Exper.*, 42(8):945–965, 2012.

[340] R. Toegl, T. Winkler, M. Pirker, M. Steurer, and R. Stoegbuchner. IAIK Java TCG Software Stack - jTSS API Tutorial. `http://trustedjava.sf.net`, 2011. Website accessed November 14, 2012.

[341] R. Toegl, J. Winter, and M. Pirker. A path towards ubiquitous protection of media. In J. Lyle, S. Faily, and M. Winandy, editors, *Proceedings of the Workshop on Web Applications and Secure Hardware (WASH), Co-located with the 6th International Conference on Trust and Trustworthy Computing (TRUST 2013)*, volume 1011 of *CEUR Workshop Proceedings*, pages 32–38, London, United Kingdom, 6 2013. Sun SITE Central Europe, RWTH Aachen University. Position Paper.

[342] Trusted Computing Group. TCG Software Stack (TSS) Specification Version 1.2 Level 1 Errata A. `http://www.trustedcomputinggroup.org/`

resources/tcg_software_stack_tss_specification, 3 2007. Website accessed January 29, 2013.

[343] Trusted Computing Group. TCG Mobile Trusted Module Specification version 1.0 revision 7.02. http://www.trustedcomputinggroup.org/developers/mobile/specifications, 4 2010. Website accessed January 29, 2013.

[344] Trusted Computing Group. TCG PC Client Specific TPM Interface Specification (TIS) specification version 1.21 revision 1.00. http://www.trustedcomputinggroup.org/resources/pc_client_work_group_pc_client_specific_tpm_interface_specification_tis, 4 2011. Website accessed January 29, 2013.

[345] Trusted Computing Group. TCG TPM specification version 1.2 revision 116. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 3 2011. Website accessed January 29, 2013.

[346] Trusted Computing Group. TCG Website. https://www.trustedcomputinggroup.org/, 2013. Website accessed October 30, 2012.

[347] Y.-J. Tu and S. Piramuthu. RFID distance bounding protocols. In *First International EURASIP Workshop on RFID Technology*, 2007. http://www.eurasip.org/Proceedings/Ext/RFID2007/pdf/s5p2.pdf Website accessed October 18, 2013.

[348] J. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. In *Technological Strategies for the Protection of Intellectual Property in the Networked Multimedia Environment*, pages 121–152. Interactive Multimedia Association, 1994.

[349] R. van Rijswijk-Deij and E. Poll. Using trusted execution environment in two-factor authentication: comparing approaches. In *Open Identity Summit 2013 (OID2013)*, Kloster Banz, Germany, September 2013. Accepted for publication. Draft available from http://www.cs.ru.nl/~rijswijk/pub/TrEE-oids13.pdf.

[350] 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 2010)*, volume 6101 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, June 2010.

[351] A. Vasudevan, B. Parno, N. Qu, V. Gligor, and A. Perrig. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In *TRUST*, volume 7344 of *Lecture Notes in Computer Science*, pages 34–54. Springer, 2012.

[352] S. Vaughan-Nichols. How trustworthy is trusted computing? *Computer*, 36(3):18–20, 2003. IEEE.

[353] T. Vejda, R. Toegl, M. Pirker, and T. Winkler. Towards trust services for language-based virtual machines for grid computing. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Trusted Computing âĂŞ Challenges and Applications First International Conference on Trusted Computing and Trust in Information Technologies, TRUST 2008 Villach, Austria, March 11-12, 2008 Proceedings*, volume 4968 of *Lecture Notes in Computer Science*. Springer Verlag, 2008.

[354] C. Vishik, A. Rajan, C. Ramming, D. Grawrock, and J. Walker. Defining trust evidence: research directions. In *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research*, pages 1–1, Oak Ridge, Tennessee, 2011. ACM.

[355] W3C XML Protocol Working Group. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, W3C, 2007.

[356] 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*, volume 6531 of *Lecture Notes in Computer Science*, pages 84–98. Springer Berlin / Heidelberg, 2011.

[357] 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 *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 247–254, 2011.

[358] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In V. Shoup, editor, *Lecture Notes in Computer Science*, volume 3621, pages 17–36. Springer Berlin Heidelberg, 2005.

[359] R. Want. Near field communication. *Pervasive Computing, IEEE*, 10(3):4–7, 2011.

[360] T. Winkler and B. Rinner. User-based attestation for trustworthy visual sensor networks. In *Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC), 2010 IEEE International Conference on*, pages 74–81, 2010.

[361] J. Winter. Trusted computing building blocks for embedded linux-based ARM TrustZone platforms. In *Proceedings of the 3rd ACM workshop on Scalable Trusted Computing*, pages 21–30, Alexandria, Virginia, USA, 2008. ACM.

[362] J. Winter and K. Dietrich. A hijacker's guide to the LPC bus. In S. Petkova-Nikova, A. Pashalidis, and G. Pernul, editors, *EuroPKI*, volume 7163 of *Lecture Notes in Computer Science*, pages 176–193. Springer, 2011.

[363] J. Winter and K. Dietrich. A hijackerâĂŹs guide to communication interfaces of the trusted platform module. *Computers & Mathematics with Applications*, 65(5):748–761, Mar. 2013.

[364] 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*, pages 1–15, Beijing, China, 2012. Springer-Verlag.

[365] R. Wojtczuk and J. Rutkowska. Attacking Intel Trusted Execution Technology. Technical report, Invisible Things Lab, 2009.

[366] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another way to circumvent Intel Trusted Execution Technology technology: Tricking SENTER into misconfiguring VT-d via SINIT bug exploitation. Technical report, Invisible Things Lab, 2009.

[367] W. Xingkui and P. Xinguang. The trusted computing environment construction based on jTSS. In *Mechatronic Science, Electric Engineering and Computer (MEC), 2011 International Conference on*, pages 2252–2256. IEEE, 2011.

[368] P. Xinguang and J. Wei. Filter-based trusted remote attestation for web services. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 3, pages 5–9, 7 2010.

[369] J. Yan and X. Peng. Security strategy of DRM based on trusted computing. *Journal of Computational Information Systems*, 9(7):3226–3234, 2011.

[370] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006.

[371] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. L. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical report, University of Cambridge, 2005.

[372] Y. M. Yussoff and H. Hashim. Trusted wireless sensor node platform. In S. I. Ao, L. Gelman, D. W. Hukins, A. Hunter, and A. M. Korsunsky, editors, *Proceedings of the World Congress on Engineering 2010 Vol I, WCE '10, June 30 - July 2, 2010, London, U.K.*, Lecture Notes in Engineering and Computer Science, pages 774–779. International Association of Engineers, Newswood Limited, 2010.

[373] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18–, 2010.

[374] Y. Zhang, C. Wang, J. Wu, and X. Li. Using SMV for cryptographic protocol analysis: a case study. *SIGOPS Oper. Syst. Rev.*, pages 43–50, 2001.

[375] Z. Zhou, V. Gligor, J. Newsome, and J. McCune. Building verifiable trusted path on commodity x86 computers. In *2012 IEEE Symposium on Security and Privacy*, volume 0, pages 616–630, May 2012.

[376] J. Zic and S. Nepal. Implementing a portable trusted environment. In D. Gawrock, H. Reimer, A.-R. Sadeghi, and C. Vishik, editors, *Future of Trust in Computing*, pages 17–29. Vieweg+Teubner, 2009.

[377] J. Zlattinger. Randomized software testing. Bachelor Project, Graz University of Technology, 2009.

Author Index

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 entnommenen 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)