Department of Computer Science and Engineering

Securing Software Architectures for Trusted Processor Environments

Jan-Erik Ekberg

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 75/2013

Securing Software Architectures for Trusted Processor Environments

Jan-Erik Ekberg, Lic. Sci. (Tech)

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held in the lecture hall T2 of the school on May 24, 2013 at 12.

Aalto University
School of Science
Computer Science and Engineering
Data Communications Software

Supervising professor
Prof. Tuomas Aura

Thesis advisor
Prof. N. Asokan

Preliminary examiners
Dr. Jonathan McCune, Google Inc, U.S.A.
Dr. Mohammad Mannan, Concordia University, Canada

Opponent
Prof. Dr. Jean-Pierre Seifert, Technische Universität Berlin, Germany

Aalto University publication series
DOCTORAL DISSERTATIONS 75/2013

ISBN 978-952-60-5149-9 (printed)
ISBN 978-952-60-3632-8 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-3632-8

Unigrafia Oy
Helsinki 2013

Finland

441 697
Printed matter

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Jan-Erik Ekberg

Name of the doctoral dissertation
Securing Software Architectures for Trusted Processor Environments

Publisher School of Science

Unit Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 75/2013

Field of research Platform Security

Manuscript submitted 23 January 2013 **Date of the defence** 24 May 2013

Permission to publish granted (date) 12 March 2013 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

Processor hardware support for security dates back to the 1970s, and such features were then primarily used for hardening operating systems. This idea has re-emerged as hardware security features in contemporary cost-efficient mobile processors. These support specific operating-system functionality such as communication stack isolation and identity binding, which are needed on mobile devices to satisfy regulatory requirements for e.g. cellular phones.

This thesis builds on these hardware security features to implement a generic trusted execution environment (TEE) that can be used for a larger variety of applications. We present software building blocks and infrastructure for isolated trustworthy execution on these hardware environments. The goal is to achieve the same level of isolation as in smart cards or trusted platform modules implemented as separate integrated circuits. The thesis contributes to the state of the art in several ways: We present mechanisms for isolated piecemeal execution of code and processing of data in these very memory-constrained hardware environments. Isolation, freshness and data commit guarantees are provided by cryptographic means. We present security proofs for selected cryptographic primitives used in this hardware context. The thesis also improves on the integrity guarantees of contemporary processor support by implementing rollback protection even when the device is powered down. This is done by combining the security functionality of the processor with auxilliary hardware and firmware logic. We advance the understanding of trusted execution by describing a minimal set of hardware trust roots needed to implement an engine for isolated execution.

Ideally, advancement of computer science can be translated into implementable designs with real-world impact. The mechanims presented in this thesis were implemented and deployed in the On-board Credentials (ObC) architecture, and partly standardized as features for the Mobile Trusted Module (MTM). These technologies enable implementation of isolated execution at significant cost savings compared to the deployment of discrete hardware components. The MTM specification, co-designed by the author, is the first global security standard that provides an adaptation to processor hardware mechanisms for isolated execution. The TEE part of On-board Credentials, designed and implemented by the author, is deployed in more than 100 million devices in the field, and has already been used in several public trials and demonstrations of end-user applications. Both ObC and MTM rely on the results of this thesis research.

Keywords Platform security, cryptography

ISBN (printed) 978-952-60-5149-9 **ISBN (pdf)** 978-952-60-3632-8

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2013

Pages 230 **spl** ?ĸĸGńŜŜLIEłĮŜ2 / +ń&0 +ńŁĿłŘŁĽÍŘľĸŘĻľĮÍŘłħ

Aalto-universitetet

Sammandrag

Aalto-universitetet, PB 11000, 00076 Aalto www.aalto.fi

Författare
Jan-Erik Ekberg

Doktorsavhandlingens titel
Programvarusystem för säkra processorarkitekturer

Utgivare Högskolan för teknikvetenskaper

Enhet Institutionen för datavetenskap

Seriens namn Aalto University publication series DOCTORAL DISSERTATIONS 75/2013

Forskningsområde Säkerhet

Inlämningsdatum för manuskript 23.01.2013 **Datum för disputation** 24.05.2013

Beviljande av publiceringstillstånd (datum) 12.03.2013 **Språk** Engelska

☐ **Monografi** ☒ **Sammanläggningsavhandling (sammandrag plus separata artiklar)**

Sammandrag

Processorstöd för säkerhet introducerades på 1970-talet, främst för att förbättra operativsystemens intergritet. Med de öppna PC-plattformernas genombrott försvann dessa mekanismer för några tiotal år, men motsvarande mekanismer togs åter i bruk för omkring tio år sedan i mobila hårdvaruplattformer, nu främst för att garantera protokollintegritet för kommunikation och för att binda upp den mobila hårdvarans identitet - typiska villkor för att kunna erhålla t.ex. radiolicens för en mobiltelefon.

Denna avhandling bygger från dessa existerande hårdvarumekanismer och presenterar programvarubyggstenar för att kunna implementera säker, isolerad tolkning av programvara i en arkitektur som externt motsvarar en diskret hårdvarukomponent såsom t.ex. ett smartkort. Avhandlingen bidrar till den senaste kunskapen från många infallsvinklar. Den presenterar mekanismer för isolerad tolkning av programvara och associerad data i stycken i dessa högst begränsade omgivningar, där garantierna för isolation, versionshantering och dataflöde måste byggas upp med kryptografiska metoder. Avhandlingen bidrar också med säkerhetsbevis för valda kryptografiska algoritmer i denna omgivning. Vi förbättrar nivån av off-line integritet med att presentera en lösning där det säkra processorstödet kombineras med extern, diskret logik för att säkra mot rollback. Avhandlingen presenterar även en minimal uppsättning av säkerhetsfundament som en processor måste stöda i hårdvara för att isolerad tolkning skall kunna implementeras. Den beskriver också två arkitekturer som uppbyggts baserat på de byggstenar som presenteras i denna avhandling, och vilka var för sig erbjuder gränssnitt för mobilapplikationer och i sista hand användare.

Sin största verkan får datavetenskapen när den ibruktas medelst implementationer. Byggstenarna som presenteras i denna avhandling möjliggör isolerad programvarutolkning till en betydligt lägre kostnad än vad som är möjligt med diskret hårvara, t.ex. smartkort. Författaren har aktivt bidragit till standarden Mobile Trusted Module (MTM) - den första globala säkerhetsstandarden som definierar och möjliggör en adaptering baserad på isolation byggd utgående från processorer med säkerhetsfunktioner. Säkerhetskärnan i OnBoard Credentials arkitekturen, som planerats och implementerats av författaren, finns tillgänglig i över 100 miljoner mobiltelefoner, och har redan använts i flera publika forskningsprojekt och demonstrationer. Båda dessa arkitekturer baserar sig på metodologi och även programvara som härrör sig från denna avhandling.

Nyckelord Datasäkerhet, kryptering

ISBN (tryckt) 978-952-60-5149-9 **ISBN (pdf)** 978-952-60-3632-8

ISSN-L 1799-4934 **ISSN (tryckt)** 1799-4934 **ISSN (pdf)** 1799-4942

Utgivningsort Esbo **Tryckort** Helsingfors **År** 2013

Sidantal 230 **urn** http://urn.fi/URN:ISBN:978-952-60-3632-8

Acknowledgements

Like most endeavors, this work could never have happened in isolation. I have had the great pleasure to work for a company, Nokia, that has likely been the driving force which caused the reawakening of processor secure environments in their current form in the early 2000's. Work colleagues like Lauri Paatero, Antti Kiiveri and many others did prepare this path for me, and also were the initiators of my first project in this area as early as 2005. From that time kudos also goes to Lauri Tarkkala, who was the Mobile Trusted Module specification editor before me, and who also is the father of some of the wonderful intricacies of that specification.

Most of the published research contributions that form this thesis were preceded by full-scale implementations on real hardware that either served as the reference model for a design, or in one case, the eye-opener for a problem with a cryptographic primitive. Although the author himself did his fair share of this toiling, Aarne Rantala was instrumental as a co-engineer (and often co-architect), and I feel that the general topic would never really have gotten off the ground without his experience and help. On the "OS" side, Dr. Kari Kostiainen was for a long time the only designer and maintainer of the APIs and OS components of the main architecture presented in this thesis, and his contributions to this work speak for themselves as he wrote his doctoral dissertation on the provisioning and management aspects of this very architecture. Sampo Sovio was my oracle for all cryptography-related questions and code problems I encountered during the thesis work. With these kinds of friends anything can be achieved.

I am also indebted to many other collaborators who have contributed in more specific aspects of this overall work. Much of the early MTM work I did together with Markku Kylänpää. Sven Bugiel was one of the most productive summer interns I ever had the pleasure of instructing. Dr. Alexandra Afanasyeva was instrumental in her help with the cryptographic proofs present in this thesis. In the context of this thesis Sandeep Tamrakar is visible only as a co-author for the ticketing application, but our collaboration goes much deeper than that. Thank you all.

My supervisor, Prof. Tuomas Aura, was the one who saw the thread connecting the included publications, produced over a timespan of more than 5 years. He also provided great help with the writing of the extended abstract for this thesis.

Most of all, I feel that my advisor, prof. N. Asokan is the cause and conduit that has lead to the completion of this thesis. He has over the years not only consistently pushed me towards this end goal, he has also incited me to appreciate and understand the state of the research in this field, as well

as painstakingly helped me to build the skill to contribute to the academic community. Naturally his scientific presence is also visible in most of the contributions that form this thesis.

Finally, I wish to thank my wife Marie Selenius and my children Jonathan, Daniel and Nadja for their understanding. Any work towards this thesis was time spent apart.

Contents

Chapter 1

Publications

This thesis presents an extended summary of the following publications:

P1 Jan-Erik Ekberg. Mobile Trusted Computing Based on MTM. International Journal of Dependable and Trustworthy Information Systems (IJDTIS), Volume 1, Issue 4, 2010. 25-42, DOI: 10.4018/IJDTIS

P2 Jan-Erik Ekberg and N. Asokan: External Authenticated Non-volatile Memory with Lifecycle Management for State Protection in Trusted Computing. In Lecture Notes in Computer Science, 2010, Vol 6163/2010, 16-38, DOI: 10.1007/978-3-642-14597-1_2 (InTrust 2009)

P3 Jan-Erik Ekberg and Sven Bugiel. 2009. 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 2009). ACM, New York, NY, USA, 9-18. DOI=10.1145/1655108.1655111

P4 Jan-Erik Ekberg, Alexandra Afanasyeva, N. Asokan: Authenticated Encryption Primitives for Size-Constrained Trusted Computing. In Lecture Notes in Computer Science (TRUST 2012), Volume 7344/2012, 1-18, DOI: 10.1007/978-3-642-30921-2-1

P5 Sven Bugiel and Jan-Erik Ekberg: Implementing an application specific credential platform using late-launched mobile trusted module. In Proceedings of the fifth ACM Workshop on Scalable Trusted Computing, STC 2010, pages 21-33, New York, NY, USA, 2010. DOI=10.1145/1867635.1867641

P6 Jan-Erik Ekberg, N. Asokan, Kari Kostiainen, and Aarne Rantala. 2008. Scheduling execution of credentials in constrained secure environments. In Proceedings of the 3rd ACM workshop on Scalable trusted computing (STC 2008). ACM, New York, NY, USA, 61-70. DOI=10.1145/1456455.1456465

P7 Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. 2009. On-board credentials with open provisioning. In Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS '09). ACM, New York, NY, USA, 104-115. DOI=10.1145/1533057.1533074 http://doi.acm.org/10.1145/1533057.1533074

P8 Jan-Erik Ekberg and Sandeep Tamrakar: Mass Transit Ticketing with NFC Mobile Phones. In Lecture Notes in Computer Science (InTrust 2011), Volume 7222/2012, 48-65, DOI=10.1007/978-3-642-32298-3-4

1.1 Author's contribution

The author was the sole leading contributor in all publications except **P5** and **P7**. In each of those cases, the author was a leading contributor along with another co-author. Publication **P5** is based on Sven Bugiel's master thesis, where the research problem was set and guidance was provided by the author, who was the thesis instructor. In publication **P7**, Dr. Kari Kostiainen contributed the material on the provisioning aspects, whereas the platform isolation and interpreter parts originate from the current author.

In **P4** the security proof with OMAC was constructed by Dr. A Afanasyeva. In publication **P8** the protocols as well as trusted execution environment and smart card work is the author's contribution, whereas the user application development and performance measurements originate from Sandeep Tamrakar.

With the exception of **P3** and **P5**, where large parts of the engineering work was done by Sven Bugiel, **P6**, where part of the implementation was contributed by Aarne Rantala and **P8** for which the division of labor was outlined above, all engineering design and implementation work in conjunction with the trusted environments has been done by the author.

Chapter 2

Other publications

The following publications are not part of the thesis. The first two are technical reports that predate two of the publications included in the dissertation and address the same topics as the publications. Publication 3 describes one more application of the mechanisms included in the thesis. Publication 4 is a collaborative effort on trustworthy mobile OS security, where the introduction to trusted execution environments was provided by the author. Publications 5-9 are related to the thesis topic, but deal primarily with protocol security.

1. J-E Ekberg, N. Asokan, K. Kostiainen, P. Eronen, A. Rantala, and A. Sharma: On-board Credentials Platform: Design and Implementation. Nokia Research Center technical report series, Helsinki, Finland, January 2008. (14 citations)

2. J-E Ekberg and M. Kylänpää. Mobile trusted module (MTM) - an introduction. Technical Report NRC-TR-2007-015, Nokia Research Center, 2007 (42 citations)

3. S Tamrakar, J-E Ekberg, P Laitinen, T Aura: Can Hand-Held Computers Still Be Better Smart Cards? Trusted Systems,Notes in Computer Science ISBN: 978-3-642-25282-2, 200-218, Vol 6802, 2010 (InTrust 2010)

4. Kari Kostiainen, Elena Reshetova, Jan-Erik Ekberg and N. Asokan. Old, New, Borrowed, Blue: Perspective on the Evolution of Mobile Platform Security Architectures. In Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY), pages 13-24, February 2011

5. S Tamrakar, J-E Ekberg, and N Asokan. 2011. Identity verification schemes for public transport ticketing with NFC phones. In Proceedings of the sixth ACM workshop on Scalable trusted computing (STC 2011). ACM, New York, NY, USA, 37-48. DOI=10.1145/2046582.2046591

6. Kari Kostiainen, Alexandra Dmitrienko, Jan-Erik Ekberg, Ahmad-Reza Sadeghi and N. Asokan. Key Attestation from Trusted Execution Environments. In Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST), pages 30-46, June 2010.

7. A-R Sadeghi, M Wolf, C Stuble, N. Asokan and J-E Ekberg: Enabling Fairer Digital Rights Management with Trusted Computing , Lecture Notes in Computer Science, 2007, Volume 4779/2007, 53-70, DOI: 10.1007/978-3-540-75496-1-4

8. Kari Kostiainen, N. Asokan and Jan-Erik Ekberg. Credential Disabling from Trusted Execution Environments. In Proceedings of the Nordic Conference in Secure IT Systems (Nordsec), pages 171-186, October 2010.

9. Kari Kostiainen, N.Asokan and Jan-Erik Ekberg. Practical Property-Based Attestation on Mobile Devices. In Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST), pages 78-92, June 2011.

10. Saxena, N.; Ekberg, J.-E.; Kostiainen, K.; Asokan, N.: Secure device pairing based on a visual channel, Security and Privacy, 2006 IEEE Symposium on , vol., no., pp.6 pp.-313, 21-24 May 2006 doi: 10.1109/SP.2006.35

11. J-E Ekberg: Implementing wibree address privacy. In T Strang A Bajart, H Muller, editor, Ubicomp 2007 Workshop Proceedings Proc, First International Workshop on Security for Spontaneous Interaction, Lecture Notes in Computer Science, pages 481-485, 2007

12. Kokkinen, H.; Ekberg, J.-E.; Nöyranen, J.: Post-Payment System for Peer-to-Peer Filesharing, Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE , vol., no., pp.134-135, 10-12 Jan. 2008 doi: 10.1109/ccnc08.2007.37

13. Saarela, A.; Ekberg, J.-E.; Nyberg, K.: Random Beacon for Privacy and Group Security. Networking and Communications, 2008. WIMOB '08. IEEE International Conference on Wireless and Mobile Computing, vol., no., pp.514-519, 12-14 Oct. 2008 doi: 10.1109/WiMob.2008.91

Chapter 3

Introduction

3.1 Background

Since the introduction of stored-program computing devices (Turing / Eckert / Mauchly machines) in the 1940s, computers are amalgams of hardware and software. They can either be embedded devices that perform a predefined task defined by pre-loaded software (firmware), or general-purpose computers that execute any program loaded onto them. The distinction between embedded and general purpose computers has, however, become increasingly blurred. General-purpose computing hardware is today commonly used also in the embedded domain and software upgrades are typically the norm also for embedded devices. Mobile phones are a good example where a whole industry in essence has moved from manufacturing embedded devices to building general purpose computers.

Platform security is one property that is affected by this overall transition in computing. Application download in combination with the introduction of local and wide area networking to consumer devices makes the reliance on perimeter security on the device insufficient. The current need for reactive mechanisms like virus checkers in PCs and mobile phones is by itself proof that the perimeter of contemporary devices is already too complex to properly secure. We see this as attacks infecting the system over networking protocols, through the kernel system call interface, or viruses spreading between devices using USB sticks or MMC cards.

As a reaction to this trend, hardware manufacturers have added security functionality to their processors to enable the implementation of more secure software systems. With contemporary processor and chipset hardware, manufacturers can do more than solely trust that the software is not tampered with and running properly. Hardware mechanisms can validate software properties and sometimes even enforce a device shutdown in case, for example, a divergence from an approved software configuration is measured. Non-volatile device secrets and external trust roots can be stored on memory within the processor chip itself making them less vulnerable against off-line attacks that target hard drives or flash memory. Memory management units together with new registers implement security domain separation as well as new access control primitives like the no-execute bit for memory pages. In this work we will call such trusted hardware **Processor Secure Environ-**

ments (PSE). The primary motivation for introducing these new primitives is a better, i.e. smaller implementation of the so called trusted computing base (TCB) in the computing device: *The trusted computing base of a computer system is the totality of protection mechanisms within it, including hardware, firmware and software, the combination of which is responsible for enforcing a computer security policy*[27].

When we break down the definition of TCB further, we can identify certain functionality, so called *Roots-of-Trust* (RoT), which typically consist of some minimal hardware and software support enabling a specific security feature. One example is "root of trust for measurement" that stipulates that given an immutable value (a key or a reference hash) in hardware and an immutable processor bootstrap code containing a cryptographic hash algorithm we can launch a TCB or any software whose integrity is ascertained by virtue of applying the hash algorithm to the TCB that boots up and comparing the result of the algorithm against the stored reference. A number of RoTs can be identified, and the TCB is the computing base that performs its intended actions for protecting the system under the governance of all necessary RoTs.

To clarify the need for RoTs, we may refer to communication security and consider the so called Dolev-Yao attacker model[28]. In this model the attacker is considered "all-powerful", i.e. the attacker is free to delete, reorder, modify or add any messages or message parameters during transmission between the communicating endpoints. A TCB is a subset of hardware and software within a device, and an endpoint by itself in the Dolev-Yao. Therefore, the Dolev-Yao attack model can now also be applied within a single device, where one mostly untrusted device can contain several TCB endpoints that need to communicate with each other[1]. The definition of RoTs allows us to explore attack surfaces at an even higher resolution, i.e., the attack point may now even include interfaces between parts of the TCB that might be assumed to be trusted only at certain times of the device lifecycle. These points in time can be manufacturing time or early in the device boot sequence. RoT functions may also be physically separated from the device and only logically tied to the TCB by cryptographic means.

A security property that lies in the heart of this thesis is isolation. Logically this is a simple concept, but one that often is very difficult to achieve. Lampson identified this issue as early as 1973[56], and early computer architectures like Multics[15] and Cambridge CAP[96] did address isolation with hardware features that implemented capabilities. Salzer and Schroeder summarized such hardware "technical underpinnings" of computer protection in 1975[78] and listed the constraining of memory access based on a privilege bit as a fundamental building block of isolated virtual machines. After 35 years this statement is still an accurate assessment.

Today, we also see full hardware isolation used as part of computing devices deployed for security-critical operations. The most widespread category is the smart card, used as a token for subscriber identity in mobile networks, as credit card security element and for a vast variety of different authentication needs for physical or networked access. The smart cards generally

[1]Even if not expressed as a Dolev-Yao setup, the existence of secure sessions in Trusted Platform Modules[94] or GlobalPlatform smart cards[40] point to the assumption that these standardized, embeddable secure environments are designed to operate in devices with other, additional trusted session endpoints.

adhere to the ISO-7816 suite of standards. For wireless near-field communication (NFC) smart cards the relevant interface specification is ISO-14443. Smart cards are programmable, either by proprietary means or by using the JavaCard programming environment. Another example of a device deployed for isolation is the "classic" IBM-4758 cryptographic co-processor[82] (now replaced with the CEX3C/4765 model) extensively used in banking applications. These co-processors are in essence computers with non-volatile storage isolated in a tamper-resistant housing. Both smart cards and cryptographic co-processors are generally certified to Common Criteria security levels and implement protection against physical tampering. Essential to the security of these elements is also the **provisioning** aspect. In a baseline provisioning solution, only programs signed by the owner of the card or co-processor are installable on them. Typically also a security context between the card or co-processor and external servers can be set up to remotely provision external secrets to the code installed into the isolated element.

For computing elements like smart cards and cryptographic co-processors, one distinction that is relevant for security is whether they are physically affixed to a host computing platform or not. Removable smart cards are the norm when the element is used for end user authentication. However, trusted platform modules (TPMs), specified by Trusted Computing Group[94], are isolated elements with the distinct requirement that they are physically affixed to the computing platform. By adding a set of additional trust roots and functionalities, provided by the host platform and external to the isolated element, security services like **remote attestation** of the software and hardware state of the host platform can be achieved by the combination of the isolated computing element and the few added trust roots.

Only non-removable isolated elements, i.e., ones embedded into the host computing device, are considered in this thesis. Also, the emphasis will be on leveraging and extending PSE environments to achieve isolation properties corresponding to those of physically isolated environments like embedded smart cards and cryptographic co-processors.

3.2 The cost of environments realizing a trusted execution environment

Adding security to a computing device induces cost in several ways. If the security component is implemented in hardware, there is a design or purchase, as well as a real estate cost related to the chip on which the security mechanisms have been implemented. If some security mechanisms are to be used only rarely, it makes sense to apply them by making use of computing infrastructure already deployed for some more frequent activity like running the operating system and applications. The monetary cost of security consists of a fixed, up-front development cost as well as variable cost accumulating during deployment, where the end customer will have to pay for any extra hardware embedded into the device. In mass-produced devices, the variable cost is always the dominant concern. In addition to the cost of physical hardware, also unit costs related to manufacturing, testing and care (warranty service) can significantly add up, especially if the implemented security design does not account for these issues.

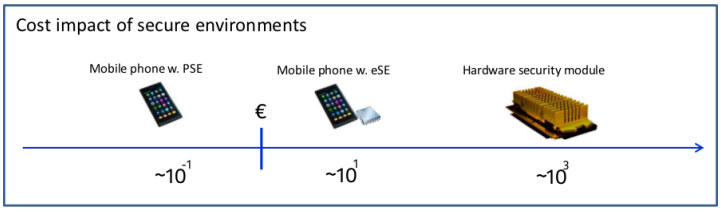

Figure 3.1: Secure environment cost

Energy consumption is a cost that is especially prominent in battery-operated devices and very noticeable to the end user. For security functionality, the energy cost category can be split into computational cost, memory cost and, for radio devices, also the communication cost. Any computation will consume energy, and the resource requirements of security algorithms are often significant compared to normal processing load in a mobile device. The energy consumption of volatile memory, if it needs to be permanently allocated to some security processing, is also easily of the same order of magnitude as the radio communication cost.

Indirect costs of stand-alone hardware secure elements embedded into the host device are the cost of communication and the usability cost induced by lower clock speeds in the separate secure element. As security functionality for consumer devices typically is only a supporting sales argument rather than the primary feature, we see that smart cards and trusted platform modules as a rule are designed to minimize monetary cost rather than performance[80]. The communication channels to these embeddable elements are serial channels, and their clock frequencies and computing speeds are low compared to the main CPUs of the computer and mobile phones. As a result, secure computation using embedded hardware is several orders of magnitude slower than the same logic executed with the full speed of the host processor.

The pricing of secure environments is mainly based on the fixed and variable production costs of the element itself, but also takes into account costs related to business risks, localized certification, regulatory needs, etc. Thus even rough monetary cost figures are hard to state in a fully objective manner. However, to put the work of this thesis into perspective, Figure 3.1 outlines the relative price points of technologies that are discussed and referred to in this work. The PSE, i.e., the hardware security functionality that allows the manufacturer to add software and construct a trusted execution environment, adds up to 10 cents to the processor cost. A programmable smart card chip with a cryptographic engine costs from a few Euros to ten Euros depending on production volume and performance. Programmable hardware security modules that attach to servers as add-on cards are typically worth 1000 Euros or more.

The cost reasons listed above are all significant motivating factors for why separated isolated secure elements have not gained widespread acceptance especially in mobile phones[2]. Instead, as mentioned earlier, trusted execution has been realized as adjunct functionality in the main CPUs and ASICs. In the next chapter we will systematically examine this functionality in more detail,

[2]In contrast to the mobile ecosystems, TPMs, which as a rule are separate discrete components, are widely deployed in business laptops.

and motivate its purpose as a hardware base for achieving isolation and other building blocks needed when implementing trusted software environments.

3.3 Outline of the thesis

This thesis is structured as follows. After this introduction, technological foundations on which trusted execution can be built are presented in Chapter 4. An attacker model relevant for the context is defined in Chapter 5. Related academic work and technologies are overviewed in Chapter 6. With reference to the attacker model, goals and requirements for a TEE environment are presented in Chapter 7.

After providing the necessary background, in Chapter 8 we define the specific goals addressed by this thesis. The chapter first provides the general goal and then in a more fine-grained manner lists research questions solved by the academic publications that support the thesis.

The practical realizations of the thesis concepts are presented in Chapter 9. Three main software architectures have been built around the academic contributions — the On-board Credentials programmable TEE architecture, a TEE instantiation of the Mobile Trusted Module specification and a rollback protection architecture with an external non-volatile memory connected to the TEE. Additionally, a public transport ticketing solution is presented as one application example that leverages the contributions of this thesis. The chapter concludes with a description of the real-world impact of these architectures.

Future work on the general topics addressed by this thesis is considered in Chapter 10. Chapter 11 concludes the extended thesis abstract, and the original research publications follow. They are ordered starting from publications related to hardware and trust roots and ending with architecture- and application related research publications.

Chapter 4

Trusted Execution

This chapter provides an overview of hardware and low-level security features, some of them RoTs, which are design options for constructing *Trusted Execution Environments* (TEE). We follow the Global Platform[40] terminology and define a TEE as the combination of a *hardware platform that provides isolation* and a *software and operating system residing within the security domain defined by that hardware* that is capable of running programs launched into that environment. With this definition also physically isolated smart cards become viable instantiations of a TEE, but we do not explore such implementations further. Instead we concentrate on presenting designs where isolation and other security functions are constructed out of special processing resources and security functions available in the very same processor that also serves normal OS and application execution. The PSE constitutes these hardware features in a mobile processor. When a *Trusted Operating System* or some other scheduling solution is run under the protection of the PSE, the software also supports provisioning and a cryptographic API, and is given access to device secrets and trust roots, then the combination is called a TEE.

Figure 4.1 provides an abstract overview of a TEE implemented with one type of common PSE. The processor core hardware provides extra register banks for a new, secure processor context, a "secure world". Memory isolation is provided by an extension to the memory management unit (MMU) or the memory protection unit (MPU). This extension makes access to some predefined memory pages conditional to the processor being in the secure context. Some non-volatile persistent (ROM) memory is in practice needed for booting the core and possibly for software checks related to entering the secure processing context. If secure boot is implemented as part of the TEE, an internal immutable cryptographic algorithm is also needed.

In addition to the **isolation** property provided by the hardware PSE, the following features will further characterize the software and persistent data that constitutes the TEE:

1. **Secret(s):** Trusted applications that operate in a TEE isolation domain should have exclusive access to derivatives of device secrets, provisioned by the chip manufacturer, device integrator or operator. These provisioned secrets must not be visible to untrusted code, such as the operating system or its applications.

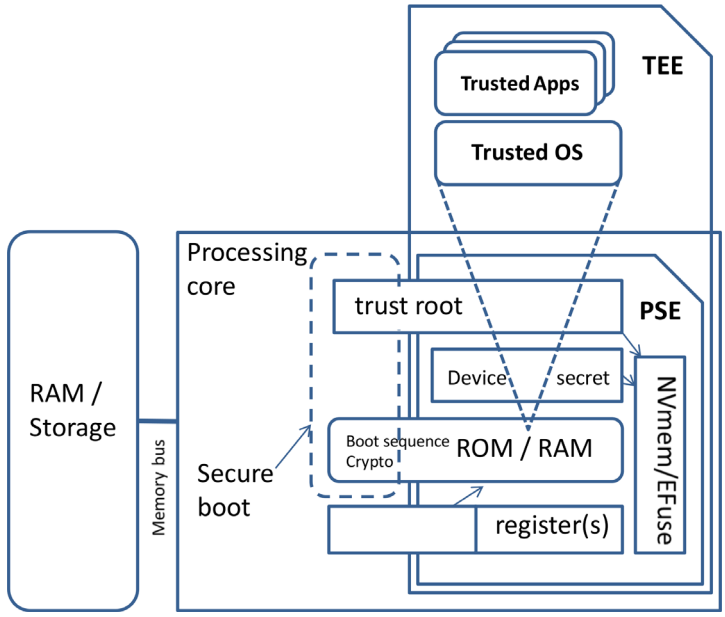

Figure 4.1: Trusted Execution Environments

2. **Cryptographic API:** Basic cryptographic primitives are available to the trusted applications in the form of libraries or other interfaces, and the cryptographic operations are run within the same isolation domain as the calling program.

3. **I/O:** Untrusted code (like applications in the operating system), as well as other trusted applications running in another isolation domain within the TEE, must be able to provide parameters to a trusted application, and to receive computation results from it. There must be a way to select and activate a trusted application.

4. **Provisioning of code:** The TEE environment should be configurable, i.e., trusted applications must be uploadable from the OS or from a device-external entity into the TEE.

5. **Provisioning of secrets:** If secrets are provisioned, they should also be bound to a specific trusted application or a set of such applications.

6. **Code lifecycle management:** Code versioning and updates should be possible, under the constraints of the code issuer policy and issuer control enforced by the owner of the TEE. This management should account for the identity of the trusted application, and secrets already provisioned to it.

7. **Randomness**[1]**:** Due to the nature of most cryptographic primitives, the isolation domain provided by the TEE in practice needs exclusive access to a source of random bits.

[1]Strictly speaking, randomness can often be replaced by counters or authenticated time. Also, the services outlined next can be implemented by a remotely seeded pseudorandom number generator.

The hardware "building blocks" of a PSE, i.e., trust roots, the isolation property and basic security services like secure boot can be implemented in a variety of ways. The rest of this chapter explores these features in greater detail and, where relevant, lists the most widely used architecture options.

4.1 Immutable data

For the purpose of uniquely identifying external trust roots as being relevant for a specific device, and to bind other assigned device identities to the same, the physical silicon chip that also hosts the processing core, needs to include a small amount of immutable, unique data such as a serial number. This data is chip-specific, and thus cannot be gated, i.e., defined as part of the digital design. Instead, e-fuses, a technology that originates from IBM[55], provides a mechanism by which immutable data for a chip can be programmed either at the chip factory, or at the time the integrator assembles a device around the chip. E-fuses are typically embedded into the silicon layers of the chip, and the unauthorized reading of the data written to them requires professional hardware that often destroys the chip in the process.

Whereas the chip identifier is unique to every chip, an external trust root binding, such as the public key hash of an external certification authority, typically is shared across a batch of chips. Still, this binding is typically stored in the same e-fuse bank as the chip identifier. This is because the trust root binding typically is integrator- and not chip manufacturer specific, and therefore it still does not make sense to include this value in the digital design even though large batches of chips do share the same binding value.

In addition, to support secure boot, an ASIC needs to include some amount of immutable program code that bootstraps execution on the main processing core (the CPU) and further steps in the boot chain. In mobile devices, this immutable code typically resides in read-only (ROM) memory on the ASIC itself. In PC technology, a dedicated external memory chip, the Basic Input Output System (BIOS) is traditionally used as the root of the booting process. The immutability of that code was identified as crucial for system security as early as in 1996[23]. Interestingly enough, the first BIOS virus, the Chernobyl/CIH, appeared already in 1998 to underline the urgency of the statement.

4.2 Secure boot

A fundamental building block for providing integrity for a TEE is secure boot. This process implies cascading verification of the code that is executed in the system.

The fundament of cascading measurements is well explained by the principle of *measuring before executing*: Assume that all executing code, before being launched, is measured by some other code launched earlier in the boot process. Also assume that these measurements are sent to and are internally stored by an isolated environment. Then any malware being executed as part of the boot sequence can at most affect the measurements that are collected after it has been launched, since those are the measurements it can modify. I.e., it cannot destroy or modify the audit trail leading up to, and including the

measurement of itself[2].

This measurement aggregation in an integrity-protected environment separate from the security domain where the execution happens is called *authenticated boot*, and this is the primary booting principle considered in TCG specifications[94]. Authenticated boot can a posteriori provide reliable remote attestation of the boot sequence, but typically does not terminate in case unknown code is measured and launched during boot.

Secure boot is in some sense a simpler version of the same principle. In secure boot each piece of executed code measures the next code to be launched and compares it to a trusted reference value, which may be hardcoded or signed. The boot process is terminated if the measurement does not match the reference value. No unmeasured code will ever be launched on the system provided that the first piece of code that the device boots (which performs the first reference comparison and termination decision) is immutable as discussed in Subsection 4.1. Secure boot does not require a separate isolated domain for the boot-up, since theoretically a secure domain is constructed as the device boots up.

Rollback protection for updates of components that are part of the secure boot sequence is needed to maintain the level of device security over the device deployment lifetime. We return to this topic in Sections 5 and 6.2.

4.3 Tamper-resistant device secrets

Immutable device secrets can be maintained alongside identifiers and trust roots in the immutable store inside the ASIC. A device secret in essence is a non-volatile device-specific random number, possibly generated during chip or device manufacturing. The secret itself and any code that uses the secret needs to reside and operate in a tamper-resistant environment to contain the information flow that eventually may reveal the secret to an attacker. A device secret also needs to be immutable, since the device for many use cases needs to use the same secret across boots, for example if the key is used for local secure storage or authentication to remote parties. A single root secret can be diversified into a whole set of mutually independent derived secret keys to cater for different device secret needs.

Clearly, the protection of this tamper-resistant secret becomes crucial for the security of the device - it will be the single point of failure for most of the security architecture built around it [3]. Thus, the protection of the device secret against reverse-engineering needs to be performed on many levels. Possible attacks include deployment of rootkits, side channel monitoring (like timing operations and memory usage), hardware attacks such as probing the memory bus between the processor and the RAM memory and also physical forms of side-channel attacks like power-level monitoring and magnetic or radio emissions. Recent attacks against the TLS protocol[5] [29] show that such

[2]When the audit trail includes only executable code then some attacks can still be applied. Associated boot data like configuration data in combination with an exploitable stack in the boot code may use attack mechanisms like return-oriented programming[18] to inject the system with malware that in principle is not visible in the audit trail.

[3]A related technology is physically unclonable functions (PUF)[38], the use of which may alleviate some issues with hardware attacks against the secret. However, at present PUFs are not available as part of mass-produced chip designs.

attacks are feasible also in real-world deployments.

The attacker model provided in chapter 5 provides one categorization of threats associated with the protection of TEE secrets.

4.4 Isolation for TEEs

The isolation of execution inside the TEE can be arranged in a variety of ways. The solutions will have different cost, security and also performance properties. The most commonly used and researched alternatives are presented in the list below.

1. **Secure boot and virtualization**: This approach builds on the isolation principles of an operating system: By leveraging normal processor memory management and processing contexts, the primary, securely booted component can reserve for itself the privileged computing contexts, and protect its memory pages against other software running at lower context levels. Whereas contemporary operating systems typically are too complex to be considered secure (to constitute the Trusted Computing Base), large operating systems can be replaced by small microkernels or hypervisors, the code of which possibly can be validated or at least examined in terms of platform security. In this architecture, the legacy operating systems are run virtualized or para-virtualized under the control of this security kernel. This setup does not protect against simple hardware attack. Also, as a consequence of the virtualized OS typically running in a user processor context, since the hypervisor occupies the privileged context, the OS may face challenges to protect its own kernel boundary towards its applications. Programs run under this isolation can run at full processing speed, and memory size is not a serious constraint. TCG/DRTM[43] provides in some Intel and AMD processors a hardware-assisted option to setup the TCB for a secure kernel (so called late-launch) also after a possibly insecure OS boot sequence has completed.

2. **New processing contexts with dedicated memory areas**: Security architectures like TI MShield[13] and ARM TrustZone[8][9] add several sets of new processing contexts for the explicit use of constructing security domains in the beginning of the boot sequence. Here, a launched OS can still make use of the standard processing contexts like privileged mode, interrupt context and user space, and remain unaffected by the presence of the trusted OS occupying some higher-security context. This is different from the more stringent, and hard to achieve property where the presence of a trusted OS cannot be detemrined by the launched OS. If the trusted OS reserves for itself (with the assistance of the memory management unit) some ROM and RAM memory that reside on the main ASIC itself, then an isolated domain has been constructed that also is protected against simple hardware attacks. With this architecture, processing still runs at full CPU clock speeds, but volatile memory is limited to the "onboard" RAM, at least for execution that needs to be protected against hardware attacks.

3. For completeness, **full hardware isolation** needs to be reconsidered

in this context. Here the isolation is defined by a discrete chip soldered onto the motherboard. These *embedded secure elements* (eSE) can be highly secure in that they can provide tamper-resistance and even side-channel attack protection as part of their digital design. However, their memory is constrained and no direct memory access to system memory is typically available. An eSE is by itself not enough to construct device secure boot, it requires additional architectural assistance from the platform, like the ability to control the first instructions that execute on the main processor. As a rule eSEs also compute at slower clock speeds than the main processor.

4. **Processor mode with temporary isolated memory**: The main example in this category is the Flicker architecture[63] that leverages the late-launch property of Intel and AMD chipsets. The setup is interesting: Although the isolated environment does not have run-time, dedicated memory and therefore cannot persistently hold any device secrets, the late-launched environment leverages a TPM (a separate discrete chip) for secure storage in a manner where the code image to be executed in the temporary isolated environment will constitute the access control attribute for the storage. The launched isolated code gaining access to the storage can for a limited time use on-chip MMU caches as its main memory, therefore remaining resilient to memory bus probing [4]. In short, this hybrid environment shares many of the properties of the environments listed above under "New processing contexts and dedicated memories".

Some of the architectures above do not protect against simple hardware attacks, like eavesdropping the memory bus interface. However, one layout option used in mobile devices is to stack the processor and the memory to save real estate. Here the processing unit and its main memory are packaged together within the same plastic package, and the memory bus connectors can be placed in a way that accessing them by only peeling off the covering plastic is not feasible. In this case the simple memory bus access with such hardware becomes significantly more difficult, and the relative difficulty between accessing RAM/ROM memories inside the ASIC and accessing the main memory dilutes.

E.g., as described in the context of Aegis[88], another approach for improving memory protection is to add memory-page encryption on the hardware level, i.e., to the memory management unit. With this design we can use a key randomly generated at each boot to encrypt or decrypt all data written to and read from the volatile memory[89]. Eventually this approach may become practically feasible, like disk encryption has become today, but for now the challenges of arranging fast enough operation for encrypting the memory pages with necessary integrity and rollback protection[37][22] has not yet brought designs into widespread commercial use.

In this thesis, we base our TEE contribution on the isolation model 2 described above, except for publication **P5** that uses model 4.

[4] This architecture also exposes a new hardware attack surface — the connection between the main CPU and the TPM opens up to eavesdropping or modification

Chapter 5

Attacker model

For TEEs, the details of the attacker model will depend on how trusted execution is set up in terms of hardware and software protection. Inside the device, the well established Dolev-Yao attacker model[28] used in the protocol and networking community can be applied between interacting hardware components rather than between interacting devices. When examining TEEs we can approach the attack surface in terms of what security property may be breached, but also of significant relevance is how the attack is achieved. This section reviews both aspects.

At least the following attack types are relevant for TEEs implemented with a PSE in the main processor:

1. **Loss of secrets:** As a TEE ultimately is used to guard and process secret information the confidentiality protection is one of its paramount goals. In TEEs, eavesdropping applies not only to pure data, like an attacker getting hold of cryptographic key material, but may also target algorithm flow information leakage. TEE computation may happen in an environment where extensive scheduling and virtual memory caching to insecure memory occurs. Even if the caching is encrypted, the monitoring of insecure memory contents can to an adversary provide significant information about algorithm flow, and even indirect information usable to reveal secret key material.

2. **Modification:** Integrity is the other fundamental property that many of the security services in a TEE are built to maintain. These modification threats do not only apply to data but also to the boot sequence (secure boot) and the integrity of results returned from a TEE to trusted UIs and external (trusted) communication channels. Replay and splicing are important subtypes of modification attacks.

3. **Replay and rollback protection:** Wherever caching of information takes place, whether stored in a database or as an encrypted virtual memory page, state protection is needed. That is, when taking in previously cached information, a TEE should have assurance of the fact that the retrieved information is current and not an old version of the same data.

4. **Data context consistency:** When considering virtual memory, the

ordering or mapping of information across memory pages can be relevant. A *splicing attack* returns a data element in the wrong context or in the wrong order, and any caching system should include protection against such attacks.

5. **Denial-of-service:** Denial of service is the activity of blocking access to or causing the target TEE to stop servicing legitimate requests. By designing software TEE architectures to be amenable to backup and restore mechanisms, the impact of DoS attacks can be minimized.

Many forms of attacks can be used against a TEE, operating inside a device, to breach one of the security properties mentioned above. The following list provides one categorization of attack mechanisms.

1. **Brute-force and injection attacks:** These attacks are typical forms of leveraging software weaknesses to mount a software attack. A program or virus can look for implementation weaknesses by bombarding the interface of the TEE or trusted applications with fuzzed input and examine return parameters for information leakage. Randomized fuzzing may also trigger a software flaw inside the TEE that lowers the protection boundaries maintained by it. More advanced versions these attacks may inject data causing errors, as error-state security is often less thoroughly tested and analyzed than the default software path. Stack-smashing and buffer overflow attacks are well known in the software community. Such concrete attacks can be deployed e.g. based on information gathered by brute-force fuzzing.

2. **Side-channel attacks:** These are indirect attacks whereby a measurement of an external property reveals information of the internal state of TEE execution. Such properties include hardware attributes like consumed time and energy, or software ones like the size and value of encrypted data produced by and later retrieved from the TEE. Multithreaded processors with parallel execution pipelines and different levels of memory cache table lookup buffers (TLBs) further add a multitude of properties that potentially may cause information leakage[95]. Side-channel attacks can be divided into hardware- and software-originated ones. Countermeasures to these attacks are a research topic by itself, see e.g., [99].

3. **Covert channels:** In case TEE functionality intentionally leaks or distributes secrets or other sensitive data to an external party, this is a covert channel. Third-party compliance and certification procedures mitigate this risk for the consumer. The same concerns do apply to trusted applications as well.

4. **Offline attacks:** This class of attacks refers to the reading or modification of the persistent memory of a device when the device itself is in a powered-down state. An attacker may modify system code that is part of the boot sequence of the device or change configuration parameters read by launched programs. Sensitive user data can be attacked. Especially relevant for the TEE discussion are state rollback attacks, which can be considered a special case in this category.

5. **Supply chain attacks:** The manufacturing and secure imprinting of the device is critical for the security level that can be reached in terms of platform security over the whole device lifetime. During manufacturing, security contexts between the manufacturer and the device are set up to later protect functions like firmware updates, device service, operator binding and TEE code provisioning. An attacker resident in a manufacturing facility might modify data provisioned to a device or record information that enables him to replace or circumvent device trust roots used after device deployment.

6. **Software attacks:** From the perspective of the TEE, the main operating system and its applications are vulnerable to attacks. This means that a TEE can only to a very limited degree trust operation invocations or data input provided to it by this environment. Also, in the absence of non-volatile read-write storage or large volatile memory resources, a TEE must often cache both persistent and temporary data outside its own memory domain into memory controlled by this insecure environment. So both TEE input, output and context is potentially subject to software attackers in the form of root kits, viruses or even the users' own attempts to circumvent TEE protections. Also the executable code for the trusted applications is often available to code running in the privileged domain.

7. **Simple hardware attacks:** In typical computing architectures there are a number of hardware attacks that are straight-forward to execute and often leave little or no traces after the attack has been executed. These are attacks that can be executed by so called "clever outsiders"[7] without expensive equipment. External devices can often modify memory or trigger the processor core through interrupts (IRQs), direct memory access (DMA) and debugging interfaces (JTAG). In case the random access memory (RAM) is physically separate from the core processor, memory bus eavesdropping and content modification on the memory interface is a relatively easy and non-intrusive undertaking. By personal communication we learned as early as 2006 that telephone SIM locks were broken by using temporary memory bus probing. Attacks against the TPM, where grounding a (reset) pin[50] causes loss of run-time state, and attacking the LPC bus to modify TPM traffic like DRTM measurements[48] can also be considered a simple hardware attack.

8. **Complex hardware attacks:** Since main ASIC rarely are designed to be tamper-proof, they can be attacked by probing, i.e., drilling holes in the ASIC to access internal communication buses or processor debug connectors. If an ASIC can be destroyed, then its contents can be revealed by peeling the silicon and examining it layer by layer[92]. These attacks are expensive to mount but cost-effective if many devices share a common secret that is written into the E-fuses on the ASIC. Architectures considered in this thesis do follow the guideline to never share master keys between large sets of devices.

Many of the attack types listed above leverage weaknesses in the implementation of TEE code. It is hard to overstate the importance of code correctness

when implementing any critical resource. Section 9.1.2 and publication **P4** contribute to this aspect of TEE security.

Chapter 6

Related work and technologies

This section outlines technologies and research work that relates to the publications attached to this thesis. Academic work is referenced where available and appropriate, but some fundamentals are architectures brought to use by companies without publications in the traditional open innovation model — in such cases other references are cited.

6.1 Secure virtual memory

Temporarily caching information is vital for environments with constrained memory resources. Already in the 90s Blum & al.[17] set the foundation by providing proof that integrity of various data structures can be maintained in an environment with a small amount of trusted memory and a larger untrusted storage. In this model, only the active part of the data structure, i.e., the piece that is currently being operated on, is kept in the secure environment. The rest of the data structure resides in untrusted memory where it will have to be appropriately protected against eavesdropping, rollback and re-placement within the data structure. Blum & al. abstractly prove that such a construction can be made secure at least for data structures such as stacks, FIFOs and memory pages. The main mechanism for providing the integrity guarantee for variable sized data structures is a distributed tree structure, typically some form of Merkle tree[64]. To turn the proofs and mechanisms into practice, much more research has been published targeting both hardware and software implementations, especially for protecting virtual memory. Gassend & al.[37] argue that by implementing the prover as part of a processor cache efficiency can be achieved. E.g., Suh & al.[89] optimize the integrity check calculation with multi-set hashes, targeting the special case where the integrity of a set of memory reads can be validated after all reads have been completed. Elbaz & al.[33] provide a recent survey of many academic approaches for implementing secure virtual memory. However, none of these mechanisms have yet been applied to commercial processors, thus they are not available with TEE hardware architectures on the market today. One likely reason is that on-line checking in this context (limited secure memory)

cannot be done in linear time (Dwork & al.[30]), which makes its real-world implementation challenging. There is also some published research for protecting virtual memory management in software[73], but these are OS-specific solutions poorly adaptable to TEE environments.

6.2 Non-volatile state

Securing the non-volatile state of a device TEE over boot cycles is a cost consideration — adding flash memory into the security domain of a TEE on an SoC is expensive. This issue was identified by Suh[88], who does not have a solution for the problem, except for using an external trusted third party to maintain the device's non-volatile state. This approach is acceptable for devices with permanent network access and for all use cases that only need security after boot-up. However, for operations related to secure boot, for rollback protection of device bootloader updates, relying on a device-external resource is not a satisfactory solution. Schellekens & al.[79] identify this issue, and propose to include a memory with authenticated communication in the device. They however omit to consider how the security context between the TEE and the authentication is set up and how the life-cycle management like testing, repairs, and replacement of the memory can be arranged.

Research that proposes to include non-volatile memory on the SoC chip itself also exists. In fact, already in 2004, Raszka & al.[76] propose a hardware design for embedding flash memory onto the ASIC for the explicit use of security applications. Zhao & al.[98] have presented a non-volatile magnetic flip-flop that holds state across boot cycles, and thereby can serve as non-volatile secure storage. One-time programmable E-fuse technology[55] is the primary non-volatile storage mechanism on SoC processors today. Like physically unclonable functions[38][87], this technology is ill-suited for changing data, since it is one-time programmable. Still, it can be used to implement counters by burning one fuse at a time provided that a SoC chip holds a large enough set of fuses. This use of fuses is foreseen in the MTMv1 specification[67], see section9.2.

6.3 Cryptographic co-processors and smart cards

The need for isolation of computing in servers and client devices coincides with the emergence of internet as a global communication network and with the increased digitalization and computerization of financial sector. This took place, especially in Europe, in the 1980s and onward. The IBM 4758 crypto co-processor[82] card, and its successors have been one of the most used isolation environments for certification authorities, banking servers and factory networks around the world, wherever so-called hardware security modules (HSMs) have been needed.

6.4 Smart cards

A smart card is a tamper-resistant computer in the housing of a credit card. The first smart cards with an embedded processor emerged in the late 1970s[74].

By the late 80s the standardization of the form factor and its wired communications interface were well underway, and the first cards with RSA capability emerged. The dominant standard for smart cards is the ISO-7816 suite of standards, which covers everything from the physical characteristics of the interface to file organization on the card and protocols for host communication with the card.

The tamper resistance of smart cards has been evolving since the 90s. The first "cautionary note" by Anderson[7] outlined attack opportunities against smart cards as early as 1996, and over the years much work has been published on executing and preventing side-channel (power analysis) attacks on smart cards (see e.g., [65]).

In the 1990s the mass deployment of smart card technology started, driven by GSM mobile networks where smart cards are used as subscriber identity modules (SIMs). With the advent of programmable, multi-application smart card technology in the late 90s[25] with environments like MultOS[20] and JavaCard[19], the high end smart cards essentially turned into secure environments, with threat models very similar to those considered with crypto coprocessors[75]. Today, smart cards are also extensively used as identification modules for the financial sector, and the security ecosystem for smart cards is largely specified and standardized in the Global Platform Consortium[40], which also has task forces for mobile device technology and on-chip hardware architectures.

Today, most deployed smart cards are removable, i.e., they communicate with a host device through an external or internal smart card reader over wired or wireless protocols. These cards are not in scope for this thesis. However, especially for payment and ticketing applications in the wireless context, there is a small population of mobile phones that include embedded smartcards in their hardware, turning the mobile phone into a "smart card" with user interface and communication capabilities. From a hardware perspective, also TPMs[94] can be considered to be smart cards, albeit adhering to different interface specifications and protocols.

Several application standards for smart cards are also important for understanding the state of the art. The EMV payment card standards by Visa, MasterCard and Europay[36] for credit card payments emerged already in 1996, and credit cards with EMV are becoming commonplace today all over the world. Also the contactless variant of EMV[34] is widely used today, especially for small-value payments. For smart card application development and especially their remote provisioning, the Global Platform[40] specifications are the de-facto reference. Other significant smart card application standards are the network identification modules (SIM, UICC) by ETSI/3GPP. For TEE-like operation, the smart card industry in many cases still does not provide unconditional isolation between programs provisioned onto the card. The problem to overcome is that the original JavaCard language specification leaves the bytecode verification to an external trusted party rather than executing that process on the card itself. This is a recognized problem in the academic community, and several solutions for making on-card bytecode verification possible (e.g., [57]) have been proposed.

6.5 TPM

The TPM chip[94], specified by the Trusted Computing Group (TCG)[93], has been successfully deployed to hundreds of millions of laptops and PCs to date. The technology was in the early days around 2003 known by the name TCPA (Trusted Computing Platform Alliance), or by the project (and team) names in Microsoft, "Palladium", or "Next Generation Secure Computing Base". The effort that originally was initiated by the big players in the PC ecosystem: Intel, IBM, HP, AMD and Microsoft, can today be considered a global effort to bring trusted computing, especially a hardware root of trust, into the personal computing ecosystem. The original target scope, PCs and laptops, has been extended to include servers, network equipment, tablets as well as mobile phones. To date, the TPM1.2 chip can be used as a hardware trust root for Microsoft Bitlocker. Google Chrome OS for netbooks requires the presence of TPM1.2 for its platform security. A similar requirement is present for the forthcoming Microsoft Windows 8[83].

TPMs are typically stand-alone discrete chips that provide a set of security functions for the operating system and its applications. The TPM interface is a set of functional APIs for security services carried out in isolation, i.e., inside a TCB. These security services include functions for RSA key generation and subsequent key use for signing and decryption, as well as for secure storage — functions that one could expect e.g., to be available in a smart card. What sets the TPM interface apart from the one of an embedded smart card, is that TPM in its interface leverages the fact that the device TCB is bound to a specific device, and is not, by specification, removable. For this, the TPM defines a volatile set of cryptographic aggregators, the so called platform configuration registers (PCRs). With only a minimal securely booting component in the host computer, the so called Root of Trust for Measurement (RTM), the envisioned boot sequence of a TPM-enabled device is that the minimal secure boot (which in a PC may be some memory blocks in the BIOS), will measure (cryptographically hash) the next platform code to be booted, and send that measurement to the PCRs. By following this "measure before executing" principle, it is straight-forward to determine what trusted code has been booted on the device, or put another way, as long as the measurements sent to the PCRs are known to represent a well-known secure state, trustworthiness of the end configuration is maintained. This **authenticated boot** is straight-forward, but does not well accommodate mixed provenance software stacks as part of the boot sequence - a problem addressed by the Mobile Trusted Module (Section 9.2) and publication **P1**.

Based on the PCR values collected at each boot, **platform binding** can be achieved: The TPM interface allows for key usage or retrieval from storage to be *bound* to the PCR values, i.e., the key operations will only be allowed if the PCR values are the ones listed in the binding associated with the key. Ergo, the keys are only usable if the device was booted into a secure state. Since the boot sequence was never aborted if a "wrong" measurement was encountered, this way of booting the platform is called authenticated boot as was described in Section 4.2.The PCR values are also leveraged in the service of **remote attestation**, i.e., where TPM signs a challenge and the current values of the set of PCRs with a certified key. The attestation can be parsed and validated by a relying party, which now can determine something about

the current state of the booted client platform. Finally, also data can be bound to a given PCR state, to be released (decrypted) again only when the very same TPM's PCRs reach that state. This process is called data **sealing** and **unsealing**

The TPM technology has induced a large body of research publications in the field of trusted computing. Directly relevant to this thesis are publications that relate to MTM and the adaption of TPM to PSE environments.

In 2007 Constan & al.[21] defined the notion of a programmable environment — the "trusted execution module" (TEM), which could be used to implement TPM features. The same year publication **P3** was published, Kursawe & al. looked at the implementation of "microTPMs" using the TEM paradigm, and introduced an implementation based on *disembedding*, i.e., one that executed the TPM logic in parts due to trusted environment size constraints. The first published work on MTM implementation was by Zhang & al.[97], who implemented MTMs within an OS with SELinux[61]. In addition to **P3**, Dietrich and Winter look at MTM disembedding[26] with a Java interface and a mobile phone secure element — very much in the TEM model. England and Thariq[35] consider "programmable TPMs" in another context — where a physical TPM is "coupled" with a smart card. In this setting the TPM especially can provide smart card applications with platform state, a feature not otherwise available in the smart card ecosystem.

6.6 NFC

The Near-Field Communication (NFC) industry consortium was formed to standardize technology for tags supporting Wireless Radio Frequency Identification (RFID). In practice, the standard today covers several short-range radio technologies as well as many use cases, ranging from the aforementioned identity tags all the way to device-to-device (i.e., phone-to-phone) communication. The communication distance for NFC devices falls into the range of centimeters and the communication paradigm is often compared to a "touch". Communication speeds start from below 100kbps and are at best approaching 1Mbps, i.e., NFC is by modern communication standards quite slow.

The communication specifications are found in ISO / IEC 18092 [45] and ISO / IEC 21481 [46]. The NFC forum[1], provides compliance testing and additional standards for NFC use. NFC devices support one or several of so called Type A, Type B and Felica encoding, modulation and transmission schemes. Felica is used mostly in Japan and has a nominal transmission speed of 424 kbps whereas types A and B are the norm in Europe and north America (106/212 kbps). In practice, compatibility between tag readers, tags and phones is unfortunately often only found at the lowest common denominator at 106kbps. Additionally, the throughput on higher protocol layers remains significantly lower due to protocol overhead — again a typical situation with wireless communication standards.

An NFC device can be passive, i.e., wirelessly taking energy for its computation from an electromagnetic field produced by its peer. Passive devices are either memory tags or contact-less smart cards specified in ISO / IEC 14443. Active NFC devices (with their own power) can operate as readers in

[1] `www.nfc-forum.org`

so called card reader / writer mode when they communicate with a passive device, or in peer-to-peer mode when they target another active device. Some active devices also operate in so called in card-emulation mode, where the active device "emulates" a passive device. Contactless smart cards use the same ISO / IEC 7816-x base command set as their wired counterparts.

The lower layers of NFC include no communication security primitives. It is also well known that NFC technology is susceptible to both eavesdropping and relaying attacks[24], despite the fact that NFC is a very short-range radio technology. An review and taxonomy of the security threats for the baseline NFC technology can be found in[60].

NFC technology is today becoming commonly available in mobile phones. Although not yet deployed to low-end models, a list maintained by NFC forum[2] is quickly approaching 100 device models from all major mobile phone manufacturers. This evolution is significant for platform security, since many of the major use cases envisioned for NFC are services with high security requirements. Examples include payment, ticketing and physical access control.

6.7 Secure hardware

The evolution of hardware PSEs that forms the basis of a TEE has been driven by commercial companies. In this subsection, brief introductions to the most commonly deployed PSEs are given in addition to presenting Aegis — one prominent research contribution that has been published in this area.

6.7.1 Texas Instruments M-Shield

One of the first commercial on-chip secure environments is the result of a collaboration between Nokia and Texas Instruments[90][53] in the early years of the 2000s, resulting in a hardware solution minted M-Shield[13]. In early versions of this architecture, the security monitor — a hardware logic external to the ARM processing core but internal to the chip was triggered by executing an secure mode entry ROM program, and the isolation was managed by the security monitor directly controlling some of the memory address bus lines in the ASIC. When the secure mode was entered, addressing on-chip memory and e-fuses was possible, otherwise not. The HW security system also included a security watchdog timer, a hardware random number generator (RNG) and a few other hardware primitives. The software part of the solution consisted of the monitoring entry code. Its purpose was to ascertain that on secure mode entry the caches were flushed and virtual memory and interrupts disengaged, to protect against unintentional information leakage.

The M-Shield trademark still lives on today as a Texas Instruments systems offering, but the hardware fundament is now based on ARM TrustZone, described next.

6.7.2 ARM TrustZone

The main PSE architecture in use today, especially in mobile device processors, is ARM TrustZone (ARM-TZ), which has been available as a processing

[2] A list of NFC-enabled phones can be found at http://www.nfcworld.com/nfc-phones-list/

core option in ARM-enabled processors since the ARM-1176J-S[8] was released. In terms of functionality ARM-TZ[8][9] is very similar to M-Shield, but as the system is integrated within the processor core itself, it is significantly more powerful. In addition to the regular register and processing contexts (user-space, kernel mode, interrupts), the TZ adds a second set of contexts, that ARM documentation calls "Secure World". As a bridge between the baseline register set (the "Insecure World") and the secure world, one more processing context, the monitor mode, is introduced. The role of the monitor mode code, entered by the caller triggering a software interrupt, is to determine whether secure world entry is acceptable in terms of processor and device settings, akin to the ROM entry sequence in M-shield. If these checks are successful, the monitor mode has the exclusive right to clear a control register bit, the so called "non-secure" (NS) bit which defines the entry into the secure world, but also serves as the defining signal to all other security aware hardware support built around the TZ architecture. For example, the interrupt controller can be configured with a completely separate interrupt vector when in secure world and the MMU can for each mapped page also adhere to the current state of the NS bit, as can the settings of the DMA controller. The same bit is also mirrored on processor communication buses, enabling core-external logic to alter their operations based on whether the processing core is in secure- or non-secure world — this may apply to timers, clocks, watchdogs and the like, but also to external memory controllers or communication hardware. All this processor support provides a versatile base on which to build a TEE, especially if the secure memory mapping is backed up by the existence of ASIC-internal ROM and RAM memory, so that secure memory access is not visible on external interfaces.

In ARM TZ the processor core always boots "in" the secure world. This is logical, since the secure world is the trust root of the system, but this also allows for the boot sequence to configure secure world features — to read secrets and initiate isolation boundaries for the secure world — before engaging in non-secure operating system startup.

6.7.3 Intel VT-x / AMD SVM

The main companies supplying the PC industry with processors, Intel and AMD, have designed an interesting PSE-like facility that as a construct is quite different from the ARM TZ.

As discussed in Section 6.5, the TCG technologies provide authenticated boot, where the initial part of the boot sequence is measured into the TPM. There are several drawbacks of this approach. One is that the measurement cannot be repeated after the device has booted, thereby it does not adequately protect against run-time attacks — computers like PCs can have boot cycles that last for months and even years. Another issue with boot-time measurements is that in open hardware platforms like the PC, auxiliary hardware devices with memory access at hardware level can be inserted by the device owner, and the firmware of these devices and their DMA use is not in all cases adequately reflected in the boot-up measurements.

As an answer to the abovementioned concerns, in 2005 Intel and AMD for TCG designed a concept often denoted "Late launch". The feature is part of the *Secure Virtual Machine* technology in AMD processors[6][85], and the

Trusted Execution Technology by Intel[43]. In the core of this hardware support is a feature whereby a "fresh" TCG-style software measurement of a TCB can be initiated at any time when the device is running. This measurement will be performed in a processor state where possibly harmful hardware concepts like interrupts, DMA and in the PC case even multi-core operation are turned off. This processor state is similar to the state in which the ARM TZ monitor mode is launched, as described in Section 6.7.2, i.e., the measurement (computation) to be conducted cannot be affected by plugin hardware devices or malware residing in the system. The TCB that is measured is a small piece of code that will be launched after the measurement completes. The processor sends the code of the TCB to the TPM, which computes the code hash, resets a PCR dedicated for this purpose, and extends that PCR with the code hash. Then it launches the TCB.

In this state, the binding properties of the TPM chip for keys, encrypted data or non-volatile storage is a straight-forward way of achieving secure storage for this TCB. Only the code whose measurement corresponds to the binding is given access to the store, or to the use of secret keys.

The main communicated intent of including late launch in the processor architecture is to set up a hypervisor to run underneath a virtualized operating system. Other features of the VT-x and SVM architectures do support exactly such concepts — the hardware means to virtualize or compartmentalize the operating system without need for modification in the OS code. I.e., the hypervisor is measured, and can be remotely attested by TPM functions by leveraging secrets in the secure storage provided to it. The late-launched code becomes the TCB, and if that TCB provides facilities for executing secure service logic, it effectively can become a TEE.

6.7.4 Aegis

The discussion on PSE hardware would not be complete without presenting Aegis[86][88], a secure processor hardware design originating in the research community in 2003-2005, thus it predates many of the architectures presented above. Aegis introduces a processor with several relevant PSE concepts. One is the notion of attested execution of a secure program, i.e., the processor microcode combines the secure program results with a signed attestation of the device, its security kernel (the TEE), the executed program, and its inputs. With this information a remote party can validate the system (and its state) in which the returned results were computed. Aegis also includes integrity-protected, and potentially encrypted caching of memory pages using the L2 cache, making memory footprint considerations unnecessary as all memory can be considered secure. The Aegis system also includes software extensions to interrupt and later resume trusted application execution measurement when network connectivity or other system operations that by nature include too much code or state to be included in the attested algorithm execution trace. The contributions of Aegis hardware research has not yet reached contemporary processor technology.

6.8 TEE software architectures

The software architectures related to a TEE are often engineered to the specific hardware environment they operate in. However, as these software modules typically strive to overcome inherent shortcomings of the related hardware architectures, they are often instructive in their own right.

6.8.1 Early Nokia software

The early MShield hardware was in Nokia devices leveraged by a software layer that implemented a simple TEE. The basic structure of this TEE can be found in [53], where Kokkonen presents a few fundamental software concepts needed in a processor TEE. One is the presence of an external trust root, that constitutes or binds the public-key certificate hierarchies that in his model authorizes securely booting images or the loading and execution of code inside the secure environment. Another is the presence of an immutable hardware identity, to which external trust roots like secure boot certificates unequivocally can be bound.

His model also includes the use of device secrets shielded from ordinary system software, in MShield[13] implemented using device e-fuses, to provide secure storage for trusted applications.

Additionally, the architecture included an authenticating program loader, a cryptographic library, secure storage and a ROM patching facility. All in all the properties of this early architecture did provide an isolated environment that was used mostly for device manufacturer services — subsidy locks, IMEI (device identity) integrity, phone variant locking etc.

The On-board credentials architecture (ObC) was originally built as a trusted application on this TEE in [53], and first presented as a poster[11] in 2008. ObC adds additional isolation for third-party trusted applications, scheduling and open provisioning for third parties. ObC is presented in Section 9.1.

6.8.2 OSLO

OSLO[50] is not by itself a TEE, but the first publicly reported software project that made use of TCG DRTM for late-launching code. It is constructed as a solution to issues found with TCG authenticated boot and contemporary TPM chips and BIOSes. The primary objective of OSLO is to re-measure, and re-activate the OS after the initial device boot-up.

6.8.3 Flicker

The Flicker[63] architecture builds further on the OSLO concept and is the first architecture where the late-launched VT-x / SVT environment is used to execute a single secure credential under the isolation properties of the late-launched measured environment. It consists of a Linux driver that sets up the call to the late-launched environment, and a boot-strap that takes care of the essential secure credential entry and exit procedures, like passing input parameters and returning results. On return, the Flicker architecture releases control back to operating system, but the TCG/TPM measurements of inputs and outputs of the last Flicker execution are available and can be remotely

attested, much like with the Aegis security processor. The Flicker architecture has been used to demonstrate several security applications like LaLa — a late launched application[39] as well as publication **P5**.

In the x86 environment, some research projects have achieved isolation with properties that are architecture-specific. SICE[12] achieves isolation by leveraging the System Management Mode (SMM) on AMD processors. In SICE, needed trust roots are perceived to be provided by a TPM. Since SMM code needs to be authorized by the chip vendor, in a real deployment SICE needs to be released by such a company.

TrustVisor[62] is a small hypervisor that constructs a TCB - a memory environment with isolation and attestation for selected portions of applications without relying on the OS for isolation. The main protection is set-up using MMU memory protection primitives, making the the scheduling of such application critical code efficient.

6.8.4 The path to trusted operating systems

Building on the isolation property and trust roots, one path, and in many cases a likely end goal, is to either run a scheduling operation in the hardware-isolated domain to improve the trustworthiness and firewalling between several OSs in the same device or between applications in the untrusted environment. These approaches are quite different from those brought forward in this thesis, but nevertheless form an important research direction in the field of trusted computing.

Hypervisors are small microkernels that schedule de-privileged operating systems within a computing device. Among others, the Trustvisor project mentioned above is one project that uses hardware isolation as a fundament for improving hypervisor trustworthiness and to leverage existing hardware trust roots. The NOVA project by Steinberg and Kauer[84] has similar end goals.

Microkernels and hardened, minimized OS kernels designed with security and isolation in mind is a research direction initiated in the 90s, e.g., by Liedtke & al. [58][59]. The L4 microkernel has over the years been used extensively in mobile environments, and one variant, the seL4 kernel has recently been formally verified for security[52][51]. The smart card embedded OSs like MultOS[20], share many of their security requirements with microkernels. Future Trusted Operating Systems, i.e., operating systems that run and isolate applications within the PSE isolation domain will likely leverage these or similar technologies. This is discussed further in Chapter 10.

Chapter 7

Security Goals

In Section 4.4, the basic hardware properties of PSEs were outlined. As an implementation of these we describe one real-world PSE hardware architecture, available on a vast majority of mobile phones deployed today, and explore the software requirements needed to complete the building of a fully functional TEE on this hardware base. Some of the requirements are by default fulfilled by existing software and earlier work[53] — these properties are acknowledged in this section.

In the context of this thesis, the main implementations of the software architectures satisfying these security goals will be described in Chapter 9.

The hardware architecture is fundamentally ARM TZ, with three on-ASIC memories — a bank of SDRAM, a ROM and a bank of e-fuses for non-volatile storage. The e-fuses hold at least a device-specific secret seed, an external trust root (a public key hash), and a public chip identifier — the latter can also be a derivation of the secret. The secure booting of such a TEE is straightforward, and described in Figure 7.1. The chip boot vector (0) points to the on-chip ROM, which securely boots a bootloader that sets up the ARM TZ. The MMU is configured to assign the whole or parts of the on-chip memories to the secure world. Additionally, a monitor code and entry scheduler is installed in the secure RAM (1+2), to provide some general services like a) validation of the public keys of the external trust root used for further secure boot and other authorization, b) signature verification primitives for validating further code launched as part of secure boot, and c) interfaces and authorization support for subsequent installation of secure-world code to the secure RAM. At this point the TEE basic properties are set up and the isolation of its domain is configured by means of ARM-TZ. The entry scheduler is henceforth the guardian of all TEE entries, both for entries where the functions of the scheduler itself is called, and for entries where uploading and executing trusted applications takes place. Such code needs to be digitally signed by the device manufacturer to be allowed for execution inside the TEE, an approach that is no different from what was deployed already in the IBM cryptographic co-processor[31].

As the TEE has been set up at the initial stage of the bootloader, it can continue booting up the rest of the OS and platform by using the interfaces provided by the TEE (3). The general setup of this baseline TEE architecture is well understood and dates back at least to 2003 as was described in section

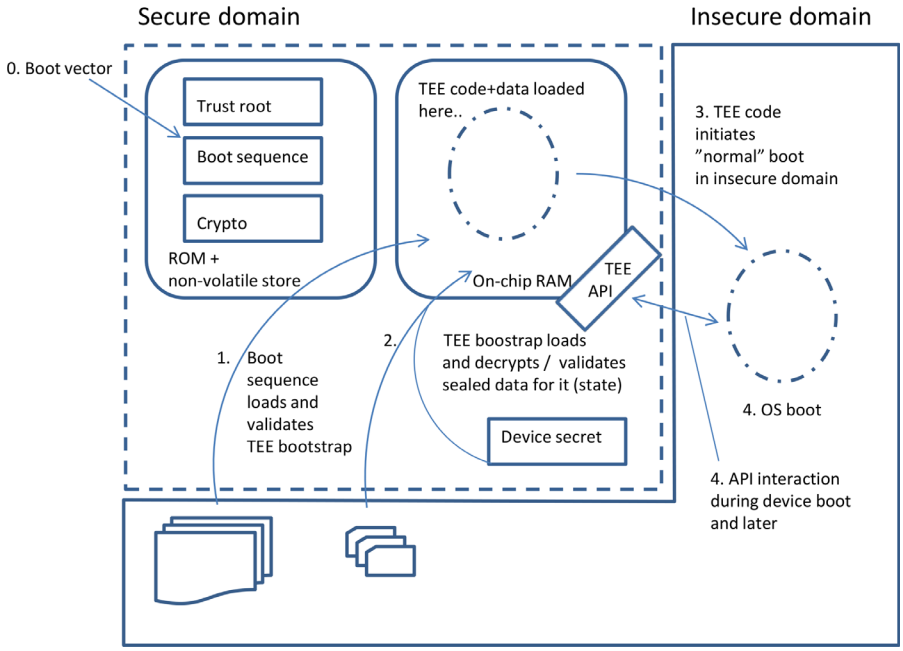

Figure 7.1: Securely booting a TEE

6.8.1.

To program for the TEE, the uploaded trusted applications can leverage a TEE-internal API that typically includes cryptographic primitives like RSA, AES, DES and several hash functions. These operate on their inputs within the secure-world domain, sometimes making use of on-chip hardware accelerators. The API also gives the uploaded trusted applications access to the device secret or a derivation of it, a hardware RNG, and possible some limited storage inside the secure RAM that is available for the TEE, and persistent for the duration of one boot cycle.

The secure memory available for the secure-world programs varies depending on whether the TEE Internal API and secure-world installation code resides in RAM or ROM. It is not uncommon in contemporary smart phones that only 10-15kB of secure memory remains for secure-world program execution, i.e., its code, heap and stack. Additional memory is accessible from secure-world code, e.g., memory used for parameter I/O to the secure world — this is typically arranged by the calling driver as physically mapped I/O buffers from insecure memory. All operations involving these I/O buffers must however be considered insecure in the sense that such memory access will happen over a memory bus outside the chip, open to simple hardware attacks.

Our extended TEE architecture does not make use of virtual memory (MMU), interrupts and DMA when operating in the secure world. Thus, in the following discussion, we assume that these hardware features are turned off on secure world entry and remain so until the secure-world program exits. Instead, scheduling is done purely in software. For us, the scheduling in software was an imposed system requirement. However, the results of this

thesis regarding scheduling are applicable also to hardware-triggered operation, since the security issues arising from the scheduling of execution between secure and insecure memory do not depend on the source of the trigger.

The baseline TEE described above is by itself usable for a variety of device manufacturer security services. However, if we consider any trusted application, implementing a security service or an algorithm on behalf of an external service provider or even an OS component, the following system properties are required:

1. **Rollback protection:**

   The statefulness of operations and non-volatile data in a TEE is crucial for most services, especially if the services are executed in an interleaved fashion. Several types of rollback protection is needed: For provisioned code and data, which may be upgraded over time, the TEE should have mechanisms to ascertain that only the latest versions will be usable — A trusted application version upgrade may be motivated by a flaw in earlier application versions. Second, many trusted applications need the means to store information locally in a secure, non-volatile storage. In this type of TEE such storage is by necessity arranged outside the TEE. The TA, when reading in information from the store, must however be able to rely on that the store information is current and that an attacker has not replaced the store with an older version. Third, if a TEE architecture will execute the TA code in a piecewise manner, scheduled from an OS driver, replay protection must be applied within the TA execution session to guarantee that data flowing between the partial executions are not replayed out of context or that the execution steps are not executed out of order.

2. **Isolation:**

   The TEE provides hardware isolation from the rest of the system, by virtue of the memory banks that reside on the same chip as the processor core with the PSE features for context separation.

   Additionally, any trusted application that is uploaded to the TEE with different authorization than the one used for uploading TEE system code must remain logically isolated from the TEE system level as well as from other trusted applications, unless authorization for interaction between trusted applications is provided.

3. **Secure scheduling:**

   By consequence of the small footprint of the TEE, uploaded trusted applications need to be split up and executed in pieces. For the same reason, the scheduler itself needs to reside in a driver in unprotected memory, and only a minimal security context shall be maintained in secure RAM to ensure the integrity of the expected trusted application program flow.

4. **Data caching:**

   In the absence of a standard solution for applying secure virtual memory, data caching needs to be implemented otherwise, especially related

to scheduling. The program state, subroutine input and output parameters, big cryptographic keys and other data may need to be at least temporarily cached in insecure memory during program execution. These data caches must be cryptographically protected for confidentiality and integrity, as well as bound to mechanisms protecting against rollback and data re-ordering.

5. **Provisioning:**

   Trusted applications and data for them need to be provisioned to the device from a remote location. Both the data and the program code may need to be protected for confidentiality in addition to providing integrity guarantees. It should be possible to uniquely bind secrets usage to a specific set of trusted applications. Trusted applications originating from the same author or such that are otherwise associated may need to construct a device-local security domain in which secrets are shared among all applications assigned to the domain.

Chapter 8

Theses

Low-cost trusted hardware is becoming widespread especially in mobile devices but also in the personal computer context. These environments lack in many significant areas such as non-volatile storage, statefulness and size of secure memory.

It is not difficult to pinpoint many specific security functions and services that are straight-forward to implement with PSEs, and such services have already significantly advanced the level of security for devices where they have been deployed. Mobile phone manufacturers have successfully used these environments for integrity protecting software (secure boot), forcing the binding between an operator and a device for the duration of a subsidized contract (subsidy locking), enforcing operator-specific variations in device software and asserting device identity. However, in this thesis we aspire to go further, to make full third-party secure service development available. This includes open programmability, provisioning and the ability of the TEE to support any number of such third-party programs deployed on the device. In this setting, the constraints of the baseline TEE environments as described in Chapter 6 become apparent. A number of new requirements have to be catered for, and software primitives need to be formulated to accommodate the scarcity of memory available for PSE-managed isolation.

Researching mechanisms for achieving goals that also can be met by existing discrete hardware, like smart cards, is motivated by efficiency and cost. Figure 8.1 illustrates the problem space: Programmable secure environments can be considered bound by the aspects of price, achieved security level and overall performance — generally it is hard to satisfy more than two of these constraints at once. On a high level this thesis sets out to increase the performance of the end solution without sacrificing either the security level or the low price point of the default configuration.

With this motivation, the following research goal defines the scope of this thesis:

We aim to define a software and hardware infrastructure around a commonly available PSE to enable secure third party use and programmability. We do this by adding at least partial support for off-line roll-back protection and scheduling in the untrusted domain for secure data caching. We aim to construct a system that in

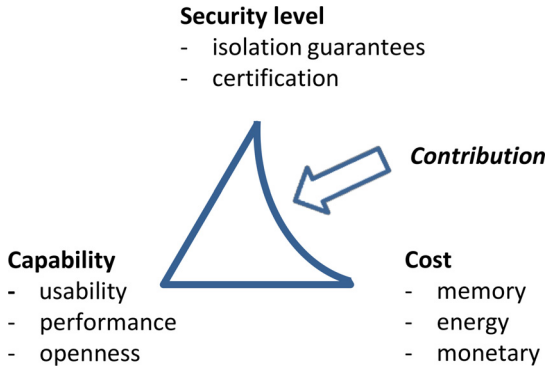

Figure 8.1: Research motivation

terms of isolation and performance approaches the level of security provided by dedicated security hardware — without the additional hardware cost. The provisioning of data and code to these environments shall be possible to all application developers, without constraints on who can issue the trusted applications.

8.1 Detailed research goals

In this section, the contribution of this thesis is summarized by highlighting the research questions which the individual research publications solve. We take a bottom-up approach, i.e., we begin by examining trust roots and secure device boot-up and end by discussing trusted application provisioning and isolation domains.

Compared to operations executed in stand-alone or self-contained security elements like cryptographic co-processors or smart cards, it is more difficult to reason about the security level for software running under the protection of a PSE. A set of requirements of the underlying platform need to be defined to measure the achievable trustworthiness in some quantifiable manner:

> **Research question 1:** What hardware and software *trust roots* are necessary to set up a functioning TEE on a PSE?

The Trusted Computing Group has for the TPM[94] chip done seminal work of defining the concept of a *Root-Of-Trust* in the meaning that we will be considering here. Although the TPM[94] is a physical chip, its correct operation relies on an external security function, the Core Root of Trust for Measurement (CRTM). The Mobile Trusted Module (MTM) specification defines a collection of security operations similar to the TPM, but does not necessarily rely on being physically instantiated on a chip. Instead it allows for a software implementation within a PSE, provided that a set of trust roots for MTM engine verification, stateful storage and integrity of external trust roots for secure boot are available. Publication **P1** outlines these roots in the

context of contemporary processor security environments.

Secure boot was outlined in Section 4.2. A simple implementation of secure boot relies on an external trust root, usually a signature key pair K_{priv} / K_{pub}. Only the sequence of programs signed by this trust root may be executed on the system.

This kind of secure boot is very inflexible. It makes software upgrades difficult. Another problem is that modern computers and mobile phones include drivers and other software from many vendors, which need to be able to publish and update them independently.

Research question 2: How can secure boot be made more flexible, allowing software patching, conditional and parallel execution and variable ordering of the code in the boot sequence as well as constrained delegation of trust to other software publishers?

Publication **P1** describes a secure boot architecture based on MTM commands. The first goal is to implement roll-back protection of code updates. Additionally, the architecture supports parallelization and re-ordering of the boot sequence. The latter is needed if many processors securely boot up a system in parallel or if the boot sequence is conditionally executed depending, for example, on user input.

In order to support multiple software vendors, the architecture also allows delegation of authority from the single trust root to other public keys. This delegation can be constrained so that each vendor is able to only authorize code for a specific part of the boot sequence. For example, execution of code from the vendor may be conditional on other components already run as part of the boot-up. This mechanism can also be used to check for the presence of required libraries, interfaces or even software licenses.

For rollback protection, contemporary PSEs have little integrated support, as was discussed in Section 6.2. Since rollback protection is required for many services, it needs to be externalized with respect to the secure environment. As the PSE has read-only nonvolatile secure storage for key material, a secure session between the PSE and the external component can be set up based on such key information. This session must account for manufacturing requirements like key set-up, chip testing and device service where either component has to be replaced:

Research question 3: Can we move the persistent state for the PSE to an external memory? How is the secure context between the PSE and this external memory set up, especially as part of manufacturing, testing and device care requirements?

Publication **P2** presents a hardware design for persistent, non-volatile storage, where the logic of the external hardware component allows for lifecycle management of its secure session setup and storage. This includes support for chip testing at the factory and provisioning secrets at early stages of production, support for chip replacement at service points as well as a possibility for fault analysis in case the hardware component's failure and return rates

turn out to be unusually high.

Such procedures are necessary from the point of view of deployability and quality assurance in large-scale manufacturing, and non-trivial to achieve when combined with the security and integrity requirements of the storage and the need to protect the device user's privacy. As a practical concern, **P2** also emphasizes the cost minimization of the external storage. Since the overall rollback protection functionality is divided between the TEE and the storage component, we minimize the hardware logic in the external component side whenever possible, at the cost of adding more logic in software inside the TEE.

For the topic of rollback protection, **P2** also highlights a very different design issue. Flash memory cells are susceptible to *wear-out*, i.e., the same memory location can be written to only a certain number of times before the cell loses its ability to maintain the written value without read errors. In dedicated storage components like memory-based hard drives, such anomalies are managed by complex algorithms that balance the cell wear-out. Our design has to work without the support of such algorithms and flash cells that are guaranteed to support only a very low number of re-writes:

> **Research question 4:** How is storage cell wear-out best considered in the context of non-volatile storage for roll-back protection data? Can the success probability of an attacker attempting to exploit wear-out using exhaustive attacks be mitigated?

Since the architecture of **P2** relies on a TEE with RAM, we do add, in the TEE, data integrity checking and algorithms to balance wear-out while also allowing us to identify when a cell has been worn out and returns unreliable data.

We can also use the TEE for non-volatile memory wear-out protection during a single boot-cycle by not committing every data update that relies on the rollback protection all the way to persistent storage. To thwart simple hardware attacks, this insight requires, in the proposed architecture, that the protocol between the TEE and the storage is encrypted so that an eavesdropper observing the communication cannot determine whether a persistent storage commit was done or not. The protocol in **P2** fulfills this property, and is further discussed in Section 9.3. With this property in place, the frequency of actual data commits to flash can be dependent on how much the storage has already been used up, i.e., how close we are to a promised number of write cycles with no errors, the time between events that require commits and how many commits all the way to flash memory have been done since the last boot-up. This statistical optimization is another example of how an architecture can be made to perform even when cost needs to be minimized — in this case in terms of the amount and quality of persistent memory. The discussion is only briefly touched on in publication **P2** due to publication size constrains, but further discussion is available in Section 9.3.

Random Access Memory (RAM) in the CPU is costly, and if parts of it are dedicated to only security, like the on-chip memory reserved for the TEE, the memory amount provided for this purpose in mass-produced devices will be

put under careful scrutiny. As a result, any TEE code design must be made to fit into a computing environment with an extremely challenging memory resource size, often as low as 10-15 kB. Well established TEE programs that are designed and implemented at the time of chip manufacturing can partially rely on being run from ROM but for any new or evolving logic the RAM budget is a constraint - the code, its heap and stack must all fit in the RAM reserved for the TEE.

One option to extend the TEE isolation domain is to securely page data between the secure and insecure memory. However, the needed code for implementing this operation in a general way, especially with integrity protection, may itself be larger than the available secure side RAM. Publication **P3** presents a design where a standardized feature set (the Mobile Trusted Module specification) is implemented as a set of TEE programs with a minimal memory footprint. The publication mainly addresses the following issue:

> **Research question 5:** For resource-constrained PSEs, how do we securely assign a shared data state to a set of distinct programs that as a collective implement a function set or service? What are the necessary trust roots in such an architecture?

In addition, publication **P3** provides a reference implementation of the complete TCG Mobile Trusted Module (MTM) functionality on a TEE. Measurements included in **P3** shows the performance benefits of taking the TEE approach, when compared to stand-alone chips. Stand-alone hardware TPM/MTM chips run at lower clock speeds than the PSE (running at the full speed of the main processor core), and thus the cryptographic performance of stand-alone TPMs is significantly worse. On the other hand, a TEE-based MTM implementation needs to use encryption for preserving its long-term state. Even with this shortcoming, the TEE MTM implementation is significantly faster for MTM commands that execute complex cryptography like digital signing or key generation. Publication **P3** successfully argues that a TEE implementation of MTM functionally is a viable alternative to stand-alone security logic.

Publication **P4** describes an architecture for piecewise program provisioning as well as for the scheduling and temporal caching of the executing trusted application and its data. The secure scheduling happens under the supervision of an untrusted entity in insecure memory. This architecture may at first glance seem contrived, since a PSE-internal OS with trusted application isolation can utilize memory management and processor context isolation features provided by the processing core to achieve this. However, footprints of contemporary "minimal" trusted OSs or microvisors[44][81], i.e., the size of their code and data structures, even without the secure data scheduling, today exceeds available secure hardware memory by at least 400%, often by significantly more. To achieve this, we are faced with an extension to research question 5:

Research question 6: In a PSE where the trusted application code and data need to be provisioned and executed in a piecewise manner, what cryptographically protected data structures can be applied to guarantee the integrity, rollback protection and confidentiality of the code and data during the execution session?

The previous contributions all rely on the interaction between logic inside the PSE and logic elsewhere, whether in untrusted memory or in another TCB, like the non-volatile secure storage chip. Especially in interaction with untrusted memory, any cached data needs to be both confidentiality and integrity protected in addition to maintaining rollback protection for it. A typical and well-analyzed cryptographic tool to achieve confidentiality and integrity data protection at the same time are so called authenticated encryption (AE) primitives. In a constrained PSE, which as a rule can address both trusted and untrusted memory at the same time, it is tempting to use AE primitives to stream data from the secure environment into untrusted memory, and encrypt it on the fly. The opposite operation, decryption into the secure environment, is of course equally useful. This need caused us to examine a very specific issue:

Research question 7: Are authenticated encryption primitives secure in a model where encryption and decryption is done as a stream, and partial encryption and decryption results are by necessity exposed to a potential attacker before the integrity check calculation is completed?

The answer to the question is not in all cases obvious, and the existing security proofs for AE primitives are done in a model where the operations, i.e., encryption or decryption including the integrity check generation or validation are done as an atomic activity rather that as a stream. Publication **P5** explores this problem, both in the PSE use case described above, but also in a use-case where a co-processor provides the AE service for streamed data. With some minor constraints we formally prove the streaming mode to be as secure as the baseline model for one AE primitive (AES-EAX). We also report on possible implementation pitfalls that need to be avoided when applying AE for streaming data and point out that a recently standardized PSE interface, the GP Device Internal API[42], provides an AE API that is non-ideal in this respect. Its uninformed use may open up unintentional attack vectors when AE is used.

Publications **P1**-**P5** collectively provide a framework in which different aspects of executing code securely in a memory and feature-constrained PSE can be achieved. The MTM logic is one example of a feature set that leveraged the extra protection and freedom of implementation provided by this framework.

Publication **P6** approaches third-party code provisioning and security domain setup from the perspective of applications and in the context of MTM and TCG technologies. Here, the initial assumption is that independent third parties want to write and distribute their own applications (mobile apps) to devices, and such applications need to leverage baseline MTM services like

secure storage, cryptographic key generation and use, and possibly attestation. We also assume that the third parties operate in isolation from each other i.e., we do not consider it feasible that applications would share a single MTM resource and collectively manage keys and state updates. Instead we take the approach that every application gets its own MTM resource that will be executed in the PSE whenever the application requests for it. The need for isolating these MTM resources from each other is evident. But we also need to uniquely bind a MTM instance to an OS application or resource:

> **Research question 8:** How are trusted applications inside the TEE uniquely bound to applications in the operating system, and how can this fact be reliably attested to third parties?

In **P6** the used PSE is the AMD SVM[6][85], with the Flicker[63] architecture described in Section 6.8.3, i.e., a TCG late-launched environment in Linux.

Application identification is done with the Integrity Measurement Architecture (IMA)[77]. The isolation between domains is constructed around a cryptographic key, which is bound to an underlying hardware TPM. This key is included in necessary x86 late-launch platform measurements and used as a decryption key for PSE application state (in this case the MTM state). In other words, the combination of TPM platform measurements for a) the PSE code being launched, b) the security context identified by a key, and c) the OS application currently running in a late-launched environment are used to construct an isolated (data) security domain for a third party application. Although the implementation in this case is specific to TCG standards, the threat model and overall architecture is applicable to any PSE.

The MTM implementation above exposes another shortcoming, visible when the configuration for the function sets and their shared data needs to be set up. Even though TPMs can bind code measurements to sealed data or key usage, it is difficult to use TPM functionality to set up security domains for several pieces of such code to share secrets. This issue is not limited to MTM: any program set running with the TPM late-launch methodology as the PSE fundament and needing to share some secret or protected state faces the same issue.

For this purpose, publication **P6** introduces an enabler, the **setup PAL**, where PAL is the Flicker term for a trusted application. The setup PAL contains a hardcoded trust root from which signed statements are received, containing the measurements of other TAs (PALs) that are to become part of the security domain. Using this information, the setup PAL defines a local security domain based on a symmetric key and binds that key to future measured launches of the PALs that are to be part of the security domain. Even though the design in **P6** is TCG late launch specific, it in no way is limited to the MTM function set — this solution is applicable to any set of PALs that would operate on behalf of a remote party:

Research question 9: On the assumption that trusted applications can be remotely provisioned on behalf of a third party, what are the TCG/TPM mechanisms by which a remote party can define and maintain a secure domain for provisioned sets of code and data as well as content locally generated by the provisioned code?

The complexity of code provisioning mechanisms as described in publication **P6** with TPM late launch mechanisms leads to the question whether open provisioning could be designed better, if we would have the freedom of choice also for the TEE design?

Research question 10: In open provisioning, the provisioning party has the freedom to define, for his own use, the security domain and which code and data will belong to it, without any constraints from a centralized issuer. How is such a security domain set up in a logical manner, if we also account for code updates from the provisioning entity and possible code (subroutine) sharing between security domains?

The code provisioning problem and code isolation for such provisioned code, is explored in publication **P7**. The On-Board Credentials architecture presented in publication **P7** was also earlier presented in a larger technical report[32] and a poster session[11] by the same authors. It is also further summarized in Section 9.1. Like Flicker[63], ObC will execute third party credentials sequentially, one at a time, and use this for isolation between credentials. However, unlike Flicker, which uses x86 hardware and TPM for isolation, ObC isolates third party provisioned, untrusted code against the underlying trust roots in software inside the PSE. This can be formulated as:

Research question 11: In a PSE that executes third party code, how is that code, which is untrusted from the PSE perspective, isolated from trust roots and critical system resources that make up the overall PSE trust domain?

In the absence of hardware mechanisms to provide isolation, some virtualization will be needed for the third-party code. The practical non-applicability of off-the shelf hypervisors and trusted OSs for this task was discussed in publication **P4**. Instead, ObC relies on a PSE-internal interpreter to isolate third-party ObC PSE code from the internal trust roots (keys) in the PSE. Parts of this byte-code interpreter, designed for the specific purpose of isolation, is formally validated (see Section 9.1.2).

Chapter 9

Overview of the software architectures

This chapter presents an overview of the individual solutions that have been developed as parts of thesis, to complement the often brief introductions available in the research publications. In most cases also additional research work is presented — protocols and research aspects that did not fit in the original publications or are subject to future public dissemination. In its own section, the practical impact of these architectures is discussed.

The main effort of the thesis work focuses on constructing an environment for third-party program execution, not only third party access to a set of functions. Publications **P5**, **P6** and **P7** together describe the inner workings of On-board Credentials — an interpreted environment, where the interpreter itself, the code it executes as well as the data of the code being executed is processed in pieces inside the PSE, operated by a driver engine in the OS. The contributions together assert the security properties of this arrangement, i.e., the security and isolation of the execution of the third-party credential.

Additionally, two validation topics are presented as part of this chapter. We explore the formal validation of necessary parts of the ObC software. Publication **P4** already sets the fundament for arguing about the correctness of the sealing and unsealing cryptographic primitives, but Section 9.1.2 builds further on the argument and discusses work done towards proving the correctness of the ObC architecture. Section 9.4 presents one specific application example — a ticketing protocol and system which for its security relies on a programmable secure environment like the one being provided by ObC. The system makes use of the specific features provided in the thesis context: The ticketing algorithm needs to be executed in isolation, it originates from a third party, and it will require isolation, programmability, remote data provisioning and rollback-protected secure storage. Also the third party and the customer have a shared incentive to authenticate or attest the security of the environment and the level of its roots of trust. The baseline protocol is provided in publication **P8**, but we also explore ticketing system extensions that do provide a migration path from mobile devices without an integrated feature like ObC.

The chapter content that has not been peer reviewed include the formal validation conducted for ObC in Section 9.1.2 and the detailed description

of the probabilistic update process for the non-volatile storage presented in Section 9.3 which did not fit in the page allocation for the publication. The filter based mechanism to identify double-spending for identity-based ticketing described in Section 9.4 has been separately submitted for publication.

9.1 On-board Credentials

Much of the work presented in the attached academic publications relates to an architecture designed and implemented at Nokia Research Center during the years 2005-2012. The architecture was mainly designed for two TEEs available in mobile phones, the Texas Instruments' M-Shield and ARM TrustZone, although research instantiations of the design have over the years been ported onto several other platforms including a "secure kernel" design with a TPM for storage, the Flicker architecture on AMD processors, at least one hypervisor, and even onto smart cards. The thesis author is the main architect and implementor of the ObC baseline that constitutes the PSE/TEE code and its internal as well external interfaces.

The starting point for the On-board Credentials project was to explore the potential for using mobile device TEEs to provide security benefit to third party services, and thereby the end user. The architecture was also seen as a means to achieve portability of legacy TEE code across processor architectures. Being a proprietary design, and in relying on PSEs in handsets, which only to a limited degree are tamper-resistant, ObC is designed to fill a niche where the security service deployed within it does not motivate the inclusion of a high-cost secure element in the mobile device but where the services still are an attractive enough attack target to require the extra protection ObC provides. As stated earlier, the TEEs in contemporary handsets govern mechanisms like asserting the non-modifiability of device identities and for providing DRM mechanisms and subsidy locks. I.e., the business incentives for securing these services are high from the perspective of device manufacturers, operators and content providers. That the PSE/TEE used by ObC successfully has been used for manufacturer services gives some assurance of the achievable security.

At the heart of the On-board Credentials architecture lies its TEE-side interpreter. The default choice for a TEE — a JavaCard interpreter — was ruled out in 2005 due to the available evidence based on the ROM/RAM requirements of virtual machines. Instead, the bytecode originates from a subset of Lua 2.4[71] bytecode which was chosen as the starting point for the project, although quite soon the bytecode diverged from its origin due to the specific needs of the platform. Code portability is mostly a side-effect of the interpreter design, its main function is isolation — of TEE secrets and state with respect to the programs executed within the interpreter.

Functionally, the major constraint behind most ObC design decisions has consistently been the scarcity of PSE isolated memory, and much of the work has revolved around the problem to achieve security when burdened with this constraint. A straight-forward model where a trusted application code and secrets are first uploaded to the secure environment, then execution happens, and finally results are returned has often not been possible. Since the main target PSEs can address insecure memory also, another solution could have

been to extend the secure memory by designing a concept for secure virtual memory, where memory pages are encrypted and temporarily stored in insecure memory. This architecture more or less requires the presence, in secure memory, of a scheduler or small OS that supports secure virtual memory, and uses the MMU and page-fault interrupts to trigger this activity. Unfortunately, the existing TEE based software environments for now are mainly geared towards secure function execution (IRQs and DMA are often turned off for security reasons during secure mode invocation), and to implement a scheduler or hypervisor and setting it up would exhaust, or often simply not fit in the available 10-20 kB isolated RAM memory.

Instead, the ObC core is built as a scheduling system where the ObC functionality is built as a set of "remote procedure calls", in fact spread out over many TEE applications. The scheduling between these applications as well as managing the encrypted state of the currently interpreted ObC program is left to a scheduler operating as a driver inside the untrusted OS. Figure 9.1 shows the overall setup of ObC. The scheduler will receive the ObC program invocation, i.e., the locally encrypted program code, and any possible inputs, including input parameters, and storage data sealed by a previous invocation of the code. The scheduler temporarily stores all this information, and then invokes the bytecode interpreter, i.e., uploads the appropriate TEE program piece for ObC program upload and execution initialization, and also gives the encrypted ObC program bytecode as input for that TEE program. On successful ObC bytecode decryption, the interpreter starts executing, but for many of the more complex bytecodes, one of several events leading to scheduling will happen. Such events include bytecode requesting an input parameter, or bytecode needing to invoke a library function, or in some platforms even the case where individual bytecodes handled by a different TEE program is encountered. The event request will cause the interpreter to collect its runtime state (including the virtual machine's program counter, local variables and stack), encrypt it, and return to the OS scheduler. Furthermore it will indicate in a return parameter what the reason for the scheduling event was. Consequently, the scheduler will act on this information and re-invoke the same or different TEE program, possibly with some of the temporarily stored data, and if it is a re-invocation of the interpreter, the stored state[1]. In this manner the overall bytecode execution will continue, until the next ObC bytecode returns an error or a complete indication. The ObC API caller typically has to indicate which return parameters it is interested in, so on successful bytecode termination, the scheduler will look in its temporary store (dynamic state), and dig out the parameters that are returned to the caller. All other variables in the store are considered temporary in nature and destroyed at return to the untrusted application context.

This way of operating causes a significant processing overhead to the execution of the ObC programs, due to the often numerous context switches and corresponding encryption of state that occurs over the execution of a single ObC program. However, since the PSE system runs at the full processing

[1] E.g., if the interpreter bytecode invokes a bytecode library call entry, the scheduler will stack the calling interpreter state, and launch the invoked call in the interpreter with a clean state context. The stacked interpreter state is restored after the library call execution successfully terminates, and the scheduler re-invokes the calling code execution — which presumably starts by collecting results from the library call.

speed of the processor, we do see that ObC program execution speed (with the overhead) is comparable to that of smart card applications which are limited by the lower clock frequency of the cards. Publication **P3** provides some measurements to assert this fact. The frequent returns to the scheduling operation inside the OS also has the advantage that OS IRQs and even DMA will be served during those intervals — if this would not be the case, the user might perceive a lengthy ObC invocation as a temporary device freeze, since at the times the TEE is used with no DMA and IRQ handling, the rest of the device is on hold and no user or external I/O can happen.

In addition to the scheduling via the OS, the interpreter also implements simple virtual memory management for bytecode pages. The bytecode encryption is done in 64-byte units, with integrity checks and a plaintext header indicating the relative position of the code page in the overall code. Currently the interpreter reserves internal space for four pages (256B) to reside in secure memory at a given time, and memory pages are decrypted into secure memory in an on-demand manner without rescheduling through the OS — for a single ObC program, all encrypted code pages are always available to the secure-side TEE program by memory reference. The virtual-memory support for code-page aging is minimal, and constitutes 194 bytes of compiled C-code. Overall, minimization of the interpreter/TEE code is obviously crucial to the architecture, and thus ObC uses only a single, shared AEAD encryption primitive (AES-EAX) for all its operational security, data formatting (type-length-value encoding) is common over all data. These design decisions not only saves algorithm and parsing space, but also makes integrity checks, rollback protection, memory buffer checks, and attribute conversions common throughout the implementation.

The bytecode set of the ObC interpreter to some degree reflects on the environment in which ObC trusted application are run. In addition to a very standard set of functions like arithmetic operations, memory (variable) access and jumps, mostly inherited from the LUA bytecode set, the bytecodes include functions for sealing or unsealing a variable. These commands include parametrization for a variety of security contexts. The sealing can happen in a storage context, where the program stores information for itself persistently. It can also be done in a subroutine context, where the encrypted data is bound to the currently executing session so that another compiled ObC program (a kind of a subroutine or DLL "heap"). Sealed data always enjoys confidentiality and integrity protection. For run-time seals, like temporary cached subroutine I/O, also rollback protection is provided as part of the seal. Other special types of seal contexts include externally provisioned secrets and contexts usable for temporary caching of data during one ObC program invocation — a sort of code-initiated virtual memory for data arrays. All complex library function APIs provided to the ObC interpreter, like asymmetric key generation and use, hash algorithms or random number sources are also accessed through a sealed "DLL interface" via the insecure scheduler, orchestrated by the ObC program through a set of bytecodes.

The ObC interpreter has a few esoteric bytecodes that relate to the very specific security properties provided by the platform. One is the counter interface, which provides, to the executing ObC program, a way to increment and read non-volatile system counters provided by the platform, where available. Using this primitive, a bytecode program can for itself build rollback

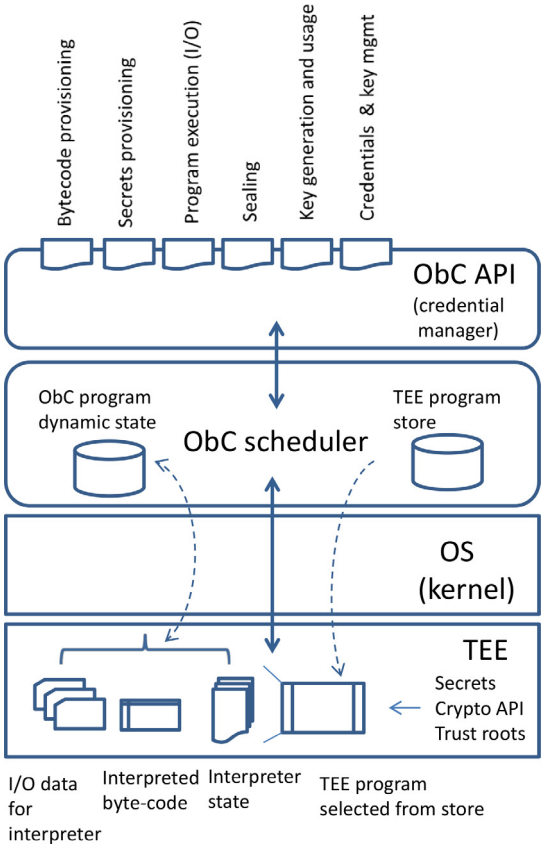

Figure 9.1: On-board Credentials architecture

protection between device boot cycles. Another bytecode is the endorsement of another ObC trusted application to the same security domain as the currently running application. This means that an ObC program locally on the device can attach other programs (or bytecode libraries) to its own security domain. This functionality is needed for constructing non-remotely managed security contexts, as opposed to remotely configured ones. Such contexts can be used if the terminal device itself is responsible for provisioning code or data to other devices with ObC. They can also be used for distributing secrets between devices. Furthermore, an extension bytecode set with privilege escalation also exists for programs originating from the device manufacturer — this is a platform specific way to also enable the writing of system services as bytecode.

For bytecode programs, ObC has its own provisioning model, which has been minted "open" in some of the public materials. The provisioning model is not in the core of this thesis, but in short it relies on the following basic elements: A certified public platform key, akin to the endorsement credential in TCG[94] can be used by a third party to determine that the device contains an ObC environment and that the key is bound to that environment. Where, in TCG/TPM protocols, that certified key is used to encrypt data in a protocol used for binding privacy keys for attestation, in ObC it is used to encrypt a symmetric encryption key, originating from the server, that will come to define, in the target device, the security domain of the programs and secrets originating from the server. In the device, the ObC provisioning system will input, but never reveal, this domain key encrypted for it, and will further assign all programs and secrets encrypted with the domain key into the security domain defined by it.

9.1.1 ObC development environment

As the first bytecode interpreter was written directly for the Lua bytecode, the first ObC programs were written in Lua 2.4[71], and a standard Lua compiler was used to produce the bytecode. The support for on-demand loaded subroutine modules (akin to dynamically loaded libraries), with the related support for integrity, confidentiality and rollback protection for the parameters and responses channeled through the untrusted OS caused the bytecode to evolve in directions where the LUA language would have needed significant extensions to support the evolving virtual machine design. Instead, a dedicated macro assembler was designed for the bytecode. Finally, the toolchain was completed by adding call-frame bytecodes to the virtual machine and writing a fairly standard BASIC language compiler to the environment and leveraging the C preprocessor for including byte-code assembler snippets to the compiled binary as a form of STDIO library functionality. A bytecode emulator and debugger is also available to test the developed algorithms in isolation on a PC.

The following simple example provides an insight to how ObC programs look like. The example code implements password-protected sealed storage for any caller. We see that environment I/O is performed using subroutines, and the logic is in plain BASIC. The word size for the ObC interpreter is 16 bits. Automatically allocated word arrays are supported as part of the language and bytecode, and byte-referencing half-words (bytes) in arrays is

possible.

```
rem ##########################################################################
rem # A program in evo that implements a sealing functionality based on
rem # an application-specific identifier (secret / password) provided
rem # by a driver
rem #
rem # Compile and run this in the development env e.g. with
rem #
rem #  'bin/compile_run_sealed.sh evoexamples/seal.evo + +'
rem #
rem ##########################################################################

rem # Declarations must come before includes
declare array iarr 10
declare integer rint 11

#include "io_codes.evoh"
#include "program_io.evoh"
#include "arrays.evoh"
#include "arraycmp.evoh"

rem --- Declare the "name" of the array

function main()

    dim mode as integer
    dim j as integer
    dim password as array
    dim data as array

    rem --- The system operates based on a 'mode' identifier which is the
    rem --- first argument always. The values are
    rem --- 0x0000: Insert new password + sealed data
    rem --- 0x0001: Recover sealed data based on the password
    rem ---
    rem --- NOTE! The password should always be 16 bytes / 8 words.
    rem --- The driver should make sure of this, e.g. by hashing

    mode = read_integer(IO_PLAIN_RW, 0)
    read_array(IO_PLAIN_RW, 1, password)
    j = alen password
    if j != 8
      return 0
    end
    if mode == 0
      read_array(IO_PLAIN_RW, 2, data)
      append_array(password, data)
      write_array(IO_SEALED_RW, 2, password)
    else
      read_array(IO_SEALED_RW, 2, data)
      j = array_part_match(password, data, 0, 8)
      if j == 0
        return 0
      end
      j = alen data
      j = j - 8
      copy_array_part(password, data, 8, j)
      write_array(IO_PLAIN_RW, 2, password)
    end
    return 1
end
```

9.1.2 ObC validation

The security validation of the ObC concept is a multi-faceted issue which has been approached from a number of angles over the years. Principles of the provisioning protocol as well as secrets backup and migration has been validated with the AVISPA tool[10]. Some of these proofs have been reported

on in [54]. Publication **P4** argues about the formal correctness of applying authenticated encryption for sealing in the most relevant hardware instantiation. Both of these implicitly rely on work conducted to list all deployed key diversifications and seal header content patterns to ascertain that these values are unique to each seal usage, so that replay of sealed data from the untrusted OS side out of context will not be an attack vector.

In the ObC architecture, no issuer control is enforced on provisioned code. This means that the ObC interpreter bytecode interface must withstand all attacks at the bytecode level. The interpreter design attempts to alleviate this risk in the following ways:

1. A weighted count of interpreter execution steps[2] is maintained for the duration of an interpreted program. When the count reaches a predefined maximum, execution will be terminated. When a lower sub-limit is reached, execution is temporarily halted for OS interrupt handling. These measures guarantee that OS mechanisms are given the opportunity to abort inappropriately time-consuming computation or infinite loops.

2. Before each bytecode execution, the placement of the bytecode in the program text is examined to ascertain that the remaining space is enough to accommodate the longest bytecode possible (3 bytes).

3. Data is kept separate from the program text, and structured as an array store rather than memory accesses.

4. General computation is virtually limited to stack-based operation. This simplifies the bytecode language significantly. Only a few specific register arithmetic operations (2) are provided to enable efficient stack-frame operation for programs compiled from BASIC more size-efficient.

5. All stack and storage accesses in the interpreter code is channeled through 4 interfaces (PUSH / POP and STORE / GET). The code / logic implementing these interfaces has been formally proved correct against boundary conditions with Event-B[3] and the Rodin platform[4].

The B methodology[2] for system analysis was developed by Jean-Raymond Abrial in the 1990's. It has been applied for safety critical software systems, e.g., the automation for the 14th Paris Metro line[14]. Event B[3] is an evolution of the B methodology for which the open toolset Rodin[4] has been developed in a EU research project. The code that maintains the dynamic data of ObC programs, i.e., the stack and data store have been formally modeled using the Rodin tool, and proven correct against intentional or unintentional overflow. Abstractly the Rodin tool inputs a system description, in the form of a set of events. For the purpose of this discussion, an event can be a function or a method in the code that we are validating. The description of an event includes guards, i.e., assertions regarding the correct behavior of the event. The tool includes descriptions regarding parameters, their types and possible range limitations. What Rodin can prove is that the system never can be run into a state where the assertions fail.

[2]The count is weighted based on consumed time, i.e., invoking sealing / unsealing primitive costs more than executing a JMP operation.

Event *add_global_tuple_existing* $\widehat{=}$

any

id1
index
value
s s will be the index of the variable which has the identifier id1

where

grd1 : $id1 \in \mathbb{N}$
grd2 : $index \in \mathbb{N}$
grd3 : $value \in \mathbb{N}$
grd4 : $id1 \neq 0$
0 is reserved for free variable slots
grd5 : $id1 \in ran(variable\_name)$
a variable with the name id1 must already exist
grd6 : $s \in dom(variable\_name)$
s must be within bounds for the indexes of variable_name
grd7 : $(\exists x \cdot x = s \wedge$
$variable\_name(x) = id1 \wedge$
$(\forall y \cdot y \neq s \wedge y \in dom(variable\_name) \wedge variable\_name(y) \neq id1))$
s is the index of the variable whose identifier corresponds with id1. there must be no other variables with the same identifier.

then

act1 : $variable\_len(s) := variable\_len(s) + 1$
act2 : $elements :\in \{\{(a \mapsto b) | (a \mapsto b) \in (1 \mathinner{..} (variable\_start(s) + index - 1) \lhd elements)\} \cup$
$\{(variable\_start(s) + index) \mapsto value\} \cup$
$\{((c + 1) \mapsto d) | (c \mapsto d) \in ((variable\_start(s) + index) \mathinner{..} (ELEMENT\_CNT - 1) \lhd elements)\}\}$

The second part is the new relation we want to insert.
The third part is the set of relations which are located from the inserted element to the end elements set. Indexes for the third part are shifted forward by one.
act3 : $variable\_start :\in \{\{(a \mapsto b) | (a \mapsto b) \in ((1 \mathinner{..} s) \lhd variable\_start)\} \cup$
$\{(c \mapsto d + 1) | (c \mapsto d) \in ((s + 1) \mathinner{..} VARIABLE\_CNT) \lhd variable\_start\}\}$
The first part of this union is the set of relations which have not been affected by the shift done in act2. The second part of the union is the set of relations which were affected, and their pointers are increased by one.

end

Figure 9.2: A detail from an Event-B proof

From the abstract function descriptions, we carefully designed C code to diverge as little as possible from the formal model event. Figure 9.2 shows one detail from one completed proof of the function adding a data (array) element to the data store of the interpreter. In Figure 9.3 the corresponding C code is presented, and the relation between these two should be obvious.

Furthermore, the dataflow of the interpreter bytecode implementation has been individually analyzed against boundary and overflow issues. All I/O buffers that channel information to and from the interpreter as well as all temporary buffers that may hold information of non-predetermined fixed length are interfaced with a macro language that guarantees overflow and underflow protection along with the distinction between accessing untrusted and trusted memory, wherever this is needed.

Even in combination, these efforts cannot provide full guarantees of the structural integrity of the interpreter codebase. Furthermore, the interpreter uses cryptographic primitives which have not been validated as part of the ObC project. Bootstrap code will authenticate the ObC interpreter and allow it to run in the TEE environment — this code is also outside the scope of the ObC itself. Last, the TEE parts of the ObC project consisting of the interpreter, I/O primitives, scheduling support and crypto interfaces is written

```
void add_global_tuple(TUPLESTATE *state,/* State vector    */
            unsigned short id,      /* Variable name: Input  */
            unsigned short index,   /* Element index: Input  */
            unsigned short value,   /* Element value: Input  */
            unsigned long *i_err)   /* Error parameter       */
{
  unsigned char *c;
  int empty_pos = -1;
  int found = -1;
  unsigned short max_pos = 0;
  int shift, i;
  unsigned int current_end;

  if(*i_err) return;
  if(id == 0) { *i_err = 0x1401;  return; }
  for(i=0; i < VARIABLE_CNT; i++) {
    if(id == state->variables[i].variable_name)
      found = i;
    if((empty_pos == -1) && ((0 == state->variables[i].variable_name)))
      empty_pos = i;
    if(max_pos < (state->variables[i].variable_start +
                  (state->variables[i].variable_len))) {
      max_pos = state->variables[i].variable_start +
        (state->variables[i].variable_len);
    }
  }
  if((-1 == found) && (-1 == empty_pos)) {*i_err = 0x1402; return;}
  if((-1 == found) && (empty_pos >= 0)) {
    state->variables[empty_pos].variable_name = (unsigned char)id;
    if(empty_pos == 0) {
      state->variables[empty_pos].variable_start = 0;
    } else {
      state->variables[empty_pos].variable_start =
        state->variables[empty_pos-1].variable_start +
        (state->variables[empty_pos-1].variable_len);
      state->variables[empty_pos].variable_len = 0;
    }
    found = empty_pos;
  }
  if(index < (state->variables[found].variable_len +
              ((state->variables[found].variable_len) % 1))/ 2) {
    c = state->elements + state->variables[found].variable_start + (2*index);
    *c++ = (unsigned char)((value >> 8) & 0xFF);
    *c   = (unsigned char)((value) & 0xFF);
    return;
  }
  shift = 2 * ((index+1) - ((state->variables[found].variable_len +
                             ((state->variables[found].variable_len) % 1))/2));
  if((ELEMENT_CNT - max_pos) < shift) { *i_err = 0x1403; return; }
  current_end = state->variables[found].variable_start +
     (state->variables[found].variable_len +
              ((state->variables[found].variable_len) % 1));
  LIB_MEMMOVE(state->elements + current_end + shift,
              state->elements + current_end,
              ELEMENT_CNT-current_end-shift);
  LIB_MEMSET(state->elements + current_end, 0, shift);
  state->variables[found].variable_len = 2*(index+1);
  for(i=found+1; i < VARIABLE_CNT; i++) {
    if(state->variables[i].variable_name) {
      state->variables[i].variable_start = state->variables[i-1].variable_start +
         (state->variables[i-1].variable_len + ((state->variables[i-1].variable_len) % 1));
    }
  }
  c = state->elements + state->variables[found].variable_start + (2*index);
  *c++ = (unsigned char)((value >> 8) & 0xFF);
  *c   = (unsigned char)((value) & 0xFF);
}
```

Figure 9.3: A detail from C code corresponding to the Event-B proof

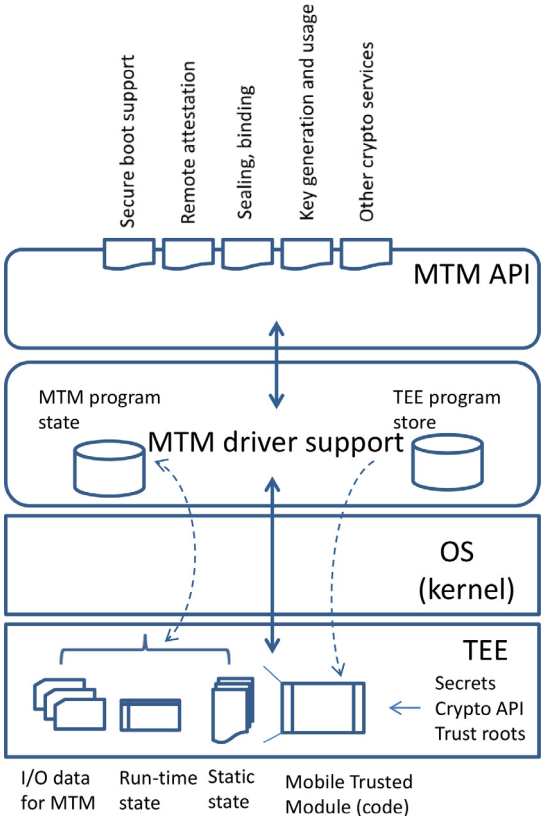

Figure 9.4: MTM / TEE architecture

in ANSI C, i.e., the compiler may well introduce additional vulnerabilities into the end result.

9.2 Mobile Trusted Module

The Mobile Trusted Module (MTM) specification started out as an adaptation of the TPM1.2 functions[94] for mobile devices. At the time the MTM work was initiated in 2005, some mobile devices were already equipped with TEEs, and thus one important incentive was to enable the use of the TPM interface on these. This affected the resulting standard in two ways: First, the mandatory command set of TPM1.2 had to be narrowed down, since the available RAM and ROM inside the TEEs was limited, as already was discussed in Section 9.1. Secondly, the isolation properties provided by the TPM1.2 when implemented as standalone chip now needed to be formalized as a set of additional roots of trust to be provided for the MTM by the PSE and TEE. MTMv1.0 specifies five of these roots, as described in publication **P1**, which must be provided by the underlying TEE to guarantee the overall security of MTM and its state.

Additionally, MTM adds support for secure boot, constructed using the system of PCRs and the authenticated boot sequence as a basis. The main

advantage of the MTM secure boot system is that it separates the authorization mechanism from the booted code measurements. In "traditional" secure boot this issue is often overlooked. All code checks based on external trust roots either share one or a few device specific public keys for all boot validations or use trust roots embedded in the code image doing the validation.

In MTM the secure boot process includes an external trust root hierarchy, using so called verification (public) keys, and signed assertions — so called reference integrity metric (RIM) certificates — to arrive at the continue or abort decision at any point during the secure boot sequence. The security-relevant processing of the keys and attribute certificates is internalized inside the MTM logic, i.e., the code partaking in secure boot does not need to contain the logic to decide on the viability of the boot flow, it only needs to abort if the MTM gives it that verdict. The key management for the secure boot as well as related signatures and certificates can be provided and managed by the device integrator, and code components taking part in the secure boot process ideally need only call a single function (MTM_verifyRIMCertandExtend) to determine whether the boot sequence should be aborted or not. Such a system clearly is useful when constructing OS- and stakeholder-agnostic, securely booting systems. This functionality is described in detail in publication **P1**.

The thesis publications provide a full set of Roots of Trust for an implementation of an MTM in the described TEE context. The Root of Trust for Measurement (RTM) derives from the legacy secure boot sequence and is thus integrity protected in a hardware-assisted manner. The Root of Trust for Storage (RTS) is provided by virtue of having secrets in the isolated domain from which storage secrets can be derived, a sealing implementation as described in the previous section or in publication **P6** and the rollback protection provided by publication **P2**. It is instructive to notice that the security of the three mandatory counters in MTM — one to protect against firmware rollback, one to protect against (MTM) secure boot policy rollback and one to protect against rollback of sealed objects produced by MTM — is implied by the RTS, as it should be. The legacy secure booting does provide sufficient protection for the Roots of Trust for Verification and Enforcement (RTV / RTE) whereas the RTS as described can provide necessary storage for the Root of Trust for Reporting (RTR), a signature key assigned into RTS at manufacturing time.

Figure 9.4 shows a generalized architecture where MTM is implemented as software inside a TEE. The figure exposes the similarities between implementing the standardized MTM functionality on a PSE compared to Figure9.1 and the ObC implementation. The closeness highlights the viability of the overall architecture — which for both ObC and MTM share the setup where a complex, insecure driver component residing in the untrusted operating system can be let to orchestrate the execution of secure functionality with little degradation of overall security properties. From the pictures one can also derive the main differences between the ObC and MTM architectures: ObC is geared towards code execution, and has only limited features for providing platform (OS) binding. MTM includes many features that can be used to improve the trustworthiness of the main OS, such as secure boot support, OS attestation and the aforementioned binding properties for keys and data, but it does not support code execution. This distinction shows the need for further development in the general TEE area - –there is no reason why isolated

domain programmability and platform binding need to be kept separate — in fact the features complement each other as is shown in publication **P5**.

Publication **P3** includes more detailed material regarding the MTM on PSE adaptation, including many performance figures.

9.3 Rollback protection with TCB-external non-volatile memory

The work towards using an external flash memory component together with a TEE for rollback protection is introduced in publication **P2**. The main objective is to provide a way for the TEE and trusted applications in the TEE to maintain state when targeted by off-line attacks (see Chapter 5). As a result, data that is managed by the TEE but stored encrypted and integrity protected outside the TEE can be bound to a counter or equivalent in the storage component dedicated to rollback protection.

The presented design has been implemented for an auxiliary service controller in the mobile phone — an energy management chip (EMC) — whose main task is to control battery charging and other processes related to powering the internal components of the mobile phone. Due to its main purpose the EMC does contain circuitry to produce high enough voltage levels to drive flash memory writing — a feature typically missing from the main CPU. The EMC controller in question also has embedded flash dedicated for configuration data related to battery charging. The design in **P2** is designed as add-on logic for the EMC controller chip to leverage "leftover flash cells" for TEE rollback protection.

The following constraints guided our specific design: The EMC had no source of randomness. Complexity of EMC side processing had to be be minimized to fit — in particular we minimized the number of registers and used only one cryptographic algorithm. Overall we follow a "leave it to the TEE" principle — whatever logic that can be outsourced to the TEE is moved there. E.g., a significant part of the burden of ensuring state correctness and consistency is left to the TEE.

The overall design is depicted in Figure 9.5. The trusted applications provide services accessible to OS-side applications and system functions. The TEE has access to device-specific secret keys using which it can construct encrypted storage containers for persistent storage maintained by the OS. To provide roll-back protection for the containers, the TEE also communicates with an external, non-volatile memory, that mainly provides non-volatile storage for rollback protection data binding, like counters. One example is to implement the MTM rollback counters using this mechanism. The main contribution of publication **P2** is the protocol and functional interface between the TEE and the logic that persistently stores the rollback protection data.

In **P2** we also note that the protocol can support confidential communication between the TEE and the external memory. This is a feature whose motivation was left out from the publication for size reasons. For straightforward integrity binding, and considering the minimization target, providing confidentiality may seem a strange requirement when communicating with a memory that typically would only contain an integrity-protected counter.

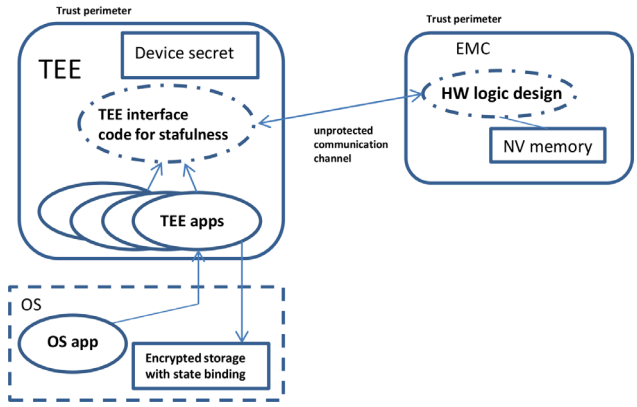

Figure 9.5: Rollback protection architecture

The reason for protocol confidentiality [3] is the probabilistic update of the non-volatile rollback storage. We want to minimize intentional attacks against storage wear-out by the user stimulating a huge number of events that cause rollback protection counters to increment until the flash cells that host them are worn out. For this, we make the unprotected communication channel between the TEE and the rollback protection memory confidential by cryptographic means. The intuition is that even if some state updates are logged only in volatile storage inside the TEE and not in the rollback protection memory, we can build the system to force the attacker to resort to probabilistic guessing about which state updates actually are logged to the rollback protection memory. Since also the erasure of TEE state requires a device reboot, the cost of a single rollback guess is quite high and not easily automated.

In our implementation, the number of guaranteed successful writes to the non-volatile memory cells is fairly limited, as low as 1000. Even with tens of memory cells available, the number of rollback-protected state transitions on a device is limited. In case the system is used as rollback protection for services with relatively low criticality, like implementing DRM music playback limitations or keeping track of wrong password entry attempts, then the available state space may quickly become exhausted. Worse, an attacker, like the user, may easily leverage such services to intentionally exhaust the rollback space by causing excessive amounts of state changes.

During system uptime, the TEE can and must ascertain the integrity of its local RAM memory also against rollbacks. Naturally, the decision whether to write to non-volatile memory or not must be probabilistic so that the attacker cannot guess which state updates remain external to the rollback protection memory, and thus remain subject to a state rollback opportunity. One simple setup could assume a requirement of a T year lifetime for the system, and N flash cells available for rollback protection, each with a minimum of C updates. If t updates occur in a day, the formula $y(t) = (86400/t)$ gives the average interval between updates.

Based on $y(t)$, a memory parameter (floating average) for the frequency of

[3] In the interest of testing, the final design variants of the rollback protection design re-introduced plaintext reads for all memory cells. For highest impact of the probabilistic update that feature must be disabled.

updates can be constructed whenever a successful update is performed:

$$m0 = y(t); m = (p-1)/p * m + 1/p * y(t)$$

where the ratio $1 : (p-1)$ in the above can be defined depending on an update limit defined by T and N:
$p = (N * C)/(T * 365)$
In such a setting, the decision whether to perform a state update in local memory by increasing a counter only in the TEE, or by updating the external memory can now be based on the following rules:

1. During system shutdown: If non-committed state updates in TEE local memory is bigger than a fixed value, then make an external update before shutting down.

2. When a state update is requested by an application, the decision to make it external can be done in the following way: A random value $r \in [0, 1]$ is compared to a linear (and suitably scaled) combination of m, p and other parameters, say a perceived "accumulated seriousness" of the state updates not yet committed to rollback protection memory. If $r < j$ then make the update in external memory, otherwise do only a local update.

To deter an attacker from determine the update state by eavesdropping the channel between the TEE and the rollback protection memory, the following approaches can be used:

1. All cell reads shall be encrypted, so that memory cell values are not exposed.

2. One flash cell is dedicated for 'dummy external writes' when the state is actually only updated locally. This cell will age prematurely, but writing to it will limit side-channel information leakage. Flash memory cell writing consumes significant energy, which makes the measurement of EMC chip power consumption a valid side channel for determining whether writes do occur or not.

3. All cell writes shall be encrypted, so that the memory cell index that is written to, i.e., the 'dummy write cell', is not exposed to an eavesdropper.

A probabilistic protection approach is by nature always attackable by probabilistic attacks tweaked to the protection mechanisms, and it is difficult to quantify in exact terms what is the gain that can be achieved with the mechanism provided above. On the other hand, not considering the problem of an attacker exhausting system resources dedicated to rollback protection also easily leads to insecure systems, or systems that are open to severe denial-of-service attacks, that in worst-case scenarios render the devices unusable.

9.4 Public Transport Ticketing Application

When a secure execution concept has been rendered functional on top of a PSE, along the lines presented in publications **P1-P7**, the architecture of

course ideally supports the development of any secure algorithms and code, for any use case. Publication **P8** and manuscript[91] describe one real-world example of an architecture implemented with the ObC architecture. The application is public transport ticketing with mobile phones and NFC. In short, we built a non-gated ticketing system for mobile phones with a built-in TEE and NFC communication primitives. The following properties defines the goals and target environment for our non-gated protocol:

R1. The location, time, traveler identity and needed cryptographic evidence shall form a tuple that defines the trip endpoints and traveler in a reliable and non-repudiable manner,

R2. Trip endpoints, e.g., touch-points at bus stops, could be equipped with contactless smart cards, but not with gates or contactless devices that require continuous power supply.

R3. The mobile phone should not be assumed to be connected to a back-end cloud infrastructure in real-time, i.e., the system must be designed to operate in a *partially offline* manner.

R4. The traveler activity with a touch-point should be modeled as 'tap', i.e., a traveler taps at a bus-stop touch-point before he travels, and taps another touch-point when he ends his trip at another bus stop.

R5. Travelers might be subjected to random ticket inspection, i.e., protocols must be designed to support this property.

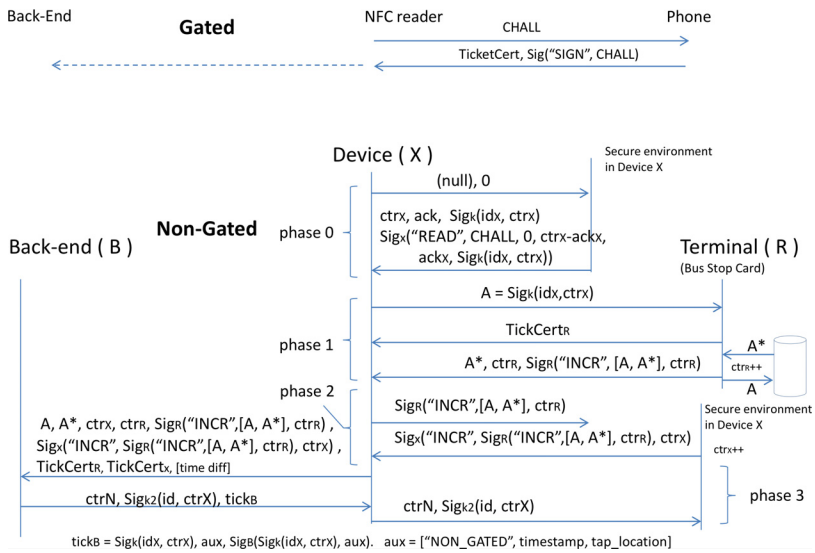

Figure 9.6: TEE operations (from **P8**)

Our solution is based on signed challenges produced by the TEEs, where each signature also included a monotonically increasing TEE counter. An overview of the tap protocol is provided in Figure 9.6. The same TEE logic is used both in the mobile phone TEE and in the contactless smart card at

touch-points. The evidence of a complete non-gated tap consists of signatures of both phases 1 and 2 in Figure 9.6 including the respective counters of the TEEs. This satisfies the identity binding based on the key stored in the TEE, and the location in the form of the touch-point identity. The time is collected from the fact that taps to especially touch-points are uniquely ordered, and based on the reporting time of the evidence by the phones in the system, an absolute time interval can be assigned to each counter value in each touch-point TEE. The evidence reported from the mobile phone of the traveler also includes the time of the taps collected using non-secure clock of mobile phone. Based on these reported tap-time by the individual traveler's phone (phase 0), the server maintains a reference time-interval which can be considered accurate.

Additionally, the phone TEE counter and its signature capability is limited by an authenticated release of the counter (phase 3), signed by the server cloud. This limits the amount of taps that a single mobile phone can perform before being forced to report evidence to the back-end cloud in order to continue tapping. Another feature of the system is that the challenges given to the touch-point smart cards by the mobile phones are commitments of the mobile phone(phase 0) from which the server cloud can at least statistically infer the mobile phone that sent the challenge. However, the final back-end reporting (phase 3) may be intentionally left out or be severely delayed. To improve on this situation, we store the challenges in the touch-point cards and probabilistically include it in and cryptographically bound it to two future responses of phase 1; i.e., every phone that taps a touch-point card is forced to relay two prior tap challenges back to the back-end cloud in addition to the response for its own challenge. This provides a back channel of taps towards the server that can be used for security auditing and even fare calculation.

Publication **P8** provides further discussion on on other system features, like enrolment, auditing, ticket verification as well as a protocol security analysis.

We believe publication **P8** describes a use case where the TEE features provide a highly useful security fundament for a real end-user service. However, since the ticketing work is motivated by the larger end-goal to enable mobile device ticketing for public transport, we have also worked on an extended design that enables ticketing on the deployed based of NFC-enabled mobile phones with no programmable TEE. Where the presence of a TEE makes partial off-line ticketing feasible, a trade-off between network access and end-user device security can be found for a system with secure touch-points. The rest of this subsection presents such a ticketing protocol improvement. It completes the discussion on ticketing and publication **P8**, but is not otherwise part of the main thesis topic.

We revisit the system assumptions of the original non-gated system in the following manner:

1. The user device / mobile phone is not trustworthy. A virus or the traveler himself potentially has access to all the code and secrets in the phone, and may report on these secrets over the Internet.

2. We increase the expectation for the capability of the phone to connect to a back-end cloud. We will design the revised protocol around a time period of t minutes. A traveler must connect his device to the back-end cloud and receive "real-time" tokens at most t minutes before traveling.

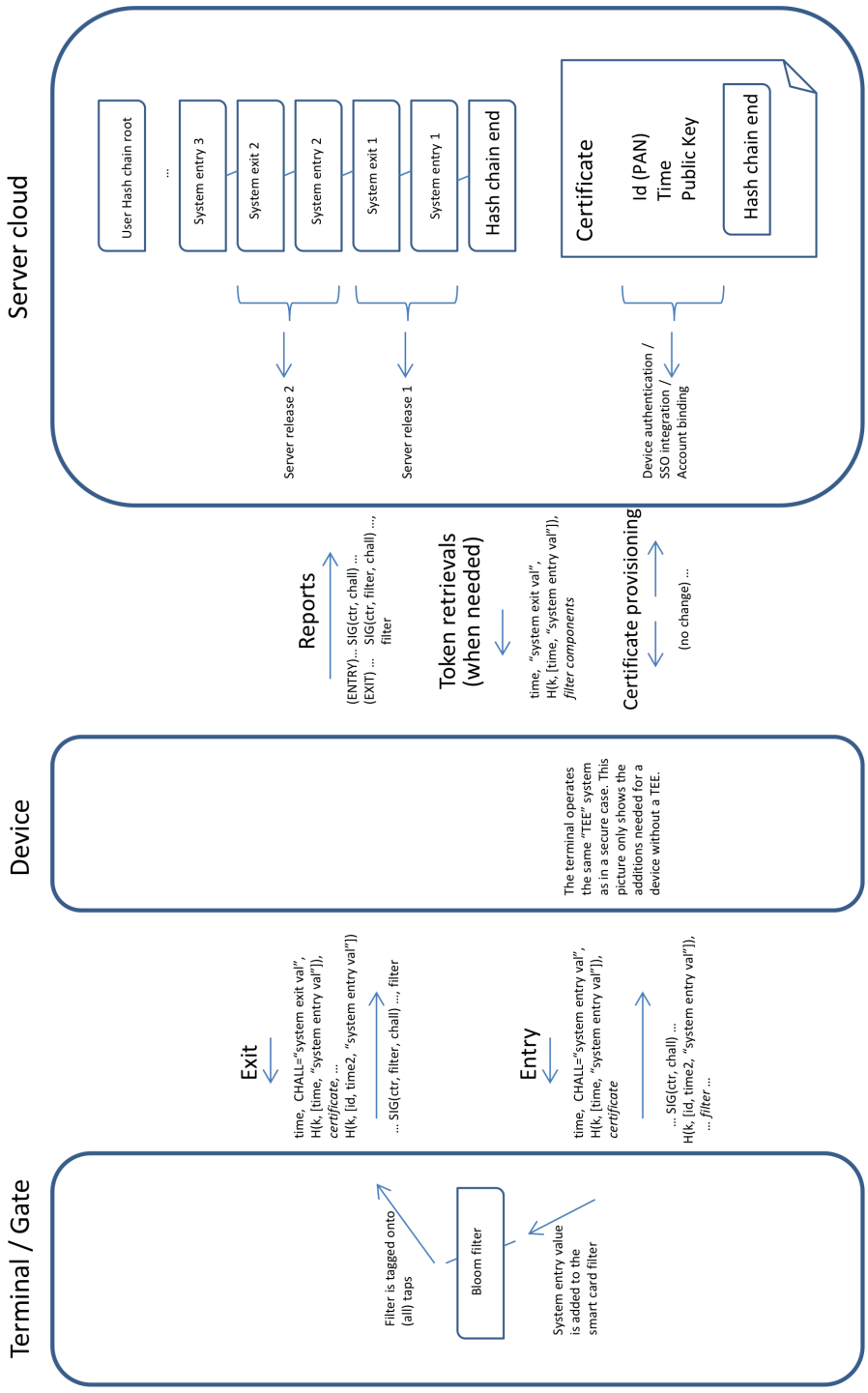

Figure 9.7: Ticketing - insecure terminals

Our main incentive for upgrading the ticketing system for open devices is to alleviate the risk of attacks potentially directed against the travelers with

open devices. Since our system is Id-based, the main threat is the misuse of identities, i.e., a liability concern for the travelers.

We assume that the main protocols and functions presented in publication **P8** still apply also to open devices. These devices will still perform the same steps of enrolment, certificate renewal, signing touch-point smart card responses and receiving authenticated release commitments for the device-specific counter. Compared to a device with a TEE, the trustworthiness of the open device is assumed to be weaker. Our only operation in the device that is partially directed against the traveler is the requirement for counter release commitments. We augment this functionality with a requirement for open devices to fetch the challenge for the touch-point smart cards in near-real time from the back-end cloud. In this manner, we still force the traveler's phone to periodically interact with a back-end server in the non-gated system. This new interaction can also be protected by validating user credentials, like a PIN, to further complicate the required system infiltration needed to mount a successful attack.

Furthermore, we add some new attributes to the transport certificates issued by the back-end infrastructure to open devices. We also augment the touch-point smart card logic with new auditing features that increase the possibility of catching identity theft in non-gated transport. For gated transport we add a feature to make tail-gating attacks[4] more difficult.

Figure 9.7 shows the overall additions done to the system. The new data structures are as follows:

1. A reverse hash-chain attribute is added to the transport certificate, signed by the server trust root and bound to an account of a traveler. The reverse hash-chain is split into run lengths of m elements ($m = 2$ in Figure 9.7). The actual elements (tokens) of the hash chain are retrieved m at a time by the mobile phone of the traveler before traveling. The token retrieval is possibly subject to user-authorization for improved end user protection.

2. A monotonically increasing time value is added to the system, and maintained by the back-end cloud. The time value is updated e.g., once a second, and is consistent across a single transport system. This time value will be signed by the server distributing the hash-chain elements and be cryptographically bound to the last token from the set of m tokens, i.e., the one that is to be spent first, on system entry.

3. All touch-point smart cards are augmented with a time-dependent Bloom filter[16] which is maintained individually by every single smart card. This is in addition to the statistically returned challenges of earlier travelers. Like the statistically selected earlier challenges, the Bloom filter is returned bound to the touch-point card signature. These forces the end-user device that taps the touch-point card to return the filter along with the challenge response to complete a valid transaction report. The time-dependence within each card is built based on the entry tap time commitments by the server, i.e., the reference time may be lagging for touch-point cards that are rarely used.

[4] A tailgating attack is where a customer intentionally throws a valid ticket back over the gate to let a friend defeat the physical access control of the gate.

The extensions for the ticketing system operate according to the message flows outlined in Figure 9.7. The touch-point smart cards now include distinct operations for entry vs. exit — intermediate taps, if supported, can be modeled according to the exit template.

The entry operation with the touch-point smart card includes the validation of the transport certificate, and that the entry token maps to the hash chain root. The entry operation will also validate that the time bound to the entry token is e.g., at most $t = 900$ (15 minutes) earlier than the last time seen by the smart card. If all validations succeed, the smart card will return to the end-user device not only the signed challenges but also a kind of verification ticket bound to the entry token value which can be validated by all other smart cards in the system. Furthermore, the entry token value will, in the card, be added to a Bloom filter that is periodically emptied, i.e., it contains only entry taps accumulated during a t-minute period. The Bloom filter is a very efficient data structure for this kind of aggregation, since filters for various smart cards can be trivially combined in the server, and the search for (the absence) of the double-spending of tokens can be made very efficient.

During exit, a touch-point smart card does not accept a tap operation without a matching system entry commitment returned by some other smart card in the transport system. An exit token must also be in the same hash chain as the entry token. These mechanisms alleviate identity theft, since an NFC eavesdropper may get the entry tap and the smart card signature, but not the exit token. Whenever tokens are retrieved with NFC eavesdropping or network-based attacks, the extra use of the token will trigger double-spending auditing mechanisms.

The traveler's incentive for reporting back evidence in the revised system is different from the protocols that use TEEs. In the latter case, the phone will force the traveler to report back on the threat of becoming dysfunctional, and all signatures are signed with keys that reside in the TEE. In the former case, the blocking mechanism relies on the conditional reception of the tokens from the server and the assumption that reporting of travel conducted based on those tokens must be performed before the retrieval of the next set of tokens. Token retrieval with no submitted evidence should by default be considered to represent the maximum fare of any travel that can be done on the system. In this way, the user is always incentivized to report the evidence correctly and promptly. Timely evidence feedback also benefits the user by improving the auditing mechanisms for catching double-spending.

Based on the Bloom filter contents, and the knowledge of tokens active at a given time (the only condition by which they are accepted at touch-point smart cards), every card returns, on every tap, a statistical representation of the recent entry taps at the touch-point card that is being tapped. This information is channeled by the mobile phones to the back-end cloud. With the assumption that 50-70% of phones report back (their own taps) almost immediately, it is easy enough in the back-end cloud to aggregate the Bloom filters and pinpoint double-spending occurring in the transport system - since all tokens are generated in the back-end cloud, full information of their contents and validity (in terms of t) is known to the back-end.

Brief security analysis of the added features

We assume that the phones, in addition to the augmentation proposed here, operate the default signing scheme already deployed. Thus a replay attack entails both stealing the longer term signature key from a phone, capturing the token over the air (and replaying it) or alternatively mounting a harvesting attack using a real-time virus in the attacked phone. We can identify the following threat categories and corresponding ways the described solution mitigates these issues:

1. The attacker has learned the long-term secrets of the victim. If the attacker copies the entry code off the air, he can likely in a non-gated environment produce a tap and a smart card response that will withstand at least cursory ticket inspection. However, the system will catch double-spending by aggregation and inspection of the touch-point card Bloom filters. In a gated system, entry duplicates are likely caught immediately and even access may be denied for either the attacker or the correct traveler. With token copies retrieved by eavesdropping the NFC interface, the attacker cannot exit a gated system if he does not follow the victim like a shadow.

2. Any copying of the short-lived tokens is valid only for entry during the stated system allowance period t. In a gated system this is an absolute measure, but old copies will also be caught at ticket verification in a non-gated system and by touch-point cards in case the use of the cards has advanced its notion of time past the time constraint of the token copy.

3. The attacker travels using a complete copy of all ticketing data in the original traveler's phone [5]. This means that the attacker will report all travels to the back-end just like the original traveler would do. Based on the protocol and its secrets, there is no way of differentiating the attacker from the original traveler since we assume that the attacker has full access to the mobile phone of the original traveler. However, double-spending mechanisms will notice parallel usage quickly, and in gated transport one of the two phones may even be denied system access or exit. In any case the fraud is quickly unearthed, and appropriate measures can be taken.

4. The attacker travels using the identity of a traveler, but does not report anything to the back-end if ticket verification is not encountered. In this case, the smart card filters will provide information to the auditing server about non-reported taps. Further, tap information becomes available as part of the back-channel from smart cards to the server through other tapping travelers. Using this mechanism, or by the attacker encountering ticket verification, the system will get information of an attacked identity.

5. A widespread software attack, where a vast number of phones are infected as a botnet and, for example, one trip from each victim is used

[5] All needed information is only available for copying at most t seconds before travelling because of the requirement to fetch fresh tokens before traveling, all needed information is only available after the tokens have been fetched.

by the attacker, will be impossible to protect against with the above assumptions. To alleviate this kind of attack some other reactive security mechanism, for example a virus checker, needs to be deployed.

The protocol additions for open devices puts in place several separate mechanisms to protect both the traveler and the system against undue fraud and misplaced liability. Nevertheless, when deployed in a mass-market scenario, there is a clear threat that widespread attacks can cause significant disturbances in the perception of the ticketing system by travelers, since an attacker can easily cause denial-of-service and cases where many kinds of "plausible undeniability" may surface. Clearly, a system where the traveler's phone is equipped with a TEE is the more user-friendly choice[6].

9.5 The author's role and real-world impact

The architectures presented in Chapter 9 have had quite varying practical impact. Today, the TEE components of the On-board Credentials architecture is present on more than 100 million sold mobile phones, running on three different processor families. This includes all Nokia Symbian 3 and Nokia Lumia (Windows Phone 8) mobile phones in production.

The thesis author was the main architect and implementor of all ObC designs relating to the TEE code, including the internals of the embedded virtual byte-code interpreter, its APIs as well as its provisioning functions. Despite this large deployment base and its publication history, ObC has not yet found widespread third-party usage. It has been used as technology fundament for several important in-house tests and trials on Nokia devices. In 2008 a hardware one-time password generator - the RSA SecurId - was ported to ObC. In 2012, the large-scale ticketing trial in New York with end customers travelling on the long Island Rail network[69] was implemented with a system that was based on ObC and publication **P8**. We hope that recent efforts on transferring selected lessons from this architecture into standardization (Chapter 10) can build an evolution path from the proprietary solution ObC currently is to an open solution that is adoptable across many manufacturers' devices, thereby stimulating third-party takeup.

The specification of MTMv1 was mostly concluded in 2010, although the work group continues to advance the technology, e.g. to incorporate in future specification versions the use-case of allowing third-party programs to be provisioned and run in a TEE with MTM as an authorizing component[68]. The author served as an in-company technical advisor for the MPWG during the MTMv1 specification, and has been participating as an active member in TCG/MPWG as specification editor since 2010, concentrating on future developments of the standard. The author also supervised and participated in three MTMv1 development efforts — a public MTM emulator[49] in 2008, an MTM as a TEE reference implementation in an on-the-market device in 2009 (publication **P3**), and the development of an MTM subset for the Terminal Mode Consortium specification[66] in 2011. A recent NIST draft publication,

[6]Further refined content from Chapter 9.4 has in collaboration with Sandeep Tamrakar been accepted for publication as a conference paper (Tapping and Tripping with NFC, International Conference on Trust and Trustworthy Computing 2013)

"Guidelines on Hardware-Rooted Security in Mobile Devices"[1], takes inspiration from the Roots of Trust concepts present in TPM1.2, MTMv1 and publication **P1**, and describes a requirements architecture for Mobile device security based on these.

The Terminal Mode (now MirrorLink) protocol and architecture is the first external use case that will base its security needs on functionality provided by the MTM. This will provide an important stimulus for mobile device manufacturers to include MTMs in their TEE offering.

Chapter 10

Future work

Mobile device manufacturers have used PSEs for almost 10 years already, primarily for services related to regulation or business ecosystems. The PSEs have not been available for application services or third-party programming. This has been the emphasis of work discussed in this thesis. Going forward, PSEs do have the unique property that they are fully integrated with the core processing of the device. The execution context of trusted applications can be well isolated from the OS, but at the same time the trusted applications can be granted direct memory access to OS and application memory, as well as to all external hardware connected to the main processor. Also interrupt control, DMA and MMU configuration is typically well integrated with the PSE architecture. This allows very powerful security mechanisms to be implemented, e.g., for run-time integrity checking or secure accessory access. However, many practical obstacles lies in the way of harnessing this power. Such issues include multiplexing driver and device states if they are controlled both from the TEE and from the OS, the size of logic needed to drive a GUI, and interrupt or DMA sharing from the PSE serving both secure and non-secure environments. This context clearly merits more research — in terms of what, but also in terms of how we can use the available hardware architectures to secure our devices even better.

Despite the obstacles, there is already ongoing standardization work[42] that hints towards attempting to extend the use of code in PSEs to operate less like an isolated processing environment and more like a hypervisor that orchestrates a larger variety of security processes in a mobile device. Use case targets include trustworthy user I/O and connecting short range radio communication, like NFC, directly to the PSE without passing through the OS.

In parallel, an ongoing trend is to populate also mobile processors with an increasing number of processing cores. This will impact PSE logic. If all cores deploy their own PSE, there are immediate problems in how these PSEs and the TEE's constructed from them can interoperate and share information between the cores. On the other hand, if only a single PSE exists in the device, multi-core control becomes an issue.

Therefore, the solutions provided by this thesis at best only can serve as a first stepping stone in a much bigger evolution where TEEs are likely to develop towards scheduling engines for enabling security services and operat-

ing systems in mobile phones, much like how hypervisor technology is used in cloud servers. Still, the threat model for mobile devices in the hands of users remains quite different from that of code executed in data centers, mostly because mobile devices more readily fall into adversaries' physical control. Also, usability and manageability of mobile devices is different from PCs since they have smaller screen sizes and use more constrained input methods. Thus, in mobile devices the security solutions and PSEs in them must at least partially evolve in their own direction.

The emergence of security standards to leverage and unify the use of PSEs also serves as evidence of the ongoing interest in this area. NIST recently made public a draft guideline for hardware rooted security in mobile devices[70] that builds forward from the work on roots of trust done for MTM in TCG. Also, where security APIs for decades only focused on providing access to well defined cryptographic primitives and functions (PKCS#11[72], TPM1.2[94]), emerging standards like Global Platform TEE APIs[42][41] and future MTM versions[68] also now standardize the programming environment inside PSEs and APIs that focus on provisioning and using trusted applications. The smart card legacy (ISO 7816[47], GP Card Specification[40]) also provide an architecture for provisioning trusted applications, even though programmable and stand-alone cards by nature and ecosystem are quite different from PSEs.

Figure 10.1 summarizes currently active and existing standards in the wider context of this thesis and PSEs. A recent stack of specifications around the Global Platform TEE is a central reference and a possible convergence point for these activities in the future. The programmable smart card specifications (on the left) define an ecosystem in which trusted applications written in JavaCard can be securely provisioned to a smart card and used. The stack is dominated by standards belonging to Global Platform, but some of the fundamental smart card properties originate from the ISO 7816 set of standards. Important adopter standards include Open Mobile Alliance (OMA) and 3GPP that specify services (i.e., trusted applications) around the SIM / UICC subscriber smart card. In the financial sector, EMV defines smart card interfaces for the payment industry. The new GP TEE (middle) specifications to some degree inherit from the programmable smart card technology, but add to it and adapt these concepts to be implementable on the most common contemporary PSEs. In the TEE programming paradigm, the trusted applications are natively programmed and compiled for a trusted OS. Thus, the GP TEE defines C-language bindings to a standard support library that contains cryptography security primitives, an API as a driver interface from the untrusted OS, and an application management and provisioning interface for authorizing the programs for the TEE. The TCG specifications are mostly functional interfaces, provided to the OS and applications through the trusted software stack (TSS). The functions shall however be securely implemented, either as hardware or as a trusted applications, say in the Global Platform TEE framework.

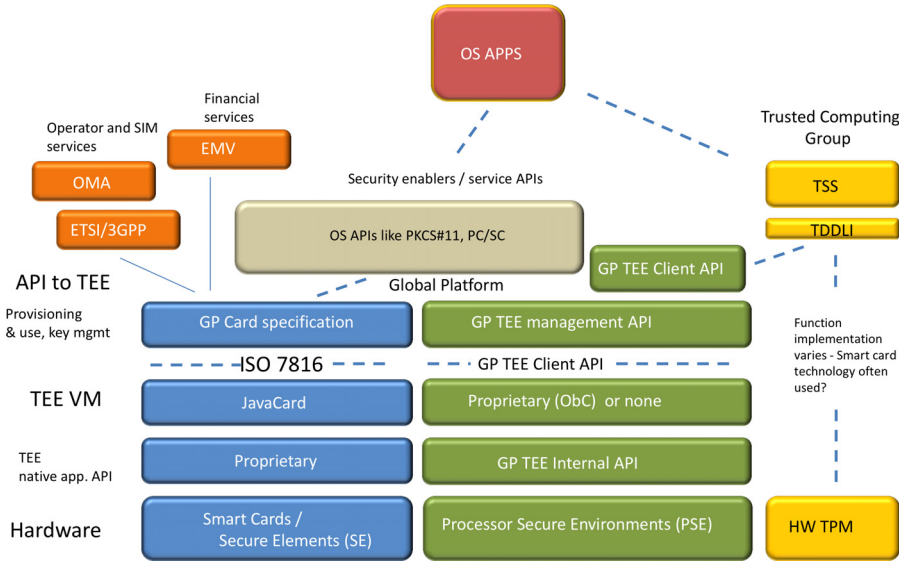

Figure 10.1: TEE-related standards

Chapter 11

Conclusions

Increasing the trustworthiness of the mobile phone is an important enabler when integrating services like wallets, access control tokens, remote banking or even theft protection into the devices that have an ever increasing presence in our daily lives.

This overall problem scope is wide. Operating systems need to be hardened. Remote provisioning of application code, firmware upgrades and restricted data needs to be reliably performed. Backup and restore of data, secure application data storage as well as firewalling between business and personal use are commonly needed features. Usability for the user as well as for application programmers is another important requirement. This thesis contributes towards solutions to all of these problems by providing a security framework within which many of these needs can be addressed. We set out to use contemporary processor hardware security features with the target to build TEEs that enable third-party use and especially third-party programmability. We define a software architecture, with some hardware aspects, that securely isolates the TEE execution from all other computation occurring on the device. The attached research publications as a whole provide evidence that by catering for off-line roll-back protection and by building external secure scheduling for an environment that is too resource-constrained to host a full-featured trusted operating system, we can still implement an architecture that despite the hardware limitations does provide the wanted programmability and isolation — in a very cost-effective manner.

To attain trustworthy execution in the given context, the thesis makes a contribution in somewhat multi-disciplinary manner. The discussion on abstract roots of trust in a phone is one that strongly relates to the field of trustworthy system research. Analyzing encryption primitives in a new data flow model is an example of cryptography research. The bulk of the thesis contributions clearly falls into the context of trusted computing, although some of the discussion on scheduling borders on the topic of system architectures. The application example also ventures into issues with security protocols.

All of the academic publications of this thesis were associated with significant engineering effort in the respective topical areas. We believe that this adds to the viability of the results. It is also humbling to know that the fruits of this labor have reached the hands of more than a hundred million end customers today and that this work feeds into standards that eventually

will define architectures available across device manufacturers and device categories. When the application programmer eventually can leverage trusted execution, secure storage and trustworthy device authentication to make his service endpoint more secure, then we will have nudged the state of the art in mobile device trustworthiness a little bit in the right direction.

Abbreviations

3GPP	Third Generation Partnership Project
AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AMD	Advanced Micro Devices
APDU	Application Protocol Data Unit
API	Application Programming Interface
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
BIOS	Basic Input Output System
BT	Bluetooth
CA	Certification Authority
CPU	Central Processing Unit
CryptoAPI	Cryptographic Application Programming Interface
DMA	Direct Memory Access
DRM	Digital Rights Management
DRTM	Dynamic Root of Trust for Measurement
eFUSE	on-chip, non-volatile, one-time programmable memory
eSE	Embedded Secure Element
EK	Endorsement Key
EMV	Europay, MasterCard, Visa
ETSI	European telecommunications Standards Institute
FIFO	First In - First Out (buffer)
GSM	Global System for Mobile Communications

HTTP	Hyper Text Transfer Protocol
HSM	Hardware Security Module
ICC	Integrated Circuit(s) Cards
IEC	International Electrotechnical Commission
IFD	InterFace Device
IMA	Integrity Measurement Architectyre
ISO	International Standardization Organisation
IRQ	Interrupt Request
JTAG	Joint Test Action Group (debugging interface)
LPC	Low Pin Count (bus)
MMU	Memory Management Unit
MMC	Multi Media Card
MPU	Memory Protection Unit
MTM	Mobile Trusted Module
MUSCLE	Movement for the Use of Smart Cards in Linux Environment
NFC	Near Field Communication
NIST	National Institute of Standards and Technology
ObC	On-board Credentials
OMAC	One key Message Authentication Code
OMA	Open Mobile Alliance
OS	Operating System
OTP	One Time Pad
PC	Personal Computers
PCMCIA	Personal Computer Memory Card International Association
PC/SC	Personal Computer / Smart Cards
PIN	Personal Identification Number
PKCS	Public Key Cryptography Standards
PKI	Public Key Infrastructure
POP	Post Office Protocol
POS	Point Of Sale

PSE	Processor Secure Environment
PUF	Physically Unclonable Function
RAM	Random Access Memory
RIM	Reference Integrity Metric
RNG	Random Number Generator
ROM	Read Only memory
RoT	Root Of Trust
RSA	Rivest, Shamir and Adleman
RTM	Root of Trust for Measurement
SDRAM	Synchronous Dynamic Random Access Memory
SE	Secure Element
SIM	Subscriber Identity Module
SoC	System-on-Chip
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCPA	Trusted Computing Platform Alliance
TEE	Trusted Execution Environment
TLB	Table Lookup Buffer
TLS	Transport Layer Security
TPM	Trusted Platform Module
TEE	Trusted Execution Environment
UI	User Interface
UMTS	Universal Mobile Telecommunications System
UICC	Universal Integrated Circuit Card
USB	Universal Serial Bus
VM	Virtual Machine
VT-x	(Intel) Virtualization processor architecture

Bibliography

[1] NIST SP 800-164. Guidelines on Hardware-Rooted Security in Mobile Devices, Draft. 2012.

[2] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings (paperback, orig 1996)*. Cambridge University Press, Nov 2005.

[3] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, May 2010.

[4] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:447–466, 2010. 10.1007/s10009-010-0145-y.

[5] Nadhem J AlFardan and Kenneth G Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. 2013.

[6] AMD. Secure Virtual Machine Architecture Reference Manual, 2005.

[7] Ross Anderson and Markus Kuhn. Tamper resistance: a cautionary note. In *Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[8] ARM. Technical Reference Manual: ARM 1176jzf-s (TrustZone-enabled processor). http://www.arm.com/pdfs/DDI0301D_arm1176jzfs_r0p2_trm.pdf.

[9] ARM. TrustZone API Specification 3.0, 2009. Technical Report PRD29-USGC-000089 3.1.

[10] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Drielsma, P. Hem, O. Kouchnarenko, J. Mantovani, S. Mdersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Vigan, and L. Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In Kousha Etessami and Sriram Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 135–165. Springer Berlin / Heidelberg, 2005. 10.1007/11513988_27.

[11] N. Asokan and Jan-Erik Ekberg. A Platform for OnBoard Credentials. In *Financial Cryptography and Data Security: 12th International Conference, FC 2008, Cozumel, Mexico, January 28-31, 2008. Revised Selected Papers*, pages 318–320, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 375–388, New York, NY, USA, 2011. ACM.

[13] J. Azema and Fayad G. M-Shield mobile security: Making wireless secure, 2008. Texas Instruments White paper.

[14] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor: A Successful Application of B in a Large Project. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM99 Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 712–712. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48119-2_22.

[15] Leonard J. Bell, D. Elliott ; La Padula. Secure Computer System: Unified Exposition and Multics Interpretation, 1976.

[16] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[17] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994. 10.1007/BF01185212.

[18] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[19] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[20] MultOS Consortium. MultOS home page and technical architecture, visited 2012.

[21] Victor Costan, Luis Sarmenta, Marten van Dijk, and Srinivas Devadas. The Trusted Execution Module: Commodity General-Purpose Trusted Computing. In Gilles Grimaud and Francois-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 133–148. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-85893-5-10.

[22] K Courtright, M Husain, and R Sridhar. LASE: Latency Aware Simple Encryption for Embedded Systems Security. *IJCSNS International Journal of Computer Science and Network Security*, 9(10), oct 2009.

[23] D. Davis. Secure BIOS, United States Patent 5,844,986, 1996.

[24] Gerhard de Koning Gans, Jaap-Henk Hoepman, and Flavio Garcia. A Practical Attack on the MIFARE Classic. In Gilles Grimaud and Franois-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-85893-5_20.

[25] Damien Deville, Antoine Galland, Gilles Grimaud, and Sbastien Jean. Smart Card Operating Systems: Past, Present and Future. In *In Proceedings of the 5 th NORDU/USENIX Conference*, 2003.

[26] Kurt Dietrich and Johannes Winter. Towards customizable, application specific mobile trusted modules. In *Proceedings of the fifth ACM workshop on Scalable trusted computing*, STC '10, pages 31–40, New York, NY, USA, 2010. ACM.

[27] DoD. Department of Defense Trusted Computer System Evaluation Criteria, DoD 5200.28 STD, 1985.

[28] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198 – 208, mar 1983.

[29] Thai Duong and Juliano Rizzo. Here Come The XOR Ninjas. 2011. Unpublished manuscript.

[30] Cynthia Dwork, Moni Naor, Guy Rothblum, and Vinod Vaikuntanathan. How Efficient Can Memory Checking Be? In Omer Reingold, editor, *Theory of Cryptography*, volume 5444 of *Lecture Notes in Computer Science*, pages 503–520. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-00457-5_30.

[31] J.G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S.W. Smith. Building the IBM 4758 secure coprocessor. *Computer*, 34(10):57 –66, oct 2001.

[32] Jan-Erik Ekberg, N. Asokan, Kari Kostiainen, and Aarne Rantala. OnBoard Credentials Platform: Design and implementation. Technical Report NRC-TR-2008-001, Nokia Research Center, August 2008. Available from http://research.nokia.com/files/NRCTR2008007.pdf.

[33] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby Lee, Nachiketh Potlapally, and Lionel Torres. Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines. In Marina Gavrilova, C. Tan, and Edward Moreno, editors, *Transactions on Computational Science IV*, volume 5430 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-01004-0_1.

[34] EMV. *Contactless Specification for Payment System*. Version 2.1, EMVCo, 2011.

[35] Paul England and Talha Tariq. Towards a Programmable TPM. In Liqun Chen, Chris Mitchell, and Andrew Martin, editors, *Trusted Computing*, volume 5471 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-00587-9-1.

[36] C.H. Fancher. In your pocket: smartcards. *Spectrum, IEEE*, 34(2):47 –53, feb 1997.

[37] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 295 – 306, feb. 2003.

[38] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 148–160, New York, NY, USA, 2002. ACM.

[39] Carl Gebhardt and Chris Dalton. LaLa: a late launch application. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, STC '09, pages 1–8, New York, NY, USA, 2009. ACM.

[40] GlobalPlatform Card Specification v2.2.1, 2011. http://www.globalplatform.org/specificationscard.asp.

[41] GlobalPlatform Device Technology. *TEE Client API Specification*. Global Platform, vrtsion 1.0 edition, July 2010. Report GPD_SPE_007.

[42] GlobalPlatform Device Technology. *TEE Internal API Specification*. Global Platform, vrtsion 0.27 edition, September 2011. Report GPD_SPE_010.

[43] David Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 1st edition, 2009.

[44] A Iqbal, N Sadeque, and I Mutia. An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems, 2011.

[45] ISO/IEC 18092:2004. *Information technology – Telecommunications and information exchange between systems – Near Field Communication – Interface and Protocol (NFCIP-1)*. First edition, ISO, Geneva, Switzerland, 2004.

[46] ISO/IEC 21481:2005. *Information technology – Telecommunications and information exchange between systems – Near Field Communication Interface and Protocol -2 (NFCIP-2)*. First edition, Geneva, 2005.

[47] ISO/IEC 7816-4:2005. *Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange*. Second edition, ISO, Geneva, Switzerland, 2005.

[48] Winter J. A hijackers guide to the LPC bus. In *8th European Workshop on Public Key Infrastructures, Services, and Applications EuroPKI '11*, 2011.

[49] M. Kylänpää J-E Ekberg. MTM emulator, now part of the TPM emulator at http://tpm-emulator.berlios.de/, 2007-.

[50] B. Kauer. OSLO: improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 16:1–16:9, 2007.

[51] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, June 2010.

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

[53] Ari Kokkonen. Development of trusted SW in HW supported security environment, Master of Science thesis, Tampere University of Technology, 4.6.2003.

[54] Kari Kostiainen, N. Asokan, and Alexandra Afanasyeva. Towards User-Friendly Credential Transfer on Open Credential Platforms. In Javier Lopez and Gene Tsudik, editors, *Applied Cryptography and Network Security*, volume 6715 of *Lecture Notes in Computer Science*, pages 395–412. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21554-4_23.

[55] C. Kothandaraman, S.K. Iyer, and S.S. Iyer. Electrically programmable fuse (eFUSE) using electromigration in silicides. *Electron Device Letters, IEEE*, 23(9):523 – 525, sep 2002.

[56] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973.

[57] Xavier Leroy. Bytecode verification on Java smart cards. *Software: Practice and Experience*, 32(4):319–340, 2002.

[58] Jochen Liedtke. Improving IPC by kernel design. *SIGOPS Oper. Syst. Rev.*, 27(5):175–188, December 1993.

[59] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[60] G. Madlmayr, J. Langer, C. Kantner, and J. Scharinger. NFC Devices: Security and Privacy. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 642 –647, march 2008.

[61] Bill McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., 2004.

[62] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, pages 143 –158, may 2010.

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

[64] RalphC. Merkle. A Certified Digital Signature. In Gilles Brassard, editor, *Advances in Cryptology CRYPTO 89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer New York, 1990.

[65] T.S. Messerges, E.A. Dabbish, and R.H. Sloan. Examining smart-card security under the threat of power analysis attacks. *Computers, IEEE Transactions on*, 51(5):541 –552, may 2002.

[66] CarConsortium home pages. http://www.terminalmode.org.

[67] Mobile Phone Work Group Mobile Trusted Module Specification v 1.0. http://www.trustedcomputinggroup.org/developers/mobile.

[68] Mobile Trusted Module 2.0 Use Cases 1.0, 4.3.2011. http://www.trustedcomputinggroup.org/developers/mobile.

[69] Newsday. LIRR tests smartphone payment system, 2012. http://www.newsday.com/long-island/transportation/lirr-tests-smartphone-payment-system-1.3580892.

[70] NIST. Special Publication 800-164: Guidelines on Hardware-Rooted Security in Mobile Devices (Draft) , 2012.

[71] Pontifical Catholic University of Rio de Janeiro in Brazil. The LUA langauge 2.4, http://www.lua.org, 1996.

[72] PKCS # 11: Cryptographic Token Interface Standard, 2004.

[73] Niels Provos. Encrypting virtual memory. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, pages 3–3, Berkeley, CA, USA, 2000. USENIX Association.

[74] Jean-Jacques Quisquater. The adolescence of smart cards. *Future Generation Computer Systems*, 13(1):3 – 7, 1997.

[75] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*, volume 3. Wiley, December 2003.

[76] J. Raszka, M. Advani, V. Tiwari, L. Varisco, N.D. Hacobian, A. Mittal, M. Han, A. Shirdel, and A. Shubat. Embedded flash memory for security applications in a 0.13 micrometer CMOS logic process. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, pages 46 – 512 Vol.1, feb. 2004.

[77] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[78] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278 – 1308, sept. 1975.

[79] Dries Schellekens, Pim Tuyls, and Bart Preneel. Embedded Trusted Computing with Authenticated Non-volatile Memory. In Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch, editors, *Trusted Computing - Challenges and Applications*, volume 4968 of *Lecture Notes in Computer Science*, pages 60–74. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-68979-9_5.

[80] J. Schmitz, J. Loew, J. Elwell, D. Ponomarev, and N. Abu-Ghazaleh. TPM-SIM: A framework for performance evaluation of Trusted Platform Modules. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 236 –241, june 2011.

[81] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Oper. Syst. Rev.*, 41:335–350, October 2007.

[82] Sean W Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831 – 860, 1999.

[83] Rob Spiger. Building hardware-based security with a Trusted Platform Module (TPM), 2011.

[84] Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[85] Geoffrey Strongin. Trusted computing using AMD "Pacifica" and "Presidio" secure virtual machine technology. *Inf. Secur. Tech. Rep.*, 10:120–132, January 2005.

[86] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[87] G. Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 9–14, New York, NY, USA, 2007. ACM.

[88] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. AEGIS: A single-chip secure processor. *Information Security Technical Report*, 10(2):63 – 73, 2005.

[89] G.E. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 339 – 350, dec. 2003.

[90] Harini Sundaresan. TI: OMAP plaform security features, whitepaper, SWPT008, july 2003, 2003.

[91] Sandeep Tamrakar and Jan-Erik Ekberg. Tapping and Tripping with NFC, 2012. Submitted for publication, Financial Crypto 2012.

[92] C Tarnovsky. Security failures in secure devices. *Black Hat DC Presentation*, page 74, 2008.

[93] The Trusted Computing Group (TCG). http://www.trustedcomputinggroup.org.

[94] Trusted Platform Module (TPM) Specifications. http://www.trustedcomputinggroup.org/specs/TPM.

[95] Zhenghong Wang and Ruby B. Lee. Covert and Side Channels Due to Processor Architecture. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 473–482, dec. 2006.

[96] M. V Wilkes. *The Cambridge CAP computer and its operating system (Operating and programming systems series)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1979.

[97] Xinwen Zhang, Onur Acicmez, and Jean-Pierre Seifert. A trusted mobile phone reference architecture via secure kernel. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, STC '07, pages 7–14, New York, NY, USA, 2007. ACM.

[98] W. Zhao, E. Belhaire, V. Javerliac, C. Chappert, and B. Dieny. A non-volatile flip-flop in magnetic FPGA chip. In *Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006. International Conference on*, pages 323 –326, sept. 2006.

[99] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, and Santosh Pande. Hardware assisted control flow obfuscation for embedded processors. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '04, pages 292–302, New York, NY, USA, 2004. ACM.

Processor hardware support for security is today found in contemporary cost-efficient mobile processors. This thesis builds on these hardware security features and presents software building blocks and architecture designs for implementing a Trusted Execution Environment (TEE) based on such hardware.

The main instantiation of this thesis work, the TEE part of the On-board Credentials (ObC) architecture, is today deployed in more than 100 million mobile phones around the world. This TEE design which to dominant parts was architected and implemented by the author has already been used in several public trials and demonstrations.

Aalto-DD 75/2013

ISBN 978-952-60-5149-9
ISBN 978-952-60-3632-8 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
Aalto University School of Science
Department of Computer Science and Engineering
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS