Cryptographic Enhancements for Trusted Computing Applications

<Kurt Dietrich>
`<Kurt>.<Dietrich>@<iaik.tugraz.at>`

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

<Doctor of Philosophy> Thesis

Supervisor: <Prof. Dr. Vincent Rijmen>
Assessor: <Dr. Liqun Chen>

<July>, <2012>

I hereby certify that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by reference to other authors.

Ich bestätige hiermit, diese Arbeit selbständig verfasst zu haben. Teile der Diplomarbeit, die auf Arbeiten anderer Autoren beruhen, sind durch Angabe der entsprechenden Referenz gekennzeichnet.

<Kurt Dietrich>

Acknowledgements

Like any other work, this thesis would not exist without the help of many other people besides the author. Vincent Rijmen deserves special thanks for taking time for supervising this work and suggesting improvements. Discussions with my colleagues Dieter Bratko and Harald Bratko their vast knowledge of public-key infrastructures. Special thanks also go to Johannes Winter and Franz Röck who I have been working together with on topics of Trusted Computing. Finally,I want to thank my wife Barbara for her support during the writing of this thesis.

Abstract

Nowadays, computers are used for many tasks of every-day life and the public is strongly dependent on these services. However, the dependency on these services bears threats, for example, to critical infrastructures but also to individuals. Consequently, securing these systems is mandatory.

Trusted Computing offers an alternative approach to system security. The concept of remote attestation that supports recording events that happened on a platform allows a detailed picture of the current state and the history of a platform. With this information, a reliable trust decision about the platform is possible. Nevertheless, improvements to this mechanism concerning performance and data security are proposed and analysed in this thesis.

Moreover, Trusted Computing has found its way to mobile platforms. However, the different security mechanisms provided by mobile devices allow different design options. Which options are possible and what requirements they have is discussed. Furthermore, concrete design proposals are given.

Anonymity protection is a major concern for Trusted Computing platforms. While anonymity protection on desktop platforms is a mandatory requirement, privacy protection on mobile hand-sets is still seen as optional by the TCG. However, especially these devices require such a protection as their manifold external communication features allow tracking of the devices and their users. Therefore, anonymity protection mechanisms for mobile platforms are investigated and new models based on different cryptographic primitives are introduced.

Keywords: Trusted Computing, Remote Attestation, TLS, DAA, mobile Trusted Computing

Kurzfassung

Computer werden heutzutage für viele Aspekte des täglichen Lebens verwendet und die Öffentlichkeit ist in hohem Ausmaß abhängig von deren Diensten. Diese Abhängigkeit birgt natürlich auch Gefahren, für kritische Infrastrukturen wie auch für einzelne Personen. Folglich ist es verpflichtend, die Sicherheit dieser Systeme zu gewährleisten. Trusted Computing bietet einen alternativen Zugang zur Systemsicherheit. Das Konzept von remote attestation das das Aufzeichnen von Vorgängen die auf einer Plattform stattgefunden hat unterstützt, ermöglicht ein detailiertes Bild des derzeitigen Zustands und des Verlaufs einer Plattform. Mit dieser Information ist es möglich, eine Entscheidung über den Sicherheitszustand einer Plattform zu treffen. Nichtsdestrotz werden Verbesserungen dieses Mechanismus betreffend Datensicherheit in dieser Dissertation vorgeschlagen und diskutiert. Darüberhinaus hat Trusted Computing seinen Weg auf mobile Plattformen gefunden. Aber die verschiedenen Sicherheitsmechanismen, die von den mobilen Geräten zur Verfügung gestellt werden, erlauben verschiedene Möglichkeiten des Designs. Es werden verschiedene Optionen und ihre Vorraussetzungen diskutiert und konkrete Vorschläge für Design angegeben. Die Bewahrung von Anonymität ist ein wichtiges Thema für Trusted Computing Plattformen. Während die Bewahrung der Anonymität für Computer zwingend vorgeschrieben ist, wird der Schutz der Privatsphäre auf moblilen Geräten von der TCG immer noch als optional angesehen. Aber gerade diese Geräte brauchen diesen Schutz, da ihre externe Kommunikation das tracking der Geräte und ihrer Besitzer ermöglicht. Deswegen werden hier Anonymitäts bewahrende Mechanismen fr mobile Plattformen untersucht und neue Modelle, basierend auf verschiedenen kryptographischen Grundlagen, vorgestellt.

Stichwörter: Trusted Computing, Remote Attestation, TLS, DAA, mobile Trusted Computing

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, a world without computers is un-imaginable. They are used for many tasks of every-day life. Some of their services are visible and directly usable by the common user like e-banking services and mobile applications. Others like traffic control systems or power-plant supervision facilities are doing their tasks transparently to the public. They are present in our personal households and even in our clothing. Consequently, public is strongly dependent on these services.

However, the dependency on these services bears major threats - cyber crimes are imminent and incidents which menace individuals as well as groups of people are reported on a frequent basis. The blackout of critical infrastructures like power plants not only costs a large amount of money, in case of nuclear power plants, for example, people's lives can be at stake. The same is true for traffic control systems and services that are provided via Internet. Large and small companies are depending on the availability of these services, a drop out is adherent with the loss of money. There are countless examples of breakdowns of IT services causing damage and loss. One thing that these IT services have in common is that they are attractive targets for cyber criminals.

As a consequence, security enhancing technologies to protect and to prevent criminals from disabling these services are required. One of these technologies is Trusted Computing which is one of the most important security technologies emerging in this decade. Many different platforms, ranging from server and desktop systems to embedded devices like mobile phones and automotive systems are taking advantage of its security features.

Trusted Computing offers an alternative approach to system security. The concept of recording events that happened on a platform allows a detailed picture of the current state and the history of a platform. With this information, a reliable trust decision is possible. However, not every computing platform is able to provide this information. In order to transform a platform into a *trusted platform* the integration of several security features is required. Trusted platforms are equipped with a set of security modules including security hardware, special boot software, and a customized BIOS. Only with support and interaction of these components according to the specifications of Trusted Computing, the platform is a trusted platform.

The core component of a trusted platform is the trusted platform module (TPM) [112]. It is a passive device meaning that it responds to requests like a smart card. In combination with a trusted software stack (TSS), the TPM is able to record events such as loading of software modules that occur on the platform. It is able to store this information in internal registers and assemble a cryptographic prove when requested. In addition, TPMs provide various cryptographic-key related operations which provide features to backup, create, delete or encrypt and decrypt keys. Signing operations are also possible. Therefore, different types of keys are supported which provide different security assumptions. Some keys may only be used inside a specific TPM were others may be transferred to another TPM.

Symmetric ciphers are not part of the TPM specification [112] and are therefore not mandatory for TPMs. Nevertheless, several use-cases exist where symmetric cryptography can increase the efficiency of TPMs. Some of them are discussed in the context of this thesis.

As an emerging technology, Trusted Computing is also an interesting area for cyber criminals as well as security researchers. Consequently, Trusted Computing mechanisms are under investigation. Most attacks are focused directly on the TPM, trying to circumvent its protection mechanisms like the attack discussed in [71]. For this reason, it is important that the emphasis of research is laid on improving the security mechanisms used in Trusted Computing.

Moreover, Trusted Computing is based on the principle that a platform behaves as *expected*. Expected in this context means that a platform in a specific configuration also has a specific behavior and that a remote platform can have trust that this platform does not deviate from this behavior. This goal is achieved by measuring and reporting of the exact configuration of a platform. However, this technique is often associated with the fear of loss of control over the own platform. Users believe that they are not able to install the software they want on a trusted platform so that the platform ends up with an untrusted configuration.

This might be true for some applications. For example, digital rights management that is often used for content protection for videos requires that the device is in a specific configuration. Otherwise, the device is not able to decrypt and show the requested video. This forces the user to have his device in a configuration that is defined by a third party.

This problem has been countered on desktop platforms by *virtualization*. Instead of providing one single platform with one specific configuration, a desktop system may support several virtualized platforms. One of these platforms may be used for personal applications, another one for work and a third one for viewing videos. With the porting of virtualization techniques to mobile platforms, a similar approach can be considered for hand sets.

Moreover, with the rich set of services provided by cell-phones new threats come into existence. The total traceability of individuals can be established by recoding their transactions they did and locations they visited. With this information specific profiles about people can be created. This overall traceability allows the intrusion into lives of individuals. While anonymity protection on desktop platforms is a major concern, privacy protection on mobile hand-sets is still seen as optional. However, especially these devices that we cannot imagine to live without today, require such a protection. Nevertheless, anonymity protection is for mobile devices is a major research goal in many of the ongoing research projects.

All the discussed issues in the previous paragraphs show that Trusted Computing - although it is an established technology - still offers room for improvements in various areas. Furthermore, the number of use-cases where Trusted Computing can be applied may be greatly improved by investigating these improvements. Consequently, the goal of this thesis is to identify and analyze such improvements and to provide discussions of their impact on further applications of Trusted Computing.

1.1 Use-cases for Trusted Computing

TPMs and trusted platforms may be employed in different usage scenarios. Therefore, the basic security mechanisms which also represent specific Trusted Computing use-cases are introduced in the following subsections:

1.1.1 Remote attestation

One of the most important use-cases of Trusted Computing is *Remote Attestation*. It allows platforms to get a trustworthy proof of the loaded software and current configuration of certain remote platforms, thereby enabling them to make decisions about the remote platforms' trust status. Common concepts like Internet Protocol security or Transport Layer Security make these decisions based on shared secrets or certificates issued by third parties. Unlike remote attestation, these concepts do not take the current configuration or currently loaded software of the platforms into account.

However, for establishing a trustworthy connection it is also important to have information about the status of the connecting platforms. Otherwise, malicious software on one of the platforms may threaten the security of the communication rendering the employed security mechanisms of the protected channel useless. Remote attestation cannot prevent a platform from being infected - it only allows a valid and trustworthy proof of the fact that malicious software has been loaded on a platform.

1.1.2 Sealing

The recording of configuration information allows another very interesting application of Trusted Computing - data *sealing*. With the sealing feature, it is possible to seal specific data - typically a cryptographic-key - to a certain TPM and to a certain configuration. The specific data is only released if the platform is in the configuration the data was originally sealed. In addition, the data is bound to a specific TPM, un-sealing of the data with another TPM is not possible.

1.1.3 Binding

A similar feature than sealing is data *binding*. In contrast to sealing, binding allows the encryption of data for a specific state of configuration in which the platform will be in the future. This means that a service may encrypt data which may be decrypted by a TPM when the platform has reached a specific pre-defined state. Compared to sealing, this data may now be used on multiple Trusted Computing platforms as it is not bound to a specific TPM.

1.1.4 Key storage

A very widely used feature, even on non trusted computing systems, is the key storage facility. TPMs do not store keys in their internal memory, they only have keys available when generating new keys or when operating with keys. Nevertheless, they provide the capability to encrypt asymmetric keys by other keys allowing them to be organized in a key hierarchy and to be stored on the trusted platform. The initial key is the *storage root key* which is stored inside the TPM and forms the basis for a key hierarchy.
Based on this basic use-cases, high-level use scenarios can be constructed. For example,

restricted access to keys provided by the sealing feature allows the integration of Digital Rights Management (DRM) services on trusted platforms. Moreover, remote platforms may use the features for authentication when connecting to a back service, thereby owning a strong security mechanism which protects their keys.

1.1.5 Anonymity protection

One additional use-case stems from the capability of Trusted Computing platforms to protect their transactions in the sense of privacy. Support for privacy protection - especially for mobile platforms - is becoming more and more important. In order to protect transactions of trusted platforms the Trusted Computing Group (TCG) has defined different mechanisms for privacy protection.

1.1.6 Electronic voting

A promising area for anonymous transactions is e-voting as they provide the enabling building block for digital voting. TPMs may be used to protect electronic identities or may be used as local watch dog that allows the voting service to monitor the platform and to exclude manipulation attempts on the voting process.

In addition, the strong and secure anonymization techniques provided by TPMs allow privacy for e-voting processed which is an absolute necessity for public acceptance of e-voting.

1.1.7 Electronic payment

Electronic payment is on of the major concerns of the banking industry. Moreover, mobile handset vendors want to spread and position their security products in this area aiming at customers to use their mobiles for these services. However, banks and financial credit services will only support payment services from devices that offer strong security mechanisms. This is especially true for mobile phones.

Nevertheless, privacy is also of major concern for this use-case as transaction from mobile devices can be traced and the anonymity of the individual user can be compromised. In the following paragraphs, some of the most important use-cases of mobile Trusted Computing and use-cases where privacy protection is mandatory are discussed in order to substantiate the relevance of this work:

1.2 Use-cases for Mobile Trusted Computing

The previously discussed use-cases are also valid for mobile trusted computing. Nevertheless, additional use-cases have been created to take the specific proprieties of mobile devices into account. In comparison to a desktop system that is solely controlled by the owner, a mobile hand set has more than one stake holder. Beside the owner of the mobile, the network provider is interested to have access to the device to do modifications of the configuration of the platform. In some cases even the original equipment manufacture (OEM) might have interest to access the device in order to execute maintenance tasks for the mobile's firmware.

Documents and reports published by the Trusted Computing Group provide an overview of the use-cases for mobile trusted computing [50], [51], however, many more exist.

The most relevant ones that are also important for this thesis are discussed in the following subsections:

1.2.1 Mobile ticketing

The possibility of using electronic tickets brings major advantages to both travelers and providers of public transportation systems. Tickets can be bought on-the go without being restrained to ticket kiosks and ticket counters. Accounting of tickets and their validation is much easier and cost effective because it can be done automatically by gateways and via the phone bill. However, security for both parties i.e. the customer who buys a ticket and the transportation service provider has to be guaranteed. On the one hand the provider wants to prevent to be cheated by customers and has to be sure that issued tickets are used on in a single device for one single purpose and that they are invalidated after being used . On the other hand, the traveler wants to be sure that he can use the bought service without problems and that he has definite evidence of being in possession of a valid ticket he has payed for.

While such services can be protected by TPM functionality, new threats like tracing of travelers is not addressed at the moment.

1.2.2 Anonymous authentication

One of the emerging use-cases for mobile transactions is authentication. With the improvements of their capabilities cell phones may now act as security tokens that, for example, allow access to certain services or locations.

However, this comes at a price. The location of a phone and its user can easily be determined. Applications like Google maps location allow to disseminate the user's location to the public. Every transaction done via the near-field-communication (NFC) interface also allows to pinpoint and track the mobile's position at a certain time.

Fortunately, there are technologies to counter these threats. Trusted Computing provides the feature of anonymous signatures for which one of the obvious use-cases is undoubtedly *anonymous authentication*.

However, detailed investigations of the feasibility and efficiency of anonymization technologies are missing. Consequently, use-cases dealing with anonymity protection on mobiles are also not investigated in great detail. In combination with new technologies like NFC, anonymous authentication for access control and location privacy should be mandatory. Extensions to passport functionality is one of the current research areas. Anonymous authentication and anonymous identities are a perfect addition in order to protect the passport owner's privacy.

1.2.3 Remote-sensing

Users personal devices are increasingly equipped with more and more features. These features include, for example, meters to measure the gravitational acceleration of the device, the environmental lightning and they come with GPS modules which allow to determine the exact position of the device and it's user. However, the devices may also be equipped with additional environmental sensors that can be used for remote sensing. A concrete example for remote sensing is the surveillance of *air quality*. Instead of a small set of surveillance units that reside on a fixed location, mobile phones can be used to get measurements on different locations at different times of the day. The handsets can be used to

form a kind of distributed network where each node (i.e. handset) in this network collects information about the environment. In this scenario, the hand-sets are used to measure the air-quality and report this information back to a back-end service. The information sent back includes the measurement value as well as time and location when the measurement was done.
It is obvious that this approach includes confidential and privacy threatening information. For analysis of the collected data, it is not necessary to know the actual device and its owner. It is sufficient that the entity processing the information knows that the information stems from a trusted source and that it is authentic.
To achieve authenticity and privacy protection at the same time, anonymous signatures, as used in the direct anonymous attestation (DAA) concept, are an ideal choice. A mobile device equipped with a mobile TPM can produce such signatures However, w.r.t. to the current mobile TPM specification, it depends on the manufacturer's decision whether to support DAA or not.

The use-cases discussed in this section are just a small portion of possible the use-cases. Nevertheless, they show the versatility and multiple applicability of Trusted Computing.

1.3 Contributions

The contributions of this thesis are manifold. They cover the analysis of existing Trusted Computing functions as well as proposals for improvements of Trusted computing security features. In addition, new concepts are introduced, supporting the prevalence of Trusted Computing on embedded systems.

1.3.1 Improvements on remote attestation

In the first Section of this thesis, the remote attestation process of Trusted Computing enabled platforms is analyzed. Several tests were conducted in order to investigate the advantages and disadvantages of this concept and to identify possible improvement opportunities.

A typical usage scenario of remote attestation is the application in secure connections. In order to establish a secure connection, previously invented technologies just validated the authenticity of the communication partners, usually done via public-key certificates. However, these kind of authentication mechanism ignores the state - especially the loaded software modules - of the platform.

The security of well established secure channel technologies like transport layer security (TLS) or IP security (IPSec) can be significantly improved by emerging concepts like Trusted Computing. How secure channel technologies can be adapted to use trusted computing concepts is subject to current research. A major part of this research addresses the integration of TLS into the remote attestation protocol. Combining remote attestation and existing secure channel concepts can solve the long lasting problem of secure channels that have to rely on insecure endpoints. Although this gap can now be closed by Trusted Computing, one important problem remains unsolved: A platform's configuration changes every time new software is loaded. Consequently, a reliable and in-time method to provide a proof for this configuration change - especially on multi-process machines - is required to signal the platforms involved in the communication that a configuration change of the respectively other platform has taken place.

With knowledge of the configuration, a remote platform can decide whether to open a channel to another platform or not. In current approaches, the proof of the platform configuration is processed before a secure channel is established. Currently, no satisfying solution how the change of a platform's configuration can be securely and reliably reported to the remote platform whilst a channel is open, exists. In Chapter 2, a reliable method to provide a proof of a configuration change is proposed.

Many research projects are working to find an answer to the question how TPMs and Trusted Computing features could be integrated into secure channel technologies like TLS or IPSec. In all these approaches, attestation information of a platform's configuration is presented to the remote platform and a secure channel is only established if the platform accepts the attested configuration. However, this attestation information only contains the configuration changes (i.e. software loaded) that have been recorded so far. Once the secure channel is opened, it is hard to detect a change in the configuration of a platform by a remote party. Detecting these changes is important as a platform that passed the attestation process with a valid configuration could change into a configuration that is not accepted by the host. This change into an invalid state could be triggered by loading a virus or Trojan Horse that could do substantial damage to the platform before the connection can be closed.

The common technique to detect such changes is to periodically read the configuration from the platform configuration registers (PCRs) or to periodically perform a TPM_Quote operation [112]. However, executing a TPM_Quote command and validating the information is time consuming. Depending on the platform's policy, each time a Quote is requested, a new attestation identity key (AIK) has to be created. To worsen the situation, a newly created AIK also has to be newly certified, which involves requesting an AIK certificate from a privacy CA. For server platforms that have to handle hundred to thousands of connections at the same time, this procedure without a doubt leads to a great bottleneck and is, therefore, not reasonable on these platforms. The ongoing loading and un-loading of AIKs also involves asymmetric cryptography.

These steps have to be performed for every newly established secure connection. When considering the situation on a common platform where multiple processes are concurrently running and opening channels to remote platforms, the situation is even worse. Multiple concurrent TLS sessions are active on the same platform, and different AIKs might be employed for each session. Consequently, each session has to load its own AIK(s) into the TPM prior to performing a quote operation.

As a first step, the impact of the remote attestation mechanism proposed by the TCG on secure channel technology - in detail transport layer security - is investigated and improvements are proposed. While remote attestation closed the gap between a overall security concept that includes the end-points into and the security properties offered by TLS, it is still prone to the problem of *time-of-check*, *time-of-use* (ToCToU). Therefore, a cryptographically *secure* solution is sought that binds the information of a configuration change to the transporting protocol (i.e. TLS) by cryptographic means and which has the property of reporting it actively at the time when the change occurs.

The research results discussed in this thesis show that a simple reporting mechanism can be integrated into current Trusted Platform Modules and Transport Layer Security implementations with a few additional Trusted Platform Modules commands and a few extensions to the TLS protocol. This hypothesis is backed by experimental results and a proof of concept protoype implementation which confirms that it is possible to implement a proof mechanism with only a few additional TPM commands.

As a result of the investigation of the existing mechanism, the following hypothesis can be constructed:

Hypothesis 1.1 *A secure platform configuration report mechanism must provide a secure proof and reporting of the platform configuration change at the time when the change occurs.*

This hypothesis basically means that the end-points of a transaction that rely on the actual information transmitted have to know the condition under which the information has been processed. This information may be shared when a change occurs. However, it does not imply that the information must be shared between the platforms immediately, it rather means that the configuration history may be also in some way *bound* to the processed data so that the verifying platform can validate the configuration at a later point in time. Data that is exchanged between two platforms should be flagged with respect to the current and past configurations so that the respectively other platform is later always able to determine the trust status of the specific data.

Note that the configuration information in the platform configuration registers does not only include the current state of the platform. It also contains the entire configuration history. Therefore, a remote verifier is able to track the different configuration setting of the processing platform.

In combination with secure channel technology, this leads to the follow-up hypothesis:

Hypothesis 1.2 *A secure platform configuration change reporting mechanism requires a cryptographic binding between the channel transporting the information, the sending platform and the actual information that is transported in order to notify the remote platform about modifications of the configuration and the change of conditions under which the information has been processed.*

Remote-attestation, in its current specification does not provide such a binding. How such an information *labeling* can be achieved is discussed in Chapter 2.

Nevertheless, one can also derive from these hypothesis that if data is sent over multiple hubs, all the different configurations of these hubs have to be included in the marking of the data. This offers interesting areas of research for information flow security.

1.3.2 Mobile TPMs

As discussed in the previous Section, a major building block for Trusted Computing are TPMs. While it is exactly specified how TPMs for desktop systems have to look like and which features they have to provides, a concrete design specification for mobile TPMs is missing. The TCG-mobile phone working group (MPWG) is working on standardization and use-cases for mobile trusted computing.

1.3.3 Mobile TPM design

However, the MPWG standard does not prescribe the actual design and implementation of a mobile TPM. This is done on purpose as the manufacturers should not be dictated how to implement their actual TPMs. The specification rather suggests a set of functions or in terms of the TCG *engines* or roots-of-trust. How these engines and roots are realized depends on the security features provided by the handset.

This approach leads to points of criticism which are discussed in this thesis as well as new approaches for solving them are proposed. For example, the lax specification allows TPM vendors to implement their mobile TPMs according to their specific designs, however, this approach also bears the problem of incompatibilities of the implementations.

This leads to the following question about the security requirements that have to be provided by the security mechanisms in order to host TPM functionality.

1.3.4 Mobile TPM security requirements

No specification exists at the moment how to deal with the different levels of security that are provided by the different security features. For example, how can the level of security of a security IC be compared to that of a non-certified TrustZone implementation? So, first of all, is a mobile TPM based on a security IC more *secure* than a MTM based on TrustZone? If so, how can this claim be satisfied?
It is obvious that there is a difference in the level of security that can be assured. While secure elements are available with Common Criteria certification up to EAL 5+, TrustZone platforms currently undergo a security evaluation aiming at EAL 3+. A lower security certification does not mean that these devices must not be used for security relevant tasks. Depending on the use-case, a lower rating might be sufficient. For example, for the DRM use-case where primarily video content is decrypted a lower rating is sufficient while for netbanking applications a high level certification is mandatory.

Dynamic command loading

High-level security evaluations are time and cost effective. The larger and feature richer the device to evaluate is, the more complex is the evaluation processes. Therefore, high security products are small, typically in the size of smart-cards. This comes with the constraint of limited processing power and memory. At the same time, the number of applications of the cards increases. For example, secure elements which are basically smart-cards that are attached to mobile devices are used to host applications like the Google-Wallet [48] and MIFARE functionality. With the raise of mobile TPMs, additional functionality has to be hosted on the security device which stress the constraints of the device even more.

To counter this issue, dis-embedding functionality has become a common technique used in security products. For example, the MultiMIFARE emulation developed by NXP provides mechanisms to support MIFARE as well as DESFire functionality with the same device. This is achieved by unloading the specific functionality depending on the current requirements. The unloaded feature is stored on the mobile host and loaded into the security device when the requirements change.

This concept can also be applied on mobile TPMs. Current mobile TPM implementations are monolithic blocks of software that either rely on security ICs or TrustZone-like protected environments. However, they are hard to update and modify in contrast to the versatility of modern smart-phones and their applications. Mobile TPMs do not take the specific requirements of the target platforms into account. Moreover, they ignore the requirements of modern trusted- execution environments (TEE) which are not fixed to a single application, but rather share their resources among different stakeholders which eventually requires a complex management unit.

Hence, an *updating*, *loading* and *removing* of TPM commands and their counterparts in the software-stack is reasonable for efficient resource usage of the TEEs. In the common sense, TEEs are either realized as TrustZone based solutions consisting of a high-security operating system kernel that manages the life-cycle of the trusted applications inside the TrustZone or they are realized as Security ICs that provide a virtualized execution environment - such as JavaCards. Both designs have in common that they share the same API for their applications. This API is the global platform API specified by GlobalPlatform [45].

The question addressed in this thesis focuses on how a framework for TPMs that allows a secure customization of functionality according to specific use-cases may look like. What are the requirements of the security device and what are the requirements of the platform that hosts this specific TPM functionality.

1.3.5 Analysis of anonymity protection mechanisms for mobile devices

Another contribution of the thesis addresses anonymous authentication. Privacy and anonymity protection are major concerns in Trusted Computing. Therefore, novel concepts like Direct Anonymous Attestation (DAA) were introduced. Consequently, this section focuses on the practicability of existing schemes. As a first step, DAA and its integration in secure channel technology and further in existing security frameworks is discussed.

Furthermore, the achievable performance of this technology is of interest. The variety of different TPM vendors and with it the variety of the different TPMs produced show interesting deviations not only in the performance of the DAA computations but also in specification compliance.

However, DAA in the sense of the TCG is only required for desktop systems - for mobile systems the TCG declares DAA as optional. Nevertheless, many use-cases where anonymity protection for mobiles is required exist. The application scenarios go far beyond the applications intended by the TCG. Modern smart-phones are used for different kinds of transactions, for example mobile banking and mobile payment. With the extensions of their capabilities and new technologies like RFID, their connectivity increases as well.

However, these new features bear new threats to the users of such devices. For example, transactions executed via the NFC interface allow tracking of the device and its user. The use-cases discussed so far are just a small set of applications where anonymity protection is appreciated.

There are unanswered questions when bringing anonymity technologies like DAA to mobile handsets. For example, which performance can be achieved on such platforms? While current high-end devices are capable of high execution performance, low-end devices might not provide sufficient computational resources. Complicating the situation even more, nearly all devices are shipped with virtual machines. While J2ME, enabled devices are the most widespread devices, Android phones are becoming more an more popular. Devices where pure native implementations are possible are hard to get by. Therefore, implementations have to be done either entirely in Java or in Java and native by support of native interfaces. While the first ones have the advantage that applications can easily be installed over-the-air, the latter one have the possibility to take advantage of native execution.

Another question focuses on the protection mechanisms provided by these platforms. How can DAA implementations for mobile handsets or embedded systems in general be designed? As previously discussed, mobile TPMs may be designed in different ways. These different design options also offer different design options for DAA functionality. While secure elements and TrustZone enabled platforms basically allow two design options, namely computing the signature either entirely in the secure environment or splitting the computation between the secure environment and the host platform, a combination of both allows many more design options. The combination of both security mechanisms is a common approach in future mobile hand-sets.

But not only the signature creation is of interest. As modern phones may connect among each other or join up in ad-hoc groups it is very likely that a mobile may take over the role of a DAA issuer or DAA-signature verifier. These roles also include the check of validation information. Hence, feasibility of processing revocation information on mobile devices is investigated.

1.3.6 Alternative anonymity protection mechanisms for mobile devices

The final contribution deals with improvements of existing schemes and alternative approaches to anonymity protection.

The first section deals with the question how the performance of the DAA signature creation process can be improved. On mobile phones J2ME and Android are the dominant Java execution environments. Therefore, the question arises how this platform can be employed for secure and efficient computations. Moreover, how can the computations be improved? It is not always necessary to change entire implementations to gain performance improvements. In some cases it is sufficient to parametrize the used algorithms accordingly. Moreover, the performance gain from native coded execution is of interest. While most Java virtual machines (VMs) provide an interface to access the native platform it is not always the case that this native code execution is faster than the pure Java implementation. Modern VMs are equipped with compilers that compile the Java code into native code before execution. In addition, hardware based accelerators improve the performance of Java byte code. The Jazelle processor extension [82] developed by ARM offers direct byte code execution. Instead of interpreting the byte code by the VM, the byte code is sent to the CPU which translates the code via an internal translation table into ARM thumb code. This technique offers a profound execution performance gain. Although smart-phone application processors are manufactured by different vendors, nearly all available smart-phones are based on CPUs designed by ARM. Hence, all smart-phones offer Jazelle byte code acceleration.

In contrast, when using the native interface the data that should be processed has to be moved outside the environment of the VM and after processing returned to it which produces some overhead.

The investigation of alternative anonymity protecting technologies includes schemes based on ring-signatures. This kind of signatures allow the creation of signatures based on public available information. i.e. public-keys. The size of the signature can be variably defined by the signer, thereby determining the level of anonymity provided by the signature. However, how can TPMs be included in the creation process?

Finally, a ECC based anonymous authentication scheme is defined and analyzed. The scheme is customized according to criteria that stem from industry requirements and use-case specific requirements with focus on anonymous authentication. Most publications in this area focus solely on the fast creation of the signature value on the client device. However, a full authentication step also includes the computation and validation of revocation information. The revocation process, however, is very time consuming especially for large revocation lists.

The introduced scheme also includes a modified revocation process that is based on symmetric cryptography instead of asymmetric as in the original proposal. This mechanism allows much faster revocation checks than with asymmetric designs and offers additional means of data protection.

Chapter 2

Secure Platform Configuration Change Reporting

2.1 Introduction

In today's computerized society, secure channels are a very important technology as they allow electronic transactions to be performed in a secure way. In combination with modern concepts, like trusted computing, the security of such channels can be considerably improved. Many attempts and proposals have been published to answer the question how to integrate trusted platform modules (TPMs) into common secure channel technologies like transport layer security (TLS) or IP security (IPSec). In principle, these proposals describe how attestation information can be used to identify the previously and currently loaded software on a platform and report this information to a remote entity by presenting an *attestation* token of a platform's configuration. A secure channel to this platform is then only opened if the local platform accepts the attested configuration. Unfortunately, the attestation information just contains the configuration changes recorded so far. Once the secure channel is established, it is hard to detect a change in the configuration of the remote platform.

A change in its configuration or *event* is triggered, for example, when new software is loaded into the memory for execution. Therefore, one problem still remains: how can a remote peer be securely and reliably informed that the local platform has loaded new software and that the local configuration has changed - in the worst case, into a configuration that is not accepted by the remote platform?

Keep in mind that remote attestation does not actively report such changes. Changes are merely detected by continuously requesting new attestation information. Current approaches periodically read the configuration from the platform configuration registers (PCRs) or periodically perform a TPM_Quote operation [112] in order to get this information. How it is actually processed and how frequently it is retrieved depends on the implementation - the remote peer has little or no influence on these parameters. In any case, executing a TPM_Quote command is time consuming. Prior to executing this command, a signing key has to be loaded into the TPM. Loading this key into the TPM involves public-key cryptography and is a rather slow process. The situation is even worse if concurrent sessions are active on the same platform, as fresh attestation tokens are required for each session. Moreover, if anonymity protection is mandatory, even different signing keys are required for each session resulting in continuous loading and un-loading of keys into and from the TPM. For server platforms that have to handle hundreds to thousands of connections at the same time, this procedure - undoubtedly - leads to a great bottleneck and is, therefore, not recommended for these platforms.

Remote attestation has several other problems than just the handling of AIKs, for example, the problem of time-of-check time-of-use (TOCTOU). Remote attestation is based on a request/response protocol where the requester sends a request - typically, through a secure channel - to a remote platform in order to get a response containing a reliable proof of the configuration of the platform. It is obvious that this proof only provides information to the point in time where the proof was requested and processed - it does not provide any reliable information about the platform after this time. With respect to secure channels, the attestation information is only valid for information that has been received until the request was processed.

Moreover, there is no guarantee that the information is delivered on time. This means that a platform can load infested software shortly after a Quote request has been processed and the remote platform cannot be informed of this new configuration early enough. In the meantime, viruses or Trojans can operate and perform malicious tasks before the remote platform is able to initiate counter measures or simply close the channel. The trust status of the platform and with it the trust status of the data processed beyond that point cannot reliably determined until the next request is processed. Consequently, it is not desirable to use the TPM_Quote command for a periodical reporting of a platform's configuration.

Another problem results from the delay and blocking of the transmission channel every time the TPM_Quote command is executed. This blocking causes interrupts during transmission resulting in a decrease of the actual data throughput.

In order to address all these problems, a concept that allows a reliable and secure reporting of PCR changes is proposed in this chapter. Moreover, the concept is designed in a way that it can easily be integrated into existing trusted computing enhanced TLS implementations like the ones discussed in Subsection 2.1.1. In the proposed approach, modifications of the TLS message authentication code (MAC) computation [27] are applied - instead of computing the MAC in software on the platform's CPU, the concept favors computing the MAC within the TPM.

This procedure has three advantages: First, if the TPM recognizes a configuration change (when a PCR is extended) it can directly incorporate this change into the MAC calculation. The differently calculated MAC can thereafter be detected by the TLS implementation of the remote platform. The remote platform, therewith, has a reliable proof of the configuration change. Second, there is no requirement to change the existing TLS protocol. Structure and size of the TLS records and the calculated MACs remain unmodified. Third, as the MAC is computed inside the TPM we have a reliable proof of the current configuration (similar to the TPM_Quote proof).

The proposed concept includes some minor modifications to the TPM specification, i.e. a few additional commands and a key derivation function. These modifications are necessary because the common idea behind trusted computing does not include support for enhanced protocols like TLS per se. Experiments with the proof-of-concept (PoC) implementation show that the modifications can be easily integrated in common TPM implementations without major modifications or addition of new cryptographic algorithms. Moreover, the concept can be applied to support multiple concurrent TLS connections.

The organization of this chapter is as follows: First, related work and background information about remote attestation and the TLS protocols which are used as initial point for the proposed concept are discussed in Section 2.1.1. This Section is followed by a detailed discussion of the TCG's proposed remote attestation protocol in Section 2.2 where experimental results for estimating the efficiency of the existing TCG approach are gathered and analyzed. Based on these results, a new approach is proposed and investigated in Section 2.3 which additionally contains an explanation of the new model and a discussion of the proof-of-concept implementation and the related TPM modifications. Further extensions and modifications to the original proposal are discussed in the consecutive Section 2.4. The modified approach provides an even more sophisticated and secure approach to event reporting via TLS by deriving and using the MAC-key inside the TPM. The Section furthermore provides an analysis of the advantages and disadvantages of the modified and the original design and provides a comparison of both approaches. Another modification to the concept is proposed in Section 2.5. While the previous approaches were only able to detect changes, the additional modifications of the third approach allow the reporting of events through the TLS channel without the constraint of closing and re-initiating the connection. In addition, experimental results are discussed that show the efficiency advantage of the new approach in comparison to the TCG definition. Furthermore, a proof-of-concept prototype is discussion and how the design can be modified to support TLS client authentication via TPMs. Finally, Section 2.7 summarizes the contributions and concludes the chapter. In addition, impulses for further research on the topic of remote attestation are provided.

2.1.1 Related work

In this section, some of the recently proposed secure channel enhancements are reviewed. All enhancements are based on the integration of TPMs into these channels, therefore, these channels are addressed as *trusted channels* for the remainder of the document.

Goldman et al. discuss several methods of linking server endpoint validation and TCG's remote attestation [47]. They address relay attacks, where a compromised server might relay a remote attestation quote from a trusted server to a requesting platform. Furthermore, they introduce a platform property certificate that links attestation identity keys (AIKs) to platform endpoint properties. This approach focuses on virtualized environments and allows fast endpoint certificate revocation and creation with application dependent security properties. However, Goldman et al. do not address the problem of reliable configuration change reporting.

Another approach is proposed by the TCG. The trusted network connect (TNC) group of the TCG is working on reference architectures that focus on policy enforcement and authentication for granting network access. The architecture is rather generic and is based on collecting and verifying integrity information of the communication partners. Which integrity information is actually included is not discussed and is part of the policy, depending on the invoked platforms. Like Goldman et al, they do not consider configuration changes and reporting while trusted channels are open. Moreover, they do not address the problem of linking the channels to certain platforms and TPMs. As the specified framework focuses on policies, it doe not require a TPM to be involved in the connection process.

The most comprehensive approach to trusted channels is discussed in [43]. The authors propose an implementation that reliably determines the trustworthiness of the communication endpoints and they show how it can be combined with trusted computing technology.

In contrast to other proposals, they try to address the problem of changing configurations. In their approach, the session keys are stored in the trusted computing base, restricting access to them. This means that access to these keys is only granted if the platform is in a certain state. However, once the platform has reached the specific state and the key is released, use of this key is not affected by further changes of the configuration. They add an extra TLS message that notifies the remote peer about a configuration change. Furthermore, they define an additional TLS extension (*state_change_extension*) that carries the encrypted configuration information. This information is exchanged between the communication partners in case the configuration changes. However, they require extra messages and extra components for notification of configuration state changes through the secure channel. Moreover, they do not create this proof inside the TPM and, therefore, do not have an implicit proof of a change in the TLS protocol as in the proposed approach of this thesis.
In [97], the authors present a concept that binds a Diffie-Hellman key to a specific platform configuration. In their approach, they incorporate a public Diffie-Hellman key in the TPM_Quote as external data. However, their concept is susceptible to relay attacks as this approach is not resistant to man-in-the-middle attacks during the remote attestation process and the problem of configuration change reporting is not addressed.

Many research projects are committed to find an answer to the question how TPMs and Trusted Computing features could be integrated into secure channel technologies like transport layer security (TLS) or IP security (IPSec). One thing that these approaches have in common is that attestation information of a platform's configuration is presented to the remote platform and a secure channel is only established if the platform accepts the attested configuration. However, this attestation information only contains the configuration changes (i.e. software loaded) that have been recorded so far. Once the secure channel is opened, it is hard to detect a change in the configuration of a platform by a remote party. Detecting these changes is important as a platform that passed the attestation process with a valid configuration could change into a configuration that is not accepted by the host. This change into an invalid state could be triggered by loading a virus or a Trojan Horse that could do substantial damage to the platform before the connection can be closed. All proposals ignore the problem of a *reliable* configuration change reporting which means that although you may send event notification messages through a trusted channel no cryptographic proof (like the Quote provides) of their validity is provided.

2.1.2 Background

In this Section, the basics of the remote attestation and the TLS protocol are introduced. As the proposed concept relies on mechanisms provided by TLS, a brief overview of the cryptographic components of the protocol is provided. Moreover, the key derivation process is discussed as it plays a major role in the following discussions.

Although the newly proposed concept does not rely on TCG's remote attestation, the basics are explained in order to understand the differences between the TCG proposal and the concept proposed in this thesis. Moreover, the analysis of the attestation protocol in Section 2.2 requires basic knowledge of the remote attestation process which is covered by the following subsection.

Remote attestation

One of the most important security mechanisms of Trusted Computing is *remote attestation.* The basic idea of remote attestation is to provide reliable information about the current configuration of a trusted platform. The term *configuration* is not exactly defined, a configuration may contain, for example, information about loaded software, information about the configuration settings of specific software or the hardware components of a platform - virtually any kind of information or *event*[1] that influences the configuration and status of a computing platform and that can be *measured.* Measure, in the context of trusted platforms, is defined as the SHA-1 hash value of a specific event [112].

In typical remote attestation scenarios, information about the current software configuration (i.e. the loaded software modules including the BIOS software) and hardware configuration is used to generate the configuration information. During the load process of the software, which is also addressed as event for the rest of the document, the SHA-1 fingerprint[2] of the loaded software is created and written into special registers of the TPM.

Platform configuration registers

The TPM contains a specified number of such registers that hold configuration- or event information, the so-called *platform configuration registers* (PCRs). The hash is not simply written into the registers, but is rather *extended* which means that the current content of the register is concatenated to the newly added configuration information and then hashed into the PCR so that the new state (x) of the PCR is calculated as $PCR_x = H(PCR_{x-1} \| H(E))$, where H denotes a hash function and E the measured configuration/event item. This way, PCRs contain the fingerprint of the current configuration and all measured events of the platform so far.

During the execution of the remote attestation protocol, a requester may now verify the platform's configuration by retrieving the PCR content which is also called "Quote". To do so, the requester defines a set with the numbers of the PCRs of interest and, along with a *nonce*, sends it to the proving platform which forwards the request to the TPM. The TPM computes the hash of the content of the selected PCRs: $PCR\_HASH = H(PCR_a \| PCR_b \| ... \| PCR_n)$ and signs this value including the *nonce*, the pre-defined string "'QUOT"' and a predefined TPM version number. This structure is called a *Quote.* Moreover, in the context of this thesis, a *Quote* or quote operation will also denote the operation which instructs the TPM to generate the structure and the signature.

In addition, a database, the so-called *storage measurement log file* (SML) is created. Although its structure and content is not specified, it typically contains a list of all events on the platform e.g. the names of all load software modules, their hash values and a description of the corresponding event. In combination with the signed Quote, a verifier can now evaluate which software has been loaded and check the reported quote value by re-calculating a reference quote from the hash values stemming from the SML and comparing the result with the originally reported quote. More information on the Quote operation and a discussion of practical results with this technology can be found in [33].

In addition to the Quote operation, remote attestation relies on mechanisms that allow a platform to record the events and loaded software during the boot phase and up-time of the platform. During this phase, the single software images that are loaded into the

[1]Basically, every change of the configuration may be called event.

[2]In the new TPM specification this will change allowing TPM vendors to select different algorithms.

platform create a *chain of trust.* This chain is created by subsequent measurement of the single software images e.g. the BIOS measures the boot loader, the boot loader measures the operating system image and so on. The measured values are stored in the PCRs as discussed in the previous paragraphs. Trusted platforms may either employ a *secure boot* or an *authenticated* or *trusted boot.* While the first boot mechanism is typically used on embedded devices [34], the second mechanism is used on desktop and server platforms. The basic difference between both mechanisms is that in a secure boot scenario the boot process is aborted if the loaded and measured software does not correlate to a predefined value. This means that the loaded software must either match a pre-defined integrity value or that after extending a certain PCR, the content of this PCR must match a pre-defined value.

When using an authenticated boot, the events and the loaded software during boot are just recorded in the PCRs. The consequences resulting from the different approaches become effectively clear when recalling the measurement process of the loaded software. During secure boot, if a software image was modified, further execution of the image is immediately aborted. During authenticated boot, the boot process is continued and the measurement of the modified image is just recorded. As a result, malicious software infecting the platform can be noticed immediately in case of secure boot. In case of authenticated boot, such a break is only detected after validation of a remote attestation request.

Anonymity protection

Remote attestation also supports mechanisms for anonymity protection of the TPM and the platform. Basically, two approaches are defined by the TCG and a third one is proposed in the context of this thesis. The first and simplest approach is to use one-time signing keys. These specific keys are called *attestation identity keys* (AIKs). They may only be used for computing Quotes and should be used only once as repeated use may allow tracking of the transactions where these keys are involved. As a consequence, every Quote requires a new key which is generated by the TPM and certified by a *privacy certification authority* (PCA). The difference of a PCA to a common CA is that the PCA does not encode specific information about the owner of the key into the issued certificate. All certificates are uniform with exception of the public-key. Therewith, a verifier is not able to identify the actual owner of the key and it is not possible to link two different public-keys to a single platform.

The second approach uses local certification instead of remote certification avoiding the need for a permanent online CA. The basic idea of the direct anonymous attestation (DAA) protocol is to sign the AIK by the TPM with a group signature. Therefore, the TPM has to be registered by a specific group and receive the corresponding group credentials. When a new AIK is generated, the TPM signs or certifies the new key with a DAA signature. The key is then used to sign the Quote information as discussed in the previous paragraphs. With the Quote and the signed AIK, a verifier is then able to check the Quote and to verify the validity of the AIK. As the AIK is certified with a group signature, the verifier is not able to identify the specific signing platform. He can only see that the signer is part of a specific group. As there is no third party, like the PCA, involved for every single transactions, the protocol is called *direct.*

The TLS protocol

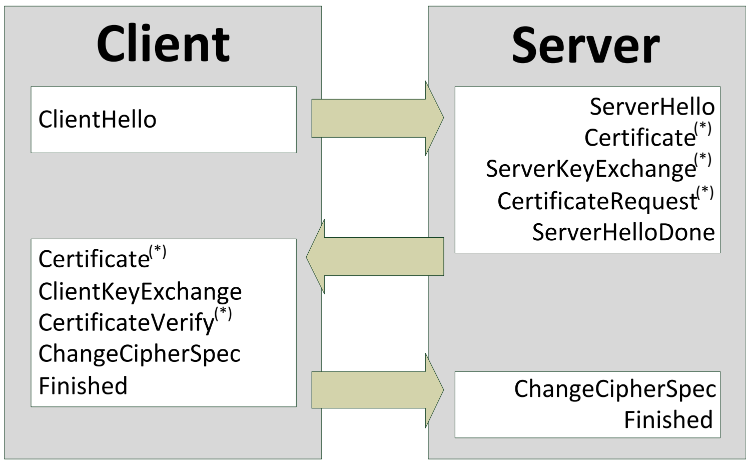
The transport layer security protocol is the most widely used protocol for securing connections in practice. It provides *confidentiality*, *authenticity* and *integrity* for protecting information that is transported between two entities. While the protocol was designed to satisfy these three security properties, it completely ignores the current state of the endpoints. The following quote demonstrates the common opinion on TLS:

> "TLS is like transporting a credit card between two homeless in an armored vehicle".

Consequently, integration of the state of the communication endpoints in the protocol flow is essential for the security of the entire communication process.
The basic design of the protocol is as follows: The protocol consists of two phases, the parameter negotiation phase and the actual data transmission phase. These phases are further separated into five protocols:

1. Handshake protocol - during execution of the handshake-protocol, the authentication- and session parameters are negotiated and exchanged between client and server. In the typical use-case, the server authenticates against the client. The negotiation parameters include random numbers from client and server. In case of non-anonymous key-agreement, the server sends its authentication information (i.e. a X.509 certificate) to the client. The client validates the certificate, generates a random number (pre-master secret) and encrypts it with the public-key of the server. This way, client and server get into the possession of a shared-secret. (Alternatively, Diffie-Hellman key-agreement may be used). The data flow in the handshake phase is depicted in Figure 2.1.

Figure 2.1 TLS handshake protocol



2. Change cipher spec. protocol - The change cipher spec protocol consists of only a single message which informs the receiver to switch to the negotiated session parameters.

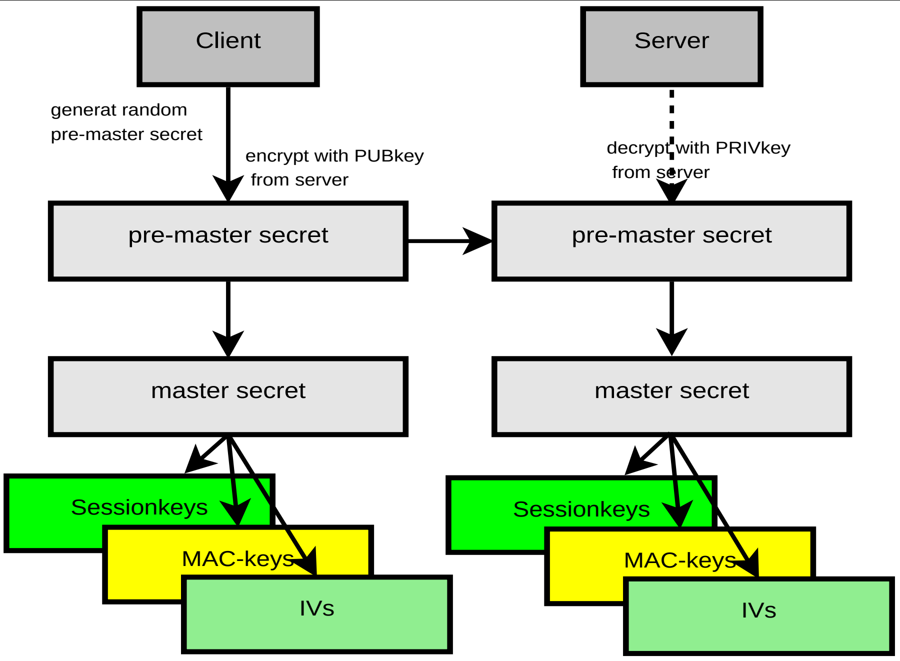
3. Alert protocol - the alert protocol supports various messages that are used to inform the opposite platform about state changes in the protocol. These changes are

triggered by errors (e.g. the authentication of record cannot be verified) or by the communicating platforms (e.g. notification to close the channel or when invalid authentication information was used). 24 different messages are specified. The alert message is constructed from: AlertLevel (1 byte: warning = 1, fatal = 2), AlertDescription (1 Byte: e.g. close_notify = 0, no_renegotiation = 100 etc.).

4. Record protocol - the record protocol is responsible for the actual data encryption with symmetric algorithms and the application of authentication information. The structure of a TLS record is basically as follows: 1 byte **content-type**, 1 byte **protocol-version**, 2 bytes **length of the data-block**. Both, server and client manage an outgoing and an incoming channel where they have to encrypt and decrypt TLS records as well as applying and checking integrity information.

After the handshake has finished, client and server can compute the *master-secret* from the pre-master secret which is then used to derive the actual cryptographic keys (see Figure 2.2).

Figure 2.2 Key Derivation in the TLS Protocol



In formulas, the master secret is computed as follows:

$$master\_secret = PRF_{pre\_master\_secret}(\text{"}mastersecret\text{"}\|random_{client}\|random_{server}) \quad (2.1)$$

where PRF is a pseudo random function based on the MD5 and SHA-1 algorithm. The actual symmetric-keys are derived via:

$$key\_block = PRF_{master\_secret}(\text{"}keyexpansion\text{"}\|random_{client}\|random_{server}) \quad (2.2)$$

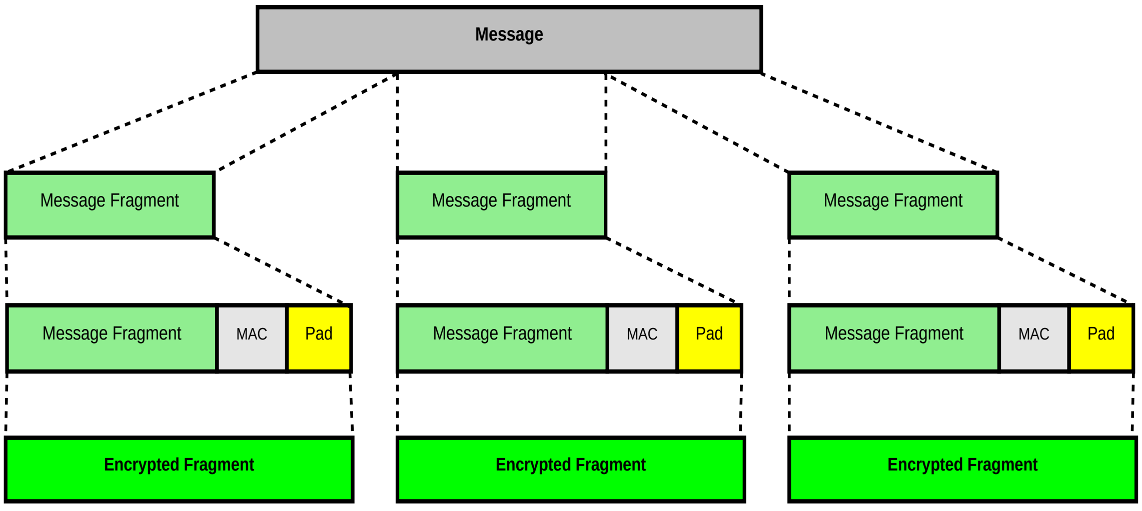
with

$$PRF = MD5(secret_{upper}, label + seed) \oplus SHA - 1(secret_{lower}, label + seed) \quad (2.3)$$

where *key_block* is split into $(MAC_{keys} \| encryptionkeys \| IVs)$. After the key derivation process is finished, client and server possess the same encryption- and integrity check-keys as well as - depending on the used cipher mode of operation - the same initial vectors (IVs).
The MAC keys are used during the data transport phase the authenticate and integrity protect single TLS data blocks (aka *TLS fragments*).

$$MAC = HMAC(MAC_{key}, MessageFragment \| sequence\_counter) \tag{2.4}$$

Figure 2.3 TLS Fragment Encryption



The integrity value (=MAC) is computed by computing the HMAC of the specific TLS fragment plus a sequence number [27]. In Figure 2.3, the basic steps for TLS record encryption are shown. The message that will be sent is split into different blocks. From these blocks, MAC is computed and attached to the message block. If necessary, the block is padded and the resulting block consisting of message, MAC and padding is encrypted.

2.2 Remote attestation and secure channels

In this section, the impact of the remote attestation protocol on the data throughput of TLS channels is analyzed. It is evident that a higher frequency of attestation requests allows a more detailed determination of the configuration and trust status of a platform. However, how does an increased number requests affect the data transport capacity of a TLS channel? In the following context, *data* denotes the part of transmitted information that represents the application data part - without the overhead produced by the TLS protocol messages and attestation messages.

For gathering practical measurement data, two measurement methods were defined - both methods rely on the assumption that an application, or trusted service, periodically requests fresh attestation information from the respective other platform.

In the first measurement method, attestation requests are sent on a regular basis, meaning that the requester sends a request to the attester every n seconds. In the second method, one request is sent after every m bytes of data that have been transported through the channel.

2.2.1 Assumptions

In order to analyze the actual decrease of the data throughput, an idealized model of the proposed approach and several assumptions are defined:

1. Prior to executing a quote operation, a signing-key has to be loaded into the TPM. This operation is very time consuming as the key has to be loaded and decrypted from external storage. In current TPMs, the key is RSA encrypted, hence, we assume that a valid key has already been loaded. Consequently, each quote operation uses the same, pre-loaded key.

2. We assume that only one PCR event is reported per request. As an arbitrary number of events may occur in the period within the requests, the reported data (i.e. configuration changes recorded in the SML) might grow to a large amount. However, this amount grows linear with the number of reported events. Consequently, the time required to transport the information also grows with the number of events.

3. As the focus lies on the cryptographic operations of remote attestation, the reported events are not validated. The validation does not involve a TPM, nevertheless, the exact method for evaluating them is not defined. The simplest way would be for the verifier to have a list of valid events (i.e. the hash of the event) and performs a look-up of the reported events in a list. A different method might involve sending the event records to a trusted third party for evaluation.

As a consequence, the measured results represent the *upper limit* of the actual data throughput possible. Moreover, TLS connections consist of an outgoing and incoming channel [86] allowing to send and receive data which may be used for mutual attestation of two platforms. Nevertheless, the experiments focus on gathering data for a single TLS channel and the attestation of a single platform.

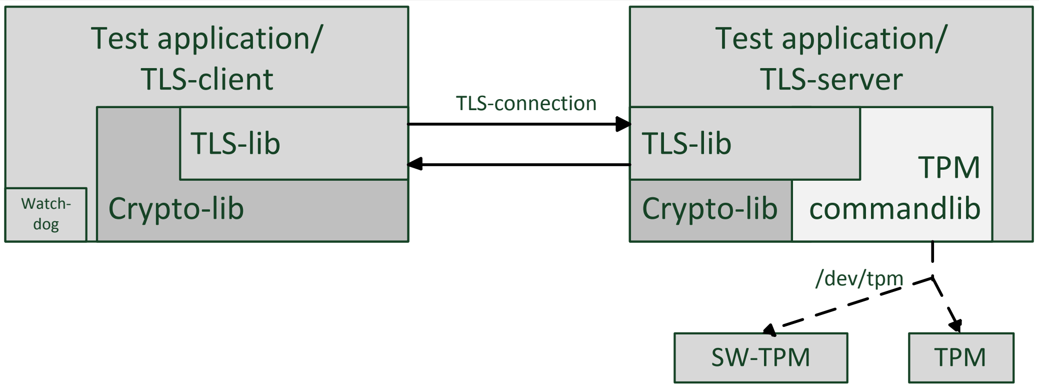
Sending and receiving Attestation requests forces the channel to block as the requesting platform has to wait for the response of the responding platform and the responding platform has to compute the attestation Quote via the TPM. The following experimental setup provides information about the effect of Quote requests and responses on the overall

TLS throughput. The test setup includes a desktop platform which can be configured to use either a hardware TPM or a software TPM.

2.2.2 Test setup

For generating the measurement values, the test setup shown in Figure 2.4 was used. The setup consists of a server and a client where the server platform can be configured to use either a hardware TPM or a software TPM. The test server- and client applications were developed in the Java programming language which used the JCE-crypto and ISASILK-TLS library from IAIK [94]. For assembling the TPM commands the TPM/J library from MIT [90] was used. The watchdog module was used to periodically issue attestation requests, either by waiting a certain amount of time or by counting the number of transmitted bytes.

Figure 2.4 Test Application Setup



The connection to the TPM is established through the TPM-TIS (TPM interface specification) driver and respectively the TPM emulator device driver. The TPMs on the server were a ST-Micro 1.2 TPM with an average speed of 806ms and and the TPM emulator from ETH Zuerich [96] with an average speed of 36ms for a single quote operation.

The measurement setup included the following equipment:

	Hardware	**Operating system**	**Virtual machine**
Server	Lenovo Thinkpad X201, Intel Quad Core 2,67 GHz, 6GB	Linux 3.1.9-1.4 x86_64	OpenJDK 1.6.0 64bit
Client	Lenovo ThinkCenter, Intel Core Dual Core 2.33 GHz, 2GB	Linux 3.1.9-1.4 x86_64	OpenJDK 1.6.0 4bit

Table 2.1: Equipment overview

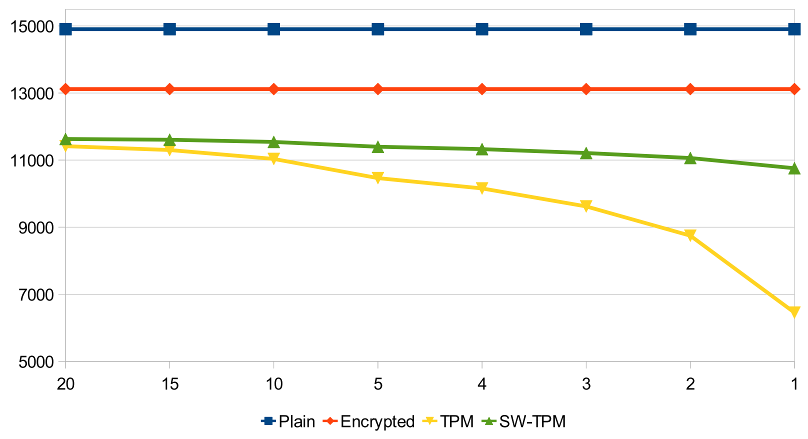
Both computers were connected directly via a 1 GBit/s Ethernet connection in order to prevent the setup to be influenced by foreign network traffic. The Java code was compiled into native code by the Java virtual machines prior to execution.

Request messages The requests are integrated directly into the TLS record-layer protocol, therefore, a new record layer message is introduced. With this message, it is possible to signal attestation requests directly via the TLS record layer, completely transparent to the application that uses the channel. The request message consists of the message type (1 byte), a *nonce* (20 bytes) plus three bytes for the selected PCRs. The response contains the Quote structure plus one event entry. The event entry consists of the PCR index that was updated (1 byte), the hash value and the event description which is limited to 120 bytes (per definition) for the test setup.

2.2.3 Remote attestation performance results

Figures 2.5 and 2.6 show the impact of remote attestation requests on the data throughput. The tests have been conducted with a genuine TPM as well as with a software emulated TPM (see test setup in subsection 2.2.2). In both diagrams, the lines marked with *Plain* denote the basic data throughput of the channel between the two test platforms without any kind of encryption or protocol overhead. *Encrypted* denotes the performance of the TLS channel (with AES 128 [26] in CBC-mode [5]) whereas *TPM* and *SW-TPM* denote the throughput when attestation requests are sent through the TLS channel and are processed by TPMs. The measured values represent only the net application databytes of the of the TLS channel.

Figure 2.5 Remote Attestation Performance with Periodic Requests (Time)

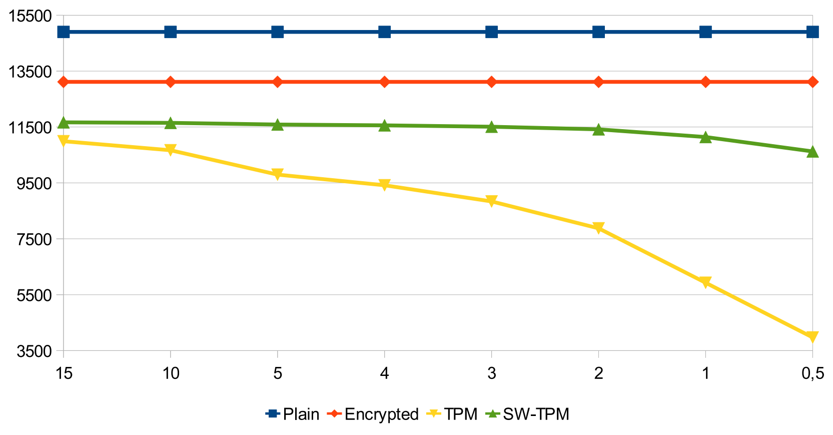


In Figure 2.5, the x-axis shows the period between the requests in seconds. The test values range from one request every 20 seconds to one request every second. The y-axis shows the data throughput in $y * 10^3$ bytes per second. The values range from the slowest case 6453,2kilo bytes/second (genuine TPM with 1 request every second) to a plain connection with $14903, 54$kBytes/second. In case of the genuine TPM, it is clearly visible that the throughput decreases rapidly with increased number of requests. In case of the emulated TPM, the decrease is smaller, but nevertheless it is about 400kBytes/s between requests that are issued every two seconds (11061,4kBytes/s) and every second (10756,1kBytes/s).

In Figure 2.6, the x-axis shows the amount of data bytes transmitted between the requests. The values range from 150000 bytes to 5000 bytes between two requests. The y-axis shows the data throughput in $y * 10^3$ bytes per second.

The values were generated by sending blocks of 2GBytes of data through the channel.

Figure 2.6 Remote Attestation Performance with Periodic Requests (Data)



During the transmission of the data, attestation requests were injected on a periodical basis resulting in 9 requests (every 20 seconds one request with a total of 182 seconds for the transmission of all 2GB) up to 269 requests in case of the *per-time-unit* test for the genuine TPM. In case of the *per-data* test, one request was sent every 150000kBytes down to 5000kBytes of data.

As TLS cipher, the AES algorithm with 128 bits key length (cipher suite: TLS_RSA_WITH_AES_128_CBC_SHA) was used. Each measurement value represents the average value of twenty test runs.

Observations The decrease of information flow stems from two sources: First, the additional messages sent over the channel are not part of the actual application data messages and have, therefore, to be removed from the net data flow throughput as they represent and additional overhead. One can easily see that the data throughput decreases with the increased number of requests. For the rate of two and one seconds between the requests (Figure 2.5), the throughput drops about 26 percent which is a large amount of data. The decrease in case of a software TPM is lower, nevertheless, the performance is reduced between 1400kBytes and 2300kBytes per second on average in relation to encrypted transmission without remote attestation.

Second, sending and receiving attestation requests forces the channel to block as the requesting platform has to wait for the response of the responding platform and the responding platform has to compute the attestation Quote via the TPM. In a different mode of implementation, requests could be processed parallel to receiving data just by putting the Quote operation and the data receive task into different threads. However, this would affect the trust status in the received data and the responses. If the requester stops sending data until he received the response of the attestation request, it has control over the point in time when they are sent as he can refuse to send more data until the request response arrives. Otherwise, the responder may reply the response at a time chosen by him which may result in malicious data to be sent to the platform before the response is submitted. Taking into account that in a period of five seconds more than $10000 * 10^3$ bytes have been sent, this is a serious amount of data which could be exploited to attack the remote platform.

Note that in the period between the requests or the amount of data passed, an arbitrary number i of events E_i might have occurred - remote attestation is based on request/response initiated by the requester and not by an actual configuration change. Consequently, a mechanism that closes the gap is inevitable.

2.3 Extending Remote Attestation for Secure and Efficient Configuration Change Reporting

2.3.1 The new model

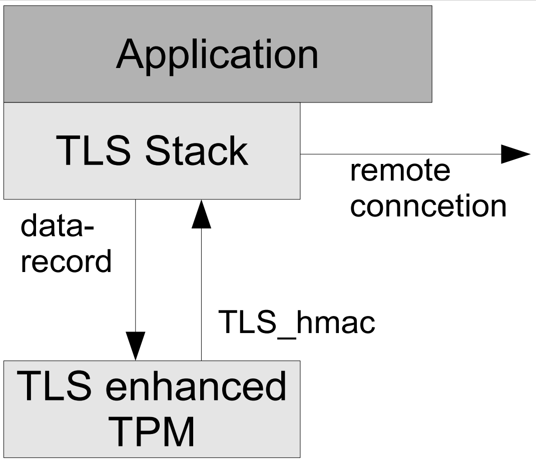
Remote attestation relies on requests and responses where a requester asks a prover for fresh attestation information. In short, it is a *passive* protocol - a platform where configuration changes occur can not actively report these changes.

The new model is based on the idea that a secure channel is cryptographically bound to the configuration of the platform and *actively* as well as securely and in a undeniable way reports changes of the configuration to a remote platform.

In common TLS implementations, the data that is sent through the secure channel is split into separated records (TLS records). For each of these records, an integrity protection value - a hashed message authentication code (HMAC) [5] - is calculated using the record data and a secret key - the MAC-key - as input (throughout the rest of the document this HMAC value is referred to as MAC). The MAC-key is only known to the local and the remote platform, which provides authentication and integrity protection for the transmitted data records.

The new concept relies on calculating the MAC values of the TLS records and the MAC-key (further referred to as TLS_{write_mac}[3]) inside the TPM. Therefore, the TLS stack sends each plain TLS record to the TPM, prior to encrypting and transmitting it to the remote platform (see Figure 2.7).

Figure 2.7 Concept of the TPM enhanced HMAC Calculation



Every record that is processed inside the TPM triggers a re-calculation of the MAC key. This key is derived from the current platform configuration and consequently changes if the configuration of the platform changes. In case of a configuration change, a remote platform will not be able to verify the HMAC values created with this key because it still uses the key that was derived from the previous configuration. Furthermore, the derivation function includes a shared secret (s_ν) that is only known to the communicating platforms. How s_ν is created and exchanged between the platforms is discussed in Section 2.3.4.

[3]The notation stems from the fact that a TLS connection consists of an outgoing channel, the *write* channel and an incoming channel, the *read* channel. Both channels use different MAC-keys for integrity protection.

A failed HMAC verification can mean three things. First, the transmitted records have been modified during transmission. Second, the configuration of the remote platform has changed since the last received record and third, the HMAC computation was done outside of the TPM.

In all cases, the remote peer has to react to this event. A simple reaction would be to send a TLS *alert* message and close the connection whereas a more previsional approach would be to restart the handshake and renegotiate new session parameters. If the new parameters and the new configuration are valid ones, the secure channel remains open, otherwise the channels is closed.

The proposed approach focuses on minimal changes to existing technology and maximal scalability and can be used with multiple TLS connections (see Section 2.3.8). A detailed discussion how the MAC key is computed is given in the following section.

2.3.2 MAC calculation and MAC key derivation

TLS records are confidentiality and integrity protected [27]. This integrity protection is achieved by applying a *HMAC* [5] with a given secret key on the application data. This secret key is derived from a *pre-master secret*; the procedure how the key is exactly derived can be found in [86]. To be more detailed, TLS requires two different secret keys as it uses a read and a write channel for one single connection. Consequently, TLS requires a *read mac key* and a *write mac key*. How the read mac key is generated is discussed in the next section. For now focus is laid on the further processing of the write mac key that is refer to as TLS_{write_mac} from now on.

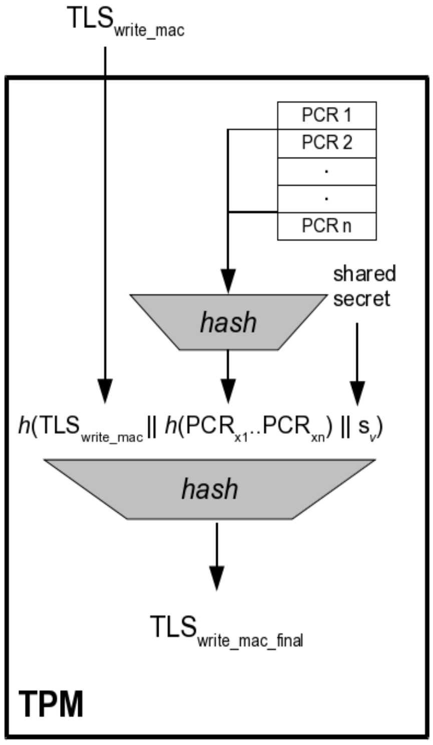
TLS_{write_mac} for the mac computation is not used directly - it is rather used as material for generating a new key inside the TPM. For calculating the final TLS write mac key ($TLS_{write\_mac\_final}$) that serves for the actual integrity protection, the TPM computes a hash from a set of PCR registers, the shared secret s_ν and the original TLS_{write_mac}. The hash over the set of PCRs is computed via $PCR_{hash} = h(PCR_{x_1}, PRC_{x_2}, \ldots, PCR_{x_n})$ where h denotes a hash function - preferable SHA-1 [78] as this algorithm is supported by all TPMs. With PCR_{hash} we can now calculate $TLS_{write\_mac\_final}$ via $TLS_{write\_mac\_final} = h(TLS_{write\_mac} || PCR_{hash} || s_\nu)$ (see Figure 2.8). Which PCRs are used for the key derivation has to be negotiated in the TLS handshake.

Two things can be achieved by including PCR_{hash} and TLS_{write_mac} in the HMAC calculation. First, the current TLS session is bound to the final key. Second, if a configuration change is recorded in the TPM, the content of one PCR is extended with the new measurement value (as defined in [112]). Consequently, the key changes when the configuration is changed as it is derived from the PCRs. In order to have a proof that all operations have been done inside the TPM, s_ν is also included in the MAC calculation. How s_ν is applied for this purpose is discussed in Section 2.3.4.

So far, the key for the HMAC calculation was derived. The HMAC of the TLS records can then be calculated from the sequence number the record type, version number, length of the application data and the application data itself.
$TLS_{HMAC}(data) = HMAC(TLS_{write\_mac\_final}, s || t || v || l || data)$ whereat s denotes the sequence number, t the type, v the version, l the length and *data* the application data of the TLS record. The TPM now returns $TLS_{HMAC}(data)$ to the TLS stack. The stack puts *data* and $HMAC(data)$ together to a TLS record, encrypts this record and sends it to the remote platform.

How the remote peer verifies the HMAC and how it can detect configuration changes

Figure 2.8 HMAC Key Derivation



is discussed in the following section.

2.3.3 TLS record verification

Once the TLS records have been sent to the remote platform, the TLS stack of the platform decrypts the incoming records and checks their integrity by verifying the HMAC over each record [27]. For a positive verification of the HMAC, the remote platform requires three things. First, TLS_{read_mac} that is equal to TLS_{write_mac} from the sending platform where TLS_{read_mac} is calculated the same way as TLS_{write_mac} (a description of the process can be found in [27]). Second, $TLS_{read\_mac\_final}$ is required. It is derived from TLS_{read_mac} and the hash over a set of PCRs. Although the PCR values of the remote platform are not available on this platform, the configuration information can be extracted from the TPM_QUOTE_INFO structure [53]. This structure is created and exchanged before the channel is established. The structure includes two constant parameters p_1 = "1100", p_2 = "$QUOT''$, the composite PCR hash $p_3 = hash$ and $p_4 = nonce$. Third, the shared secret s_ν that was created by the stack itself. With this information $TLS_{read\_mac\_final} = h(TLS_{read\_mac} || p_3 || s_\nu)$ can be computed now. The integrity of the incoming TLS records is then verified via $TLS\_MAC \overset{!}{=} HMAC(TLS_{read\_mac\_final}, record)$[4]

[4]It should be noted that these HMAC verification steps, in contrast to the HMAC creation steps, do not necessarily have to be done inside the TPM - they can be processed by the TLS stack.

2.3.4 Binding the TLS channel to the TPM

All calculations discussed so far can be done outside the TPM. Therefore, a reliable proof that all computations are actually done inside are needed. Only if we can show that all computations were done inside, we have the definite evidence that the configuration changes are reported in a secure way. This proof can easily be achieved by inserting a shared secret into the HMAC calculation (s_ν) that is only known to the communication partners - on the attester's side even only to its TPM. This way, the secure channel is bound to the specified TPM. Consequently, s_ν has to be exchanged in a confidential way before the channel is opened. Unfortunately, TPMs do not have the capability to decrypt and load arbitrary data. Therefore, a mechanism to decrypt s_ν inside a TPM is required. The most reasonable approach is to use AIKs. Currently, AIKs can only be used for signing operations. In order to allow encryption operations for the proposed approach, this usage limitation of AIKs is removed. A remote party is now able to use a public AIK for encrypting s_ν whereas the local TPM is now allowed to decrypt s_ν. Using an AIK for this task has three advantages: First, the required confidentiality is guaranteed as public-key cryptography is involved. Second, the AIK certificate proves that the receiver of s_ν is actually the addressed TPM. Third, the anonymity of the remote platform can still be guaranteed.
The procedure for exchanging s_ν is the following: The TLS stack of the local platform computes s_ν, encrypts it with a public AIK of the targeted TPM and sends it to the remote platform. The remote platform passes the encrypted s_ν to its TPM that decrypts and stores s_ν until the session is closed.

2.3.5 Session initialization

In this section, the required initialization and setup steps for the TPM and the TLS stack are discussed. These components require different initialization values depending on whether they are operating on the remote or the local platform. The local TPM requires a TLS_{write_mac} which can be obtained from the TLS stack key derivation process. The local TLS stack, however, requires the initial TPM_QUOTE_INFO structure and the AIK certificate from the remote platform. The certificate is required to verify the authenticity of the remote TPM by validating the AIK certificate whereas TPM_QUOTE_INFO contains the current configuration. This configuration information is required by the stack to compute $TLS_{read\_mac\_final}$. For this calculation, the TLS stack also requires TLS_{read_mac} which can also be obtained as described in [86]. Consequently, TPM_QUOTE_INFO and the AIK certificate must be exchanged with the remote platform before $TLS_{write\_mac\_final}$ and $TLS_{read\_mac\_final}$ can be computed. The TPM_QUOTE_INFO can be obtained from the remote TPM by invoking a TPM_Quote command (assuming that a certified attestation identity key (AIK) is already available on the platform) with a given set of PCR numbers. Moreover, the involved TPMs require a shared secret s_ν (for mutual attestation actually two) that also have to be exchanged before the HMAC computation can start.

2.3.6 Design and implementation of the prototype

The restrictions of current TPMs require adaptions of the existing TPM implementation. For the proof-of-concept prototype the freely available TPM emulator [96] was modified and additional commands and operations discussed in Section 2.3.7 were added. Furthermore, for testing purposes the handshake procedure of the open source TLS implementation gnuTLS [41] was modified which now supports an additional data structure that includes attestation information. This attestation information is composed of a signed TPM_QUOTE_INFO structure, the certificate of the signing AIK and the encrypted shared secret. Other variants how this information can be integrated into TLS can be found in Section 2.1.1.

Moreover, gnuTLS was modified in a way that the receiving platform can validate the MACs of the TLS records according to the new concept. This validation includes the derivation of $TLS_{read\_mac\_final}$ form the PCR hash value and the TLS_{read_mac} and the modified validation process itself. Additionally, gnuTLS was modified to compute and encrypt s_ν with a public AIK.

The TPM software stack TROUSERS [56] (TSS) was updated in order to support the required additional commands. Moreover, the stack was integrated into the gnuTLS library so that it is able to access the TPM functions.

2.3.7 TPM modifications

The proposed architecture requires some modifications of the current TPM specification. In detail, the TPM has to support a few additional commands, the key derivation process as discussed in Section 2.8 and the binding of the secure channel to the TPM (Section 2.3.4).
In order to implement the proposed concept, the TPM must support the $TLS_{write\_mac\_final}$ key derivation process as discussed in Section 2.3.2. The process can easily be integrated into existing TPM implementations as it only requires two additional hash calculations. As hash algorithms are available on common TPMs, new cryptographic algorithms are not required as long as the supported cipher suites are limited to cipher suites that use SHA-1 for integrity protection.
Another modification includes the binding of the channel to the TPM. For this binding, s_ν has to be imported into the TPM securely. Therefore, a modification of the AIK usage constraints is required.

Although TPMs support message digest algorithms, they do not support message authentication codes as required by TLS. Therefore, the proof-of-concept prototype was extended to support the necessary HMAC functionality.

2.3.8 Handling multiple TLS channels

Typical scenarios that utilize trusted channels often involve multiple concurrent connections. They can occur if a browser opens different connections to different peers. For security reasons, these connections should use different cryptographic parameters that are, in this case, different keys for encryption and HMAC calculation. Therefore, it is desirable to have an approach that also supports concurrent channels.

The newly proposed approach is able to handle multiple connections. Each connection instance requires its own secret key (i.e. $TLS_{write\_mac\_final}$). This can be guaranteed as each key is re-calculated for each TLS record with the parameters provided by the content of the PCRs and s_ν.

2.3.9 Impact on performance

In contrast to the original design this approach uses symmetric signatures to bind the configuration information on the data. This results in a performance improvement attestation requests and their handling can be avoided. The results are discussed in detail in section 2.5.4

2.4 Alternative MAC Key Derivation

In the previously discussed approach, the MAC-key from the standard TLS key derivation process was used to further derive the final MAC-key inside the TPM. A different approach could be to perform the common key derivation directly inside the TPM instead of using the output of the TLS stack key derivation. The TLS specification defines this derivation process as

$$key = PRF(master\ secret, "key\ expansion", random_{server} + random_{client}). \tag{2.5}$$

The mac-key is derived from a master secret (MS) which is derived from a pre-master secret. Details about this process can be found in [28]. The session key and the mac-key material are derived from the MS. The key material is computed as follows:$key = PRF(master\ secret, "key\ expansion", server_{random} + client_{random})$ where the pseudo random function (PRF) is defined as: $PRF(secret,$
$label, seed) = MD5(S_l, label + seed) \oplus \text{SHA-1}(S_r, label + seed)$. (where S_l denotes the leftmost bits of the master secret and S_r denotes the rightmost bits of the master secret).

The TPM could then compute the mac-key via:

$$\begin{aligned} key =& PRF(master\ secret, "key\ expansion", random_{server} + random_{client},\\ & h(PCR_{x_1}..PCR_{x_n})). \end{aligned} \tag{2.6}$$

In this case, the PRF must be modified in such a way that it includes the hash of the PCR registers. This could be achieved in two ways: The hash of the PCR registers could either be computed by

$$\begin{aligned} PRF(secret, label, seed, h(PCRs) =& MD5(S_1 \| label + seed \| h(PCRs)) \oplus \\ & \text{SHA-1}(S_2 \| label + seed \| h(PCRs)). \end{aligned} \tag{2.7}$$

or

$$\begin{aligned} PRF(secret, label, seed, PCRs) =& h(MD5(S_1 \| label + seed) \oplus \\ & SHA-1(S_2 \| label + seed) \| PCR_{x_1}...PCR_{x_n} \| s_\nu). \end{aligned} \tag{2.8}$$

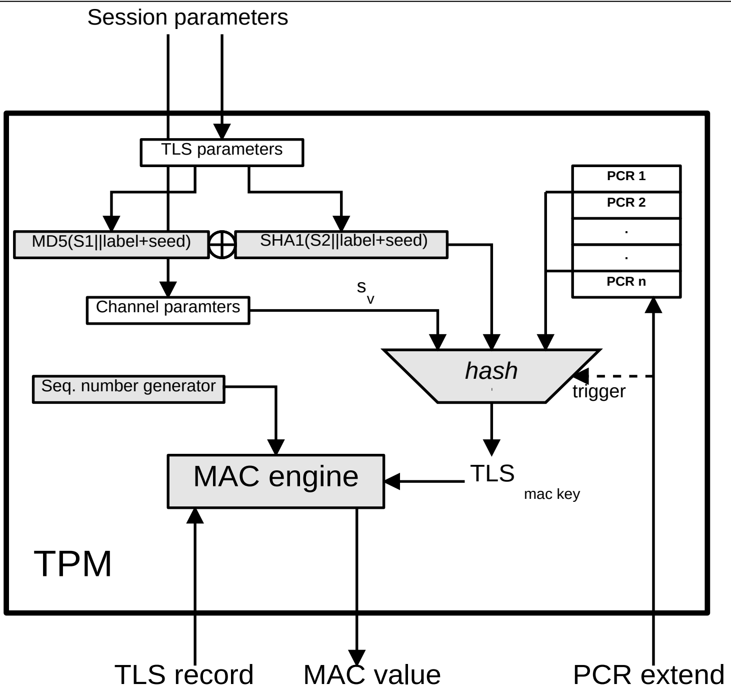
The first PRF requires the computation of an additional hash operation whereas in the second PRF the content of the PCRs is hashed together with *secret*, *label* and *seed*. For further analysis, the second approach is selected as it is the easiest one to implement and requires only one hash calculation to be performed when a PCR is extended.

In contrast to the original approach discussed in Section 2.3, the session key is now derived and stored inside the TPM. Storing the key obsoletes the requirement of sending the MAC-key computed in the TLS stack to the TPM every time a new MAC value is computed thereby reducing the communication overhead.

2.4.1 Session parameter setup

In this section, the required initialization and setup steps for the TPM and the TLS stack with respect to the new key derivation approach are discussed. For the first approach, the local TPM requires a TLS_{write_mac} which can be obtained from the TLS stack key derivation process. The local TLS stack, however, requires the initial TPM_QUOTE_INFO

Figure 2.9 Alternative HMAC Key Derivation



structure and the AIK certificate from the remote platform. The certificate is required to verify the authenticity of the remote TPM by validating the AIK certificate, whereas TPM_QUOTE_INFO contains the current configuration. This configuration information is required by the stack to compute $TLS_{read\_mac\_final}$. For this calculation, the TLS stack also requires TLS_{read_mac} which can additionally be obtained as described in [86]. Consequently, TPM_QUOTE_INFO and the AIK certificate must be exchanged with the remote platform before $TLS_{write\_mac\_final}$ and $TLS_{read\_mac\_final}$ can be computed. Moreover, both platforms require a shared secret s_ν that has to be exchanged before the HMAC computation can start. This is also true for the second approach. Moreover, this second approach has to initialize the TPM prior starting the TLS session. This requires the stack to send the master secret, client and server random to the TPM, as well.

2.4.2 Binding TLS channels to different processes

The approaches discussed in the previous sections focus on single connections. On modern platforms, concurrent connections are often used - not only on server systems but also on desktop and embedded systems. Concurrent connections can be initiated by e.g. different processes within a single machine with one single TPM or, in virtualized environments, with many different virtual TPMs. In any case, an application must not be able to read or use s_ν from a different process under the assumption that the process isolation mechanism of the underlying operating system works flawlessly.

In order to achieve this access protection, the TPM must provide an authentication mechanism that can be used by each TLS stack instance. Therefore, the approaches and prototypes discussed in this section use the standard authentication mechanisms and protocols e.g. OIAP, OSAP provided by the TPM specification.

2.4.3 TPM enhancements

The proposed mac-key computation algorithms require modifications of the current TPM specification. The TPM has to support a few additional commands which is the key derivation process as discussed in Section 2.3 and Section 2.4.

In order to implement the original concept, the TPM has to support the $TLS_{write\_mac\_final}$ key derivation process as discussed in Section 2.3. The process can easily be integrated in existing TPM implementations as it only requires two additional hash calculations. As hash algorithms are available on common TPMs, new cryptographic algorithms are not required as long as the supported cipher suites are limited to cipher suites that use sha1 for integrity protection. In order to implement the original concept, the TPM has to support the $TLS_{write\_mac\_final}$ key derivation process (as discussed in Section 2.3). The process can easily be integrated in existing TPM implementations as it only requires two additional hash calculations. As hash algorithms are available on common TPMs, new cryptographic algorithms are not required as long as the supported cipher suites are limited to cipher suites that use SHA-1 for integrity protection.

For the proof-of-concept prototype, the following list of commands was implemented:

- TPM_LoadSharedSecret(TPM_KEY_HANDLE aik key handle, UINT32 encrypted shared secret size, BYTE[] encrypted shared secret). This command initializes a new TLS session and downloads the encrypted secret s_ν from the remote platform and the handle to the AIK it was encrypted with. The response of the command is a shared secret of the type TPM_KEY_HANDLE, a handle to the shared secret within the TPM.

- TPM_ComputeMac(TPM_KEY_HANDLE shared secret handle, UINT32 keySize, BYTE[] HMAC key, TPM_PCR_SELECTION pcrs, UINT32 dataSize, BYTE[] data). This command instructs the TPM to compute the derived mac-key and mac-value of the provided data, including the shared secret addressed by *shared secret handle*. The result of this command is a TPM_DIGEST mac-value. When a session is closed, the shared secret is be deleted from the TPM, otherwise it is deleted after a defined time-interval.

- TPM_ClearSharedSecret(TPM_KEY_HANDLE shared secret handle) This command forces the TPM to clear the shared secret addressed by *shared secret handle*.

For analysis reasons, these commands were implemented as authorized as well as unauthorized commands. However, for a maximum level of security, applications should use the authorized set of commands.

For the second approach, the following commands were additionally implemented :

- TPM_SetSessionParameters(UINT32 master secret size, BYTE[] master secret, UINT32 client rnd size, BYTE[] client random, UINT32 server rnd size, BYTE[] server random, UINT32 enc shared secret size, BYTE[] encrypted shared secret, TPM_PCR_SELECTION pcrs). This command initialized a new TLS session by loading the master secret, the shared secret s_ν, client and server random and the PCR selection into the TPM. Moreover, this method resets the sequence counter.

- TPM_ComputeMac(UINT32 dataSize, BYTE[] data). This method computes and returns the MAC value for the provided data.

2.4.4 Observations

Experiments with the proof-of-concept prototype showed that both approaches are feasible. However, the approaches discussed in this section require some additional modifications of the TPM in comparison with the original approach. Moreover, the second approach requires more session specific data and additional algorithms to be stored inside the TPM. To be more detailed, the differences are: The TPM has to receive additional key derivation parameters i.e. master secret, client and server random and has to store it in the TPM every time a new session is initialized. Moreover, the TPM has to support the modified pseudo random function including the additional MD5 hash algorithm as required by the TLS pseudo random function. Implementing additional hash algorithms and a PRF into the TPM reduces the flexibility of the original approach, where most of the algorithms are implemented in software. This software can easily be replaced in future TLS implementations when new algorithms are defined in the specifications.

Nevertheless, due to the fewer data that has to be transferred to the TPM, the second approach has a performance advantage over the first approach. Both approaches require that - every time a new event is recorded - a new key is derived in order to reflect the full event history in the integrity computation. Moreover, the session parameters or the $TLS_{read\_mac\_final}$ key have to be stored inside the TPM requiring additional resources. The total required amount of resources depends on the number of concurrent TLS connections but is limited by the actual capabilities of the TPM. While this approach is unproblematic in combination with software TPMs, it may be inefficient resulting from the memory constraints of hardware based TPMs.

The efficiency of both approaches may also be increased by sending the hash of the TLS record instead the entire record to the TPM. This method allows a more efficient usage of the low-pin-count (LPC) bus[5]. However, this approach requires a modification of the TLS MAC verification algorithm as the verifier also has to compute the hash of the record after decrypting it and before checking its integrity.

From the security's point of view, both approaches provide a high level of security. All computations are done inside a tamper-resistant device. Moreover, manipulation of the session parameters requires authentication which is employed by the TPMs internal authentication mechanisms and protocols (i.e. OIAP). A modification of the platforms software configuration results in a modification of one or more PCRs. The final session keys are derived from the current stage of PCRs, hence, the configuration change is reflected in the MAC of the TLS records. The TLS records are encrypted so that the content cannot be modified. A possible attack could be to modify the encrypted data stream where an adversary could flip certain bits of the data stream. The attack would result in a denial of service as the receiving platform would have to reject the incoming packets and would have to negotiate new session parameters. This attack is not special to the proposed approach as it is also a threat to any other TLS connection.

The authentication keys are solely used and derived inside the TPM. Hence, an adversary - even if he were able to take over the platform - is not able to read or manipulate the keys. Depending on which mechanism the adversary used to take over the platform, the event is either registered or not. For example, if the attacker used a trusted software that was infected by a Virus or Trojan Horse, the event of loading this software is registered. In addition, the PCRs where the event was recorded reflects that instead of the original

[5]The LPC bus is the interface used on common desktop systems to connect a TPM to the computing platform.

software the Virus infected version was loaded. Hence, the TLS MAC-keys will also reflect this fact and a remote platform can undeniably identify that a malicious software was loaded on the local platform.

However, if the adversary uses attack methods like buffer-overruns, it is possible to circumvent the recording of that event. In this case, a remote platform would not be able to detect the malicious events on the local platform. This kind of attack is a general threat to Trusted Computing platforms and cannot be prevented by this technology. Therefore, a detailed discussion of the implications of such attack methodologies on Trusted Computing is out-of-scope of this thesis.

Hardware attacks on TPMs are not considered in the discussion as they are not part of the TCG's threat model.

2.5 Enhancements to Secure Platform Configuration Change Reporting

The approaches discussed so-far focus on trusted systems were a more or less static configuration is assumed. This means that changes of the platform configuration i.e. loading and unloading of software modules occurs sporadically. This raises the question of how this approach can be adapted for platforms that have continuous or very frequently changing configurations.
The challenge in this case lies in the on-time reportage and validation of the configuration change without losing the cryptographic bond between PCRs and TLS channel. This means that - instead of sending a *Quote* for every PCR change reported - concrete PCR change information has to be sent along with meta- and status data of the TLS channel.
Putting the previously discussed approaches into a scenario with a high PCR change frequency one problem becomes imminent: congestion.
Continuously triggering of new PCR change reporting might lead to a congestion of the network instead of transmitting application data and the TLS channel is busy with execution handshake protocols and transporting PCR change information.
The instant of time when this congestion occurs depends on different factors like network band-width, platform workload and platform performance etc. Although the congestion may vanish after a certain amount of time, scenarios where this behavior is not desired exist. Applications like video broadcast or which have real-time demands and which require a constant flow of data would lose much of their performance and would eventually become unusable.

In order to avoid this kind of congestion, modifications of the reporting mechanism have to be applied.

2.5.1 Event reporting

Until now, only the secure notification of PCR change events was cryptographically integrated into the integrity check of the TLS channel. In order to derive the MAC-key and to verify the integrity value, the remote platform also requires the measurement values that were extended into the PCRs. Therefore, a method to efficiently transport this information to the platform is required.

TLS record layer modifications

TLS is specified as a set of different protocols and layers [86]. In order to improve the configuration change reporting mechanism, one of these layers - concrete the *record layer* - has to be modified. The record layer is responsible for splitting the record layer messages and application data into fragments and on the receiver side to re-assemble the transported data into records of at maximum 2^{14} bytes of length and for providing encryption and integrity checks on these data records.

In order to send and receive data, TLS packs the data into a *record layer* structure. Listing 2.1

Listing 2.1: TLS Record Layer Structures

```
struct {
        uint8 major, minor;
} ProtocolVersion;
```

```
enum {
        change_cipher_spec(20), alert(21), handshake(22),
        application_data(23), (255)
} ContentType;

struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
        opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

A TLS data record (TLSPLaintext) consists of a structure that includes the *record type*, the *protocol version*, the *plain data length* and the *plain data field* as the payload. In general, the content of the plain data field (opaque fragment field) is application specific data and is transparently handed over to the application - it is not interpreted by the TLS layer in any way.

In order to signal the PCR change as well as the newly added configuration information, advantage can be taken of the fact that the *opaque fragment* filed may contain arbitrary data, hence, it may also carry another constructed structure. The record layer is modified simply by adding an additional message type. In addition to the messages *change_cipher_spec*, *alert* etc. the new message type *PCR_extend(24)* (similar to the TPM command) is introduced. This message informs the remote platform that: first, a change has occurred, second, which value has been extended and third, which software module has been loaded. In addition, the message carries a quantum of application data, all stored together in the opaque fragment field of the TLSPlaintextstructure. The difference to the standard TLS record layer protocol is that the information in the plaintextfield is now context specific and therefore interpreted by the receiver of the record message.

The modified opaque fragment field is defined as:

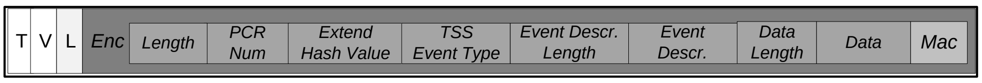
Listing 2.2: TLS PCR_extend Message

```
struct {
        uint32 PCR_num;
        extend_hash_value [sha1.length];
        uint32 tss_eventtype;
        uint32 event_description_length
        event_description [event_description_length]
        uint16 length;
        opaque fragment[length];
} event_fragment;
```

The new fragment structure now contains the PCR number of the PCR that has been extended, the hash value that was extended into the PCR, the type of event that occurred, and the description of the event. The final field element is the *opaque fragment* that contains the application data as defined in the original structure. Figure 2.10 shows the layout of the new message with respect to the existing record layer messages.

While the application data transported within the opaque fragment is still handed over to the application un-interpreted, the other entries of the field are now validated by the receiver side. The receiver side has to re-calculate the mac_{new} based on the PCR hash values sent in along with the new fragment. However, this is a delicate situation as the information received is at the moment - due to the new PCR values - signed with a new,

Figure 2.10 TLS Application Data Message Structure including Event Data



and not yet validated mac_{new} key. In the moment the PCR value is extended, the MAC-key key of the signer is changed, hence, the verifier cannot correctly verify the TLS records until he has re-calculated his MAC-key with then new PCR values. To overcome this situation, the verifier reads the event description information, validates it, recomputes its MAC verification key and validates the integrity of the received TLS fragment. To *validate* the information means that the verifier checks the entries of the `event_description` field which is typically the name of the software that has been loaded. Only after successful validation of this entry and the TLS fragment, the fragment is either accepted or the TLS channels is closed.

Alternatively, it would also be possible to modify the original `TLSPaintextstructure` field instead of adding a new message. However, the proposed approach only requires minor modifications, i.e. an additional message. Moreover, integration with legacy TLS implementations is simple as the basic is not changed and TLS messages, other than the new one, still can be processed.

Every TLS fragment sent has to reflect the current configuration of the PCRs at the time it was sent. In addition, the receiving platform has to be informed of every event that occurred in the sending platform. This is of special interest if the secure channel is not transmitting data for a time when PCR changes occur.

There are two approaches to solve this situation: Firstly, the platform immediately sends an empty packet that contains only the newly computed PCR values and PCR event fragment structures when a PCR event takes place, thereby transmitting the event information and triggering the update process of the MAC-key and PCR value list on the receiver side. Secondly, the platform gathers and stores all PCR changes and puts them into a single fragment triggering the recomputing when the next packet of data is sent.

While the second approach takes advantage of the fact that all required information is stored in the SML file and therefore only the changes between now and the time when the last packets we sent have to be used for re re-computation, the first approach requires a notification mechanism between TSS and the record layer. It is required that the TSS not only sends a command to the TPM to extend the content of a certain PCR, it must also notify the TLS implementation that a change has occurred in order to trigger the transmission of the new PCR values immediately.

However, storing the event information and putting it into a single fragment may exceed the fragments size limit requiring a split of the events into different fragments. In this case, the sender is required to send a set of fragments.

For case one and for compatibility reasons, a new record layer message is introduced (`PCR_change`) that signals the the PCR change event to the receiving platform and that contains the PCR event structure.

Another case when information can not be broadcast over the network is *network congestion*. In this case, the sending platform is not able to send the accumulated event fragments. Therefore, the fragments may be stored on a sending platform until the required network capacity is available again.

2.5.2 TLS record verification and event reporting

As in the previously discussed approaches, the TLS stack of the remote platform decrypts the incoming records and checks their integrity by verifying the MAC value over each record. However, instead of closing the connection the verifying platform is now able to re-calculate the MAC-key based on the reported PCR events.

For a positive verification of the MAC value, the remote platform still requires three things. First, TLS_{read_mac} that is equal to TLS_{write_mac} from the sending platform where TLS_{read_mac} is calculated the same way as TLS_{write_mac}. Second, $TLS_{read\_mac\_final}$ is required which is derived from TLS_{read_mac} and the hash over the set of known PCR values and reported PCRs that are extracted from the PCR event messages. Third, the shared secret s_ν that was created by the stack itself. With this information $TLS_{read\_mac\_final} = h(TLS_{read\_mac}||p||s_\nu)$ can be computed. Where $p = h(PCR_1||...||PCR_m)$ is the hash of the reported PCR value changes. The integrity of the incoming TLS records is then verified via $TLS\_MAC \overset{!}{=} HMAC(TLS_{read\_mac\_final}, record)$

This modified procedure for deriving the MAC-key has the following consequences: In case of a configuration change, a remote platform will not be able to verify the MAC values unless it recomputes the key with the newly reported events. This is especially true for the record that carries the PCR events. As a consequence, the remote platform has to validate the reported events prior to MAC-key derivation and it may only derive and use the new key if the events are considered valid and trustworthy.

2.5.3 Session parameter setup

At the start of the session, the local TPM requires a TLS_{write_mac} which can be obtained from the TLS stack key derivation process. In addition, the local TLS stack requires an initial TPM_QUOTE_INFO structure plus the content of the PCRs and the AIK certificate from the remote platform. The certificate is required to verify the authenticity of the remote TPM by validating the AIK certificate, whereas TPM_QUOTE_INFO contains the current configuration. This configuration information is required by the stack to compute $TLS_{read\_mac\_final}$. For this calculation, the TLS stack also requires s_ν . Consequently, TPM_QUOTE_INFO, the AIK certificate and s_ν must be exchanged between the platforms before $TLS_{write\_mac\_final}$ and $TLS_{read\_mac\_final}$ can be computed.

2.5.4 Security and performance considerations

In contrast to the model discussed in the previous sections 2.3, 2.4, this model does not rely on the asymmetric *Quote* operation in order to signal events. The configuration change events are rather integrated into the data stream instead of sending an explicit event reporting message. A *Quote* is only reported once, at the begin of the session to establish a common event reporting starting point.

As a result, a continuous stream of application data can be guaranteed and channel congestion by prevalent *Quote* operations can be prevented. The security of this design relies on the security of the secure channel, of the TPM - all relevant computations on the attester's side are executed inside the trusted TPM - and the chain-of-trust provided by the trusted attester platform.
An adversary might try to execute the following attack vectors:

1. **Modifications of the TLS record.** If an adversary succeeds in modifying the reported PCR events during transmission, for example by modifying single bits of

the encrypted fragment, the derived MAC-key would be different to the original $TLS_{write\_mac\_final}$. As a consequence, the verifying platform would detect this modification during integrity check. Other modifications of the record are also detected during integrity value validation.

2. **Report of false or faked events.** Malicious software might be able to modify the reported PCR events that are transmitted via the TLS record messages. As the events are recorded in the PCRs and $TLS_{write\_mac\_final}$ is derived from the content of these registers, the verifier is not able to derive the correct $TLS_{read\_mac\_final}$. Consequently, the MAC validation would fail.

3. **Attack during network congestion.** In case of network congestion, the single TLS record fragments have to be temporarily stored on the sender's side. Nevertheless, integrity and confidentiality of the fragments can be guaranteed. Although malicious software might be able to extract the encryption key and decrypt the record, it would not be able to extract the integrity MAC-key from the TPM. Therefore, when the package is sent, the remote platform is informed of the presence of this malicious software.

4. **Resend previously sent records.** An adversary could try to resend a TLS fragment recorded from an earlier session. This attack would fail on different occasions. First, for every new TLS session new session parameters are used. Hence, decryption and integrity validation would fail. Second, every TLS record has its own sequence number. Even if he manages to inject the false record at the correct position, the verifier would detect the false record as the sequence number is use during computation of the integrity value. Assuming that an adversary is able to successfully attack the verifier platform and is able to get into possession of $TLS_{read\_mac\_final}$, the encryption key and s_{ν} he would be able to intercept and modify the transmitted data. Neither Trusted Computing, TLS or the newly proposed approaches are able to prevent this attack. Consequently, for real trustworthy connections also the integrity of the target platform must be taken into account. Therefore, mutual attestation is the suggested method.

Loading malicious software into the system causes reporting of this event to the TPM. Consequently, the PCR values and the $TLS_{write\_mac\_final}$ for the next record are changed. The malicious software is only able to manipulate the TLS_{write_mac} key and the transported data including the reported PCR event. However, the remote platform that derives the MAC-key for verification from the reported PCR events and s_{ν} can detect these changes as it is not able to validate the MAC value of the record. This is also true for TLS records generated in a previous session if they are re-sent to the platform. Furthermore, the computation of the MAC is done inside the TPM and can therefore not be manipulated from the outside.

Integrity information (MAC values) generated for a different, valid record cannot be applied to authenticate a malicious record as the record and the integrity value are cryptographically connected by the MAC computation.

Resources of the TPM are very limited, hence, they must be carefully used by limiting the amount of additional data that is stored in the TPM. In this approach, the TPM only has to store s_{ν} (which is 20bytes) for every open connection.

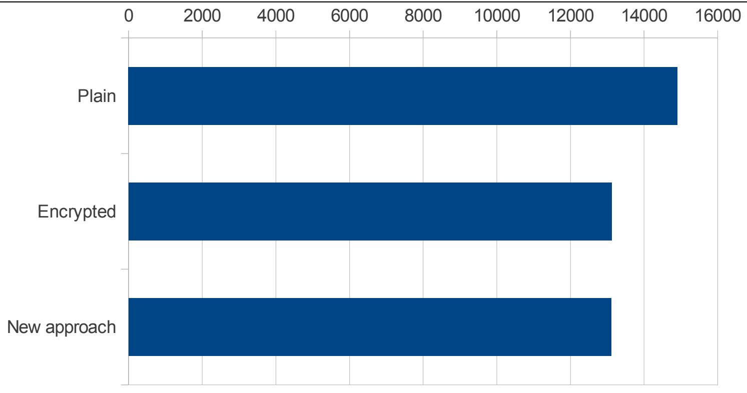
The MAC-keys ($TLS_{write\_mac\_final}$ and $TLS_{read\_mac\_final}$) should not be re-used for two different TLS session. This can be guaranteed by either using a new s_{ν} or by using a different TLS_{write_mac} respectively TLS_{read_mac} for every new session. As the values s_ν and TLS_{read_mac} are controlled by the verifier, the verifier has control over these values.

The initial remote attestation step allows the requester to check if the attester actually executed an authenticated boot and allows it to determine the initial state of the platform.

In order to obtain reference values, a proof-of-concept prototype was implemented. The basis for this prototype is the proof-of-concept implementation introduced in 2.3.6 with additional modifications to the TLS record layer on the sender and the receiver side. While the sender constructs the packet with the new structure containing the `PCR_event` message, the receiver has to dis-assemble the structure in the record layer and recalculate the PCR content as well as validate the content of the structure.

The comparison with plain- and encrypted data flow of the TLS channel shows that the new approach only marginally reduces the data throughput. The sending of the record data, the TLS_{write_mac}-key as well as the derivation and computation of the MAC-data is executed in the TPM within 2ms on the test platform. As a result, the performance of the approach (13101,69kByte/s) nearly reaches the performance of the encrypted channel (13116,6kByte/s) and is much faster than the original remote attestation mechanism defined by TCG with genuine and software based TPMs.

Figure 2.11 Comparison with encrypted and plain data throughput



For genuine TPMs that rely on the slow LPC bus to transport the information, this procedure can even be more enhanced by sending the hash of the record data instead of the full record. Consequently, the verifier has to modify the validation step according to $TLS_{mac} = HMAC(TLS_{read\_key\_final}, H(TLS_{record}))$.

Summing up, the new approach has the following advantages over the original TCG remote-attestation design:

1. Changes can be detected immediately, thereby omitting the gap between two attestation requests where no reliable information of the platform configuration is available for the requesting platform.

2. The data transported through the channel is tagged with configuration of the platform at the time it was sent. With this tagging, the receiving platform is able to determine the exact configuration of the platform when the data was sent at any - even later - point in time.

3. The time required for computing the configuration proofs (i.e. quote) is lower as symmetric cryptography is employed. Therefore, the reduction of data throughput through the TLS channels is minimal. Keep in mind that the values presented in diagrams 2.5 and 2.6 represent the results of one channel in one direction. In case of mutual attestation where two channels are used, the throughput decreases even further.

4. The fact that the MAC computation is done inside the TPM provides a secure proof that the reported events were not manipulated.

Moreover, event messages are handled by the record layer and are therefore transparent to the application. Further processing of this data may be handed over to the operating system or a trusted service.

2.6 General Considerations

In this Section, a general discussion of the influence on different components and mechanisms of TLS and Trusted Computing is given. Moreover, the relation of the proposed mechanisms to mobile trusted computing is investigated and mechanisms that allow to use TPMs in common authentication algorithms are proposed.

2.6.1 Impact on advanced TLS features

Impact on Session-resume TLS supports two kinds of session resume mechanisms. In the first mechanism, client and server independently store their specific session information. The session information includes items like the session keys or the master secret. In the second mechanism, the session information - including the items from the server - are stored on the client in order to save space on the server side [89]. The server, therefore, sends the encrypted information to the client in form of a *ticket*. The client is henceforward able to resume interrupted sessions by presenting the ticket to the server.

The proposed algorithms for configuration change reporting are not applicable to the session-resume mechanism. After presenting the ticket, client and server have to exchange a Quote as new PCR reporting starting point and negotiate new session parameters for deriving the keys making the resume mechanism obsolete.

However, modifications to the resume process where e.g. the client receives the events that occurred from disconnection to session resume, enabling him to re-calculate the keys, can solve this problem.

Impact on truncated HMACs TLS provides an optional mechanism to save network bandwidth which is called truncated HMAC [25]. This mechanism simply transmits only the first 80 bits of the TLS record HMAC and skips the remaining. While this kind of check is sufficient for integrity check in common TLS connections, it can not be used for signaling configuration changes. Because a configuration change could affect the bits in the skipped part (the last 40 bits) of the HMAC the change cannot be detected. Therefore, truncated HMACs can not be used with this configuration report approach.

2.6.2 TLS client authentication with TPMs

In common client authentication scenarios, the client credentials allowing access, for example to a company's VPN, are stored on the platform's disk where they are subject to manipulation or theft. The protection of these credentials solely relies on the security services and protection mechanisms provided by the platform. Having a security element like a TPM on the platform, it seems reasonable to use the protection mechanism provided by TPMs. A simple approach could be to use the bind mechanism of the TPM to bind the credentials to a certain platform state. However, that does not fully guarantee the nondisclosure of these credentials as they are decrypted and used in plain for the authentication process - TPMs provide shielded locations and protected capabilities which can provide a much better protection. Consequently, it is reasonable to store and use the credentials inside the TPM.

Basically, client authentication in TLS works as follows: During the handshake, the server requests a certificate from the client. The client sends a *Certificate Verify* message including the signature on the hash of the handshake messages sent so far. The server is then able to verify the signature with the previously sent certificate and by comparing the hash value of the client messages recorded by the client with the hash value of the messages received by the client, thereby authenticating the client.

Instead of creating the signature within the TLS stack, the signature could be created inside the TPM, therewith providing a strong protection of the signature key. Although this method can also be achieved with common smart cards, the TPM and its DRM capabilities offer features for managing these credentials, which is hard to achieve with smart cards. Assuming a running trusted platform, network administrators could remotely distribute or delete the client credentials among their managed platforms. Furthermore, the use of the client credentials could be limited to a certain amount of uses simply by locking them to a monotonic counter. Moreover, authorization to use the credentials could be bound to a certain platform configuration, thereby preventing a successful connection with invalid configured platforms.

A proof-of-concept implementation of the client authentication commands for TPMs is discussed in the following paragraph.

Extension to TPM authentication

In some scenarios, the TPM has to authenticate itself to a service before using the specific service. Prior the the authentication, however, the TPM has to be registered as, for example in the DAA case, a valid member which is allowed to join a group and receive DAA credentials. In the DAA scenario, parameters for the DAA protocol are encrypted via the public-EKnd decrypted inside the TPM for further use. As a result, the source or sender of the parameters can be sure that only the addressed TPM can decrypt the provided information thereby authenticating the TPM via its public-EK. A similar approach can be used to improve TLS authentication. When recalling the TLS client authentication mechanisms, one can see that the server identifies the client according to its client certificate. The private part of this certificate is used to sign the incoming and outgoing messages that are sent during the handshake. The signing-key itself may be stored inside a TPM or another security device. However, this approach requires two sets of certificates namely the EK certificate and the signing certificate (and of course all certificates required to validate these certificates). A more efficient approach would involve only one of these certificates and the corresponding certificate hierarchies. As the EK-certificate is already available on the TPM it is reasonable to go for this certificate.

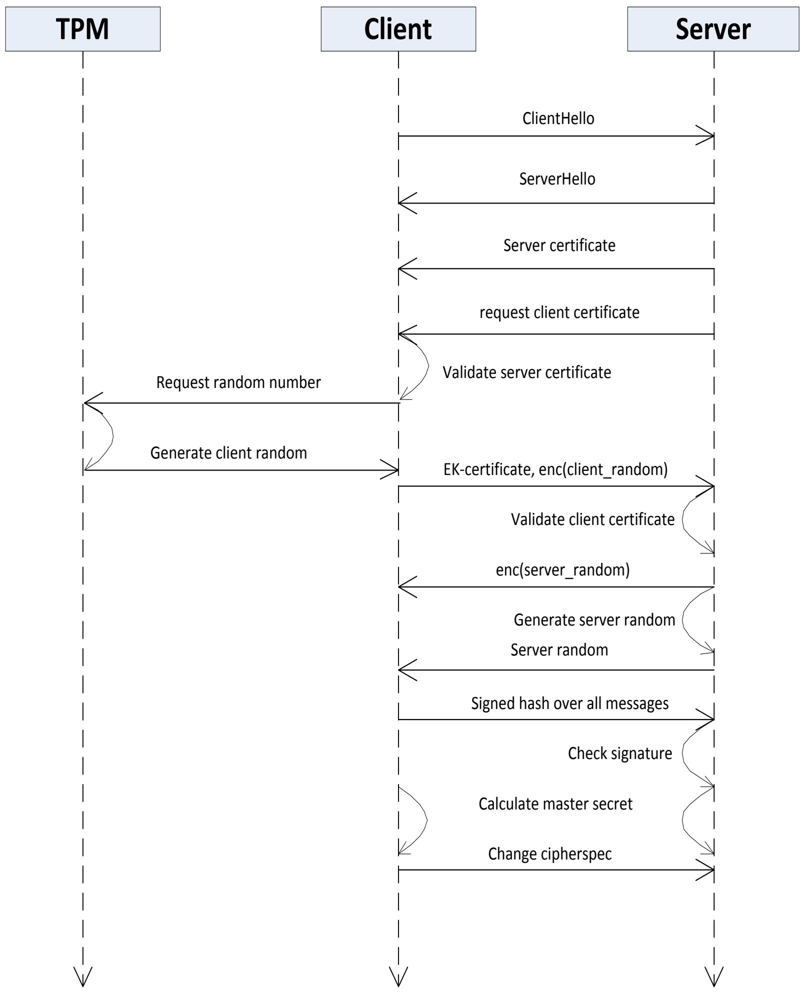
In order to involve the EK-certificate and the decryption operation of the EK, the client authentication mechanisms must be modified in the following manner:
Instead of computing a signature on the handshake messages to authenticate the client certificate, the client and the server each compute a part of the random pre-master secret[6]. The client submits its part encrypted via the server public-key as defined in the original specification. However, the server encrypts his part of the pre-master secret with the public-EK of the client. Both exchange the encrypted values and assemble the pre-master secret $pms = H(pms_{server} \| pms_{client})$. The further computations of the master secret and eventually the cryptographic keys are unchanged, except the handshake phase which is modified. The client does now not require to authenticate itself explicitly as the authentication was already established via the encrypted pms value. The authentication is established by the fact that only the TPM which is in possession of the private-key that corresponds to the public-kec encoded in the EK-certificate is able to decrypt the pms

[6]The signature on the messages is still generated as it is used to check the integrity of the messages exchanged during the handshake phase.

from the server and further compute the correct pre-master secret and session keys. If a malicious host sends an EK-credential from another host, the encrypted *pms* can not be decrypted by the TPM. Additionally, a fraud certificate would never be accepted by the server as these certificates are issued by the TPM (see discussion in Subsection 2.6.2. The fact that both partners contribute to the master secret computation lowers the risk of weak random numbers if one of the partners has a manipulated or malfunctioning random number generator. This approach may also be extended on the server side. An alternative authentication model could include a TPM on the server side. In this case, the certificate sent by the server is the EK-credential. Instead of decrypting the client random by the (server) host, it is decrypted by the (server) TPM. Hence, both sides - client and server - can take advantage of the benefits of a secure computing environment.

Figure 2.12 displays the modified handshake protocol:

Figure 2.12 TLS authentication via Endorsement Key



TLS Client Authentication Commands

For the TPM supported TLS client authentication prototype, the following commands were added to the TPM emulator:

- TPM_LoadClientAuthKey(TPM_KEY_HANDLE parent key handle, TPM_KEY tls_aut_key). This method loads the specified key into the TPM and returns a TPM_KEY_HANDLE structure.
- TPM_UpdateClientMsg(UINT32 client message size, BYTE[] client message). This command sends a single hash of a handshake message to the TPM. Consequently, this command is subsequently invoked for every message sent by the client.
- TPM_SignClientMsg(TPM_KEY_HANDLE tls_auth_key_handle). This command instructs the TPM to compute the signature over the handshake messages with the given key addressed by the key handle.

In order to use these commands, the TLS stack has to establish an authenticated session with the TPM. Furthermore, this approach does not require any modifications of the verification module of the TLS stack which is responsible for verifying the signature.

The benefit of the approach is the reduction of the number of certificates involved for establishing the session. Validating a public-key certificate is bound to a lot of effort in terms of certificate management, memory consumption and requirements on time for searching, downloading and validating certificates and revocation lists. Hence, reducing the numbers of certificates to validate is highly desirable.

A note on Endorsement Certificates In the previously discussed scenarios, it is assumed that TPMs are equipped with an endorsement credential. However, in practice this is not always the case. In reality only the TPMs from Infineon Technologies are shipped with such a credential. This fact is problematic as during communications with previously unknown trusted platforms or platforms from other certification hierarchies, it is not possible to determine the genuinity of the TPM.

The situation is even worse when taking the other certificates of a trusted platform into account.

Relation to Mobile TPMs The concept of advanced configuration change reporting is also applicable to mobile TPMs. The major difference between common TPMs and mobile trusted modules (MTMs) is that, depending on the security features provided by the mobile platform, MTMs can be implemented only in software [35]. This property makes them an ideal basis for this approach as the software based TPM implementations are easy to adapt.

2.7 Conclusion and Future Work

This chapter provides a multitude of contributions. First, a detailed analysis of the remote attestation process in combination with secure channel technology is provided. The analysis includes the investigation of the impact of attestation requests on the data throughput of TLS channels and demonstrates the performance decrease when hardware and software-based TPMs are used.

Second, based on the results of the performance analysis, a new attestation model is proposed that allows a consistent reporting of PCR events without the issues of the original approach. In the new design, the platform does not passively wait for requests to come. Rather it actively reports configuration changes by integrating them into the TLS data stream. In contrast to other schemes that report changes through the channel, this scheme additionally provides a reliable and secure proof that a change happened and that it was correctly recorded and processed in a genuine TPM.

Third, modifications of the new model are discussed allowing a more efficient use on specific platforms. The modifications allow secure processing of TLS channel credentials inside TPMs and greatly add to including the endpoints of a secure connection into consideration when investigating security aspects of trusted channels.

The newly presented method allows reporting of platform configuration changes in a reliable way. The proposal also shows how the mechanism for protecting TLS records can be used to provide a proof for configuration changes by calculating the MAC for protecting TLS records inside the TPM. The benefit from this method are different signature values on the TLS records depending on the configuration. These different signature values allow a remote platform to detect changes in the local platform configuration. Basically, configuration changes are encoded in the cryptographic keys that are responsible for computing the MAC values.

To back the proposed model and the corresponding hypothesis a proof of concept prototype was implemented to demonstrate the advantages of the new model. The TPM specification does not include the required features for this approach. Consequently, suitable TPM implementations had to be found and modified and additional commands had to be added. The implementation of choice was a pure software implementation, hence, the proof-of-concept implementation is currently only available as software emulation.

Nevertheless, it provides a valuable addition to all software based TPM implementations. Such implementations include virtual TPMs, like the one used in virtualization techniques like XEN [116] or para-virtualization technologies and mobile TPMs [111].

As, virtualization is more and more common, software based TPMs are widely used on server systems in the form of virtual TPMs. Virtualization and para-virtualization technologies take advantage of multiple software TPMs instead of relying on one singe hardware TPM. As a consequence, server systems are an ideal platform for the new approach.

The analysis of the new approach also showed that no additional cryptographic algorithms are required if the set of allowed cipher suites is limited to commonly used algorithms e.g. SHA-1. This limitation allows an efficient use of the constrained resources of TPMs.

In contrast to other published approaches (see Section 2.1.1), this model allows to bind a TLS channel to a TPM respectively to a trusted platform.

To enhance the proposed model even more, modifications to the key derivation process are discussed in this chapter. In contrast to the original approach which is rather easy to integrate into common TPMs, the second one is more complex but shows a better runtime performance. The performance advantage stems from an alternate key derivation algorithm that reduces the communication overhead between CPU and TPM. Although the new design is faster, is comes at the cost of storing more information in the TPM.

Due to the fact that authentication keys are created and used inside a tamper-resistant TPM, the security of the trusted channel is even more increased. As the keys do not leave the TPM they are not available in plain on the platform.

Moreover, the proposed approach assumes that the secure channel and the configuration is bound to a certain state that is changed if any kind of software is loaded. This causes either that the channel is closed or new session parameters are negotiated - even if the loaded software is a trusted one. This issue was solved by introducing a new TLS record-layer message. This new message allows to use an open connection without closing and re-establishing the channel when PCR changes occur.

Moreover, a method was proposed how a TPM could be used for TLS client authentication, thereby providing stronger protection of the client credentials as similar software implementations and without the requirement of additional hardware like smart cards.

Furthermore, the detailed analysis shows that the proposed approach only marginally affects the data throughput of a TLS channel - in contrast to the original approach.

Third, detailed information of how the existing TPM specification should be updated in order to support the proposed scheme are provided. In addition, a proof-of-concept implementation for generating reference measurement values was implemented.

2.7.1 Future work

The reference design of the proposed approach focuses on secure socket layers (SSL) and TLS channels. Further research may include adaption of the proposed model to other secure channel technologies like IPSec.
A further aspect that is of interest is anonymity protection. In the proposed design, PCR values are sent in plain in order to prove the platform's configuration. Although this information does not reveal the actual identity of a platform, it could be used to track platforms by means of configuration fingerprinting. Therefore, investigations how to integrate anonymity protection in the new scheme is of interest.
The new approach is also of interest for research areas like information-flow security. The property of this approach that allows data to be tagged according to the configuration of the processing platform, allows to ascertain trust estimations about the actual data.
Another interesting field of research is the application of the new approach to different modes of cipher operations. Tests were conducted with the RC4 and AES cipher in CBC mode, however, other interesting modes of operation exist. The Galois/Counter Mode (GCM) [88] supports enhanced crypto algorithms allowing to compute encryption and authentication information in a single step in contrast to common systems that compute the MAC and the encrypt the data block resulting from data payload and MAC. Moreover, the application of the new scheme in the TLS 1.2 standard [107] that supports hash algorithm agility and therewith the usage of different key derivation algorithms is of interest for further research.
The general impact and possible applications of this approach on existing virtualization techniques like XEN [116] are also subject to further research. Moreover, the proof-of-concept implementation and the proposed design are valuable additions for mobile TPMs [111] which may be implemented exclusively in software in some environments.

The current prototype only supports the creation of authentication keys inside the TPM. Future versions could include the installation of externally created authentication credentials. Moreover, a more sophisticated authentication management than the existing one could be integrated.

Chapter 3

Architectures for Trusted Computing Enhanced Mobile Platforms

3.1 Introduction

Nowadays, trusted platform modules are available for nearly every PC platform, ranging from desktop machines to notebooks. These TPMs provide the basic building blocks for Trusted Computing enabled services like authenticated boot or authenticated reporting of integrity values. Yet, these platforms are not the only ones that can host TPMs. Mobile and embedded platforms, like cell phones, can also host TPMs but may have different requirements and different use-case scenarios.

Common desktop TPMs are produced in high numbers which allows TPM manufacturers to keep the prices low. Unfortunately, these common TPMs are deprecated for mobile and embedded applications. From a certain point of view, it seems simple to put a micro controller based TPM like the ones used on desktop machines on a mobile platform. However, each new chip on a phone's mother board increases the cost of this device, not to mention the additional power consumption of the extra chip. Consequently, it is reasonable to search for alternatives - alternatives, for example that are primarily based on features and mechanisms that embedded devices already carry as part of their base configuration.

In order to keep the costs for mobile TPMs low, the TPMs might also be implemented only in software, raising questions about the security assumptions for these kinds of TPMs and the platforms they are used with.

A specialized working group within the TCG is dedicated to defining, extending and maintaining the specifications concerning mobile trusted computing. This group has published a specification that defines how mobile TPMs could be designed and which features they could provide [52]. Intentionally, this specification is written in a rather relaxed style which allows manufacturer to implement their mobile TPMs in different ways. In particular, concrete instructions how to implement a TPM on a mobile device are not provided.

Unfortunately, this relaxed way of defining the mobile TPM specification bears the problem of lacking concrete guidelines for implementers. In contrast to common TPMs, TPMs for mobile platforms do not need to be implemented as micro controllers, leading to different security assumptions. Therefore, two questions arise when designing mobile TPMs. First, how can MTMs be designed in an secure and efficient way by employing embedded security system solutions? And second, how can the security assumption of these MTMs be compared to those of TPMs? Without having clear definitions of conformance and compliance of mobile TPMs, these questions are hard to answer. Nevertheless, both questions are discussed in this chapter.

Before deploying any security related service that relies on a protected execution environment, such as creating a signature or the direct anonymous attestation protocol, the question of how trusted computing technologies - in detail - mobile trusted modules can be securely and efficiently be implemented, has to be answered. A secure basis for hosting cryptographic functionality has to be found, that does not only meet the requirements of state-of-the art cryptographic algorithms, but also allows to fulfill the standards of high-level security evaluations. Therefore, concepts for enhancing mobile and embedded devices with trusted computing (TC) technology are discussed in this chapter. In particular, two designs for providing TC building blocks on embedded devices are discussed and compared. Both building blocks rely on security mechanisms already provided by the embedded devices.

The first approach focuses on a software emulated mobile TPM that uses processor extensions to achieve protection from access by arbitrary applications. A discussion of this question is especially important as at the moment it is not clear whether the installed security features provide a sufficient level of security and which level can actually be achieved. This approach is based on the ARM TrustZone, which is basically a processor extension that separates the device into secure and non-secure domain (see Section 3.1.2).

The second approach (see Section 3.2), makes use of onboard smart cards. Many new mobile phones are equipped with an additional smart card (besides the SIM card) which can store and operate secret data like keys or certificates. These smart cards, or secure elements (SE) in the sense of the TCG, can be addressed by the mobile phone as well as from external devices via near-field communication which offers new perspectives of secure device communication. Both approaches aim to be as close to the TCG's published mobile TPM specification as possible.

Special focus is also laid on resource conserving designs of mobile TPMs. As memory is limited in TPMs, efficient use of it is one of the major design goals. Hence, approaches how to preserve and economical use of TPM resources are also part of this chapter.

This Chapter is separated into 5 Sections. In Section 3.1, an overview and explanation of the differences of the two kinds of mobile TPMs i.e. mobile-remote-owner-trusted-modules (MRTMs) and mobile-local-owner-trusted-modules (MLTM) is given. Furthermore, related background information on ARMs TrustZone architecture are provided. In Section 3.2, a mobile Trusted Computing reference architecture is proposed. A comparison of this new mobile TPM design and the ARM TrustZone approach is given in Section 3.3, followed by Section 3.4 which deals with efficient architectures that employ dynamic function sets for mobile trusted modules. Finally, Section 3.5 briefly concludes the chapter.

3.1.1 Related work

Different approaches how to implement MTMs are pursued by various research groups. The most important ones are introduced in the following paragraphs:

In [118], an idea to extend a SELinux based kernel in order to provide a generic domain isolation at the kernel level is proposed by Xinwen Zhang, Onur Aciicmez and Jean-Pierre Seifert. With this design, the research group proposes a strong and convenient mechanism to satisfy the security requirements of trusted mobile phones on isolation of engines and integrity assurance.

One of the leading mobile phone manufacturers - NOKIA - also conducts research on implementation aspects of MTMs. Jan-Erik Ekberg and Markku Kylänpää published a paper [65] that provides an introduction into the concept of MTMs from the device manufacturer's point of view. Furthermore, their work presents an implementation of an MTM that is based on the well-known TPM emulator from Mario Strasser [96].

Also from Ekberg et al. [39, 68] a proposal for a mechanism for securely executing small programs in trusted environments is introduced. Their approach is designed around a size constrained secure execution environment present on their target processor and platform architecture. Inside this secure environment, small security critical programs can be executed on a trusted byte-code interpreter. The byte-code interpreter used in [39] is built upon a simplified version of the Lua 4.0[1] virtual machine. In order to provide secure storage, the approach in [39] resorts to cryptographic methods. Based on secure storage facilities, the authors of [39] propose mechanisms for securely swapping programs and data in and out of their secure environment.

A similar approach to small trusted execution environments is presented by Costan et al. in [24]. The authors of [24] describe a "Trusted Execution Module" (TEM) which trustworthily executes secure closures. The TEM itself is a byte-code interpreter for a small special-purpose programming language. This interpreter is realized as a JavaCard applet, hosted inside a JavaCard enabled smart-card. Costan et al. propose to equip the TEM with a public key similar to a TPMs endorsement key. In the case of the TEM, this key is used to protect secret data of the closures submitted to the TEM for execution. Similar to [39], secure storage is realized outside the smart-card, using cryptographic methods.

The main difference between Costan's [24] and Ekkberg's approach is the location where trusted computations are executed on the platform.

The authors of [40] investigate the idea of a programmable TPM designed around a normal TPM with secure binding to a programmable smart-card. In their paper, they discuss a number of potential enhancements to sealing and binding, which can be implemented with their architecture. Furthermore the authors of [40] elaborate on count limited objects, as an alternative solution to the monotonic counters available in a normal TPMs. According to the authors, the architecture described in [40] has several limitations, caused by the fact that the TPM and the smart-card are separate devices. Particularly, the types of functionality which can be implemented with that architecture are limited by the TPM's policies.

In contrast to the approaches discussed in these publications discussed in the previous paragraphs, the concept focused on in this thesis uses security hardware, provided by mobile devices themselves to host MTM functionality. Such hardware is, for example, the SIM card every mobile is equipped with.

Schellekens et. al. [70] propose to dis-embed the TPM's firmware and store it on the host platform. They aim at reducing the complexity of current TPM implementation, allowing simplified hardware architectures for TPMs by introducing the concept of μTPMs. This approach is similar to the idea proposed in Section 3.4. However, Scheleken's ap-

[1] see `http://www.lua.org/`

proach focuses on monolithic blocks of firmware (of TPMs) meaning that changes of the functionality of a TPM affect the entire TPM or sets of features. In contrast, the proposed approach allows to modify specific commands of a MTM. Moreover, the concept allows to modify the trusted software stack in order to synchronize the command set of the MTM with the command set of the software stack by employing newly added security features.

3.1.2 Background

The core of every trusted platform is the *trusted platform module* (TPM). In case of mobile or embedded platforms, we speak of a *mobile trusted module* (MTM). Although an MTM has similar features as a common TPM, there are some differences, as characterized in [111].

The Mobile Trusted Module specification defines two types of MTMs: the *Mobile Remote-owner Trusted Module* (MRTM) and the *Mobile Local-owner Trusted Module* (MLTM)[2]. The difference between these two types of MTMs is that the MRTM must support mobile-specific commands defined in the MTM specification as well as a subset of the TPM v1.2 commands, whereas the MLTM only supports a subset of the TPM 1.2 commands [111]. Typically, phone manufactures and network service providers use an MRTM. These parties only have remote access to the MTM whereas the MLTM is used by the user who has physical access to the device and its applications. The first one is used as a secure entry point to the phone in order to perform updates etc. whereas the second one is used like a common TPM.

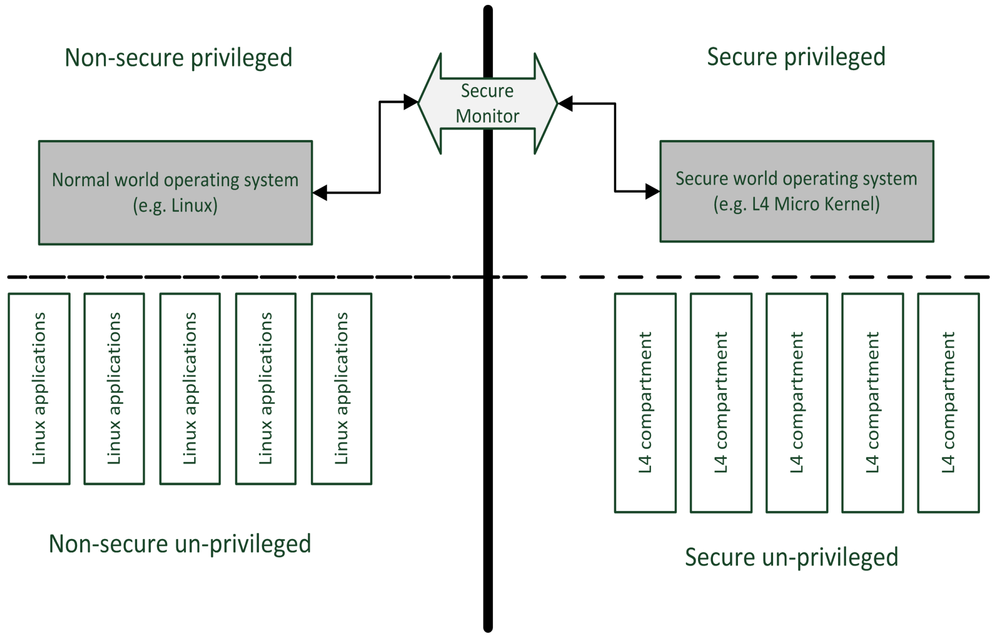
The different parties - called *stakeholders* - have different requirements on the MTM. Depending on the stakeholder, the MTMs are applied in areas such as platform integrity, device authentication, SIMLock/device personalization, secure software download, mobile ticketing and payment, user data protection and privacy, and more [115]. How these different kinds of MTMs are implemented is not defined. A discussion of how a possible realization could look like is given in section 3.2.

Another possible way for designing an MTM is as software module. The TrustZone security extension [8] offers an ideal basis for such an approach. The very concept of ARM TrustZone is the introduction of a *secure world* and a *non-secure world* operating mode on ARM11 and Cortex-Ax based processor cores. The split into a secure world and a non-secure world mode can be seen as extension to the privileged/unprivileged mode split that can be found on pre-TrustZone ARM cores. The design allows a secure/non-secure world boundary to create two separate domains. Unlike in MicroKernel architectures, the separation is hardware enforced meaning that for each domain, an extra set of CPU registers exists. From a general point of view, the TrustZone design implements two independent, strongly isolated worlds with a well defined strictly controlled interface in between. Isolation between these two worlds is provided by the protection features of the TrustZone processor extensions. Hence, in each of the domains, a different operating system is running. In the non-secure world, a standard operating system or *RichOS* like Windows, Linux or Android is executed, whereas a more specialized one is used for the secure world.

Figure 3.1 gives an overview of the TrustZone architecture. The communication between the two worlds is established via a special Secure Monitor Mode together with a Secure Monitor Call instruction that allows to exchange data between the two domains. Interrupts can be handled in a secure and deterministic way on TrustZone cores. Apart

[2]The term MTM refers to both, the MRTM and the MLTM

Figure 3.1 TrustZone Architecture Overview



from the extensions to the processor core itself, the SoC buses in TrustZone enabled systems carry extra signals to indicate the originating world for any bus cycles. Whenever a bus cycle is started by the core, the secure/non-secure world state is recorded and encoded into these extra signals. SoC peripherals can interpret the TrustZone extra signals to implement a low-level access control based on the secure/non-secure world distinction.

On the secure world side, a specialized stripped down Linux kernel is used to provide the necessary small runtime environment for security and safety critical tasks and for the components required to handle the non-secure world side. The secure world environment can be stripped down to the bare minimum of software components. In typical systems, this is a high security OS kernel which is in addition evaluated according to specific security and safety requirements [6].

The hardware supported boundary to the non-secure world environment is used to provide a sufficient protective shielding against any potentially malicious piece of code running on the non-secure side. There is no direct way for non-secure world code to access secure world data or memory areas without explicit permissions.

In addition, TrustZone provides a small amount - depending on the vendor - of non-volatile memory. This memory may be used to store secret material. The material may only be accessed by applications executed inside the TrustZone environment.

3.2 An Integrated Architecture for Trusted Computing enabled Embedded Devices

In this Section, an overall discussion of an MTM design based on secure elements is given. The proposed architecture describes one approach for defining a mobile TPM solution for embedded systems based on security mechanisms provided by the platform. Moreover, it demonstrates the interaction between the different components of the design especially of the MTM with the mobile device and its applications.

Moreover, a proof-of-concept implementation was made for gathering and analyzing experimental data and for defining a reference design for mobile trusted modules. The PoC consists of an MTM reference implementation, a command library that provides TCG compatible commands to applications and a communication interface module that handles communication between the command library and the MTM. Based on these experimental results, a discussion is provided of the feasibility and the system requirements of this approach with respect to state-of-the art mobile equipment. The details of the proposed architecture are discussed in the following section.

3.2.1 Architecture overview

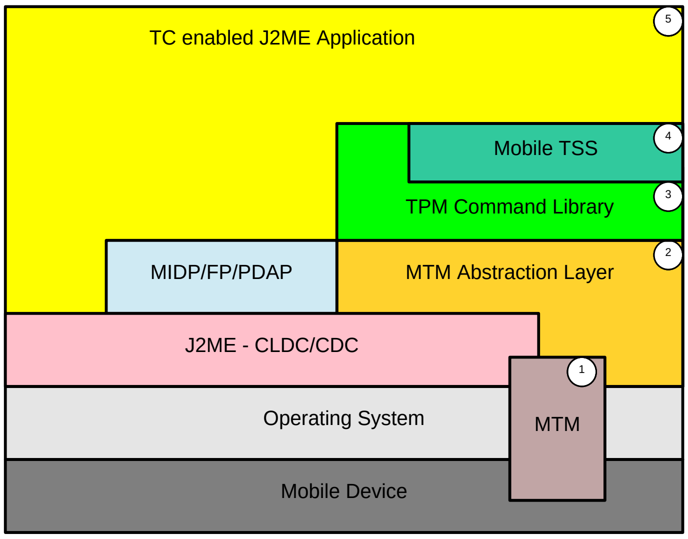
Various kinds of mobile Java platforms depending on the phones' capabilities exist. Nevertheless, the core component of all these Java platforms is the *Java 2 MicroEdition (J2ME)* [100] specification - published by SUN - that was especially designed for resource constraint devices. The specification includes two core architectures: the J2ME *Connected, Limited Device Configuration* (CLDC) [99] and the J2ME *Connected Device Configuration* (CDC) [3]. An overview of the differences between these two configurations is given in [29]. The CLDC's target devices are very resource constrained mobile phones, whereas the CDC is intended to run on more sophisticated devices like smart phones or PDAs. Nevertheless, the most widespread configuration nowadays is the CLDC allowing to address a much broader range of devices. Furthermore, the security system is more enhanced and much more restrictive than the one of the CDC [77]. The architecture proposed in this Section addresses both, the CDC and CLDC platform.

As outlined in the previous paragraph, the concept discussed in this Section focuses on Trusted Computing (TC) services for Java enabled phone platforms. Therefore, most of the components outlined here are designed to be implemented in Java. However, there are a variety of functions that cannot be addressed by Java . For this reason, features of TC outside of the scope of the proposed architecture are discussed in the following lines. Such features are e.g. *secure boot* and *authenticated boot.* It is assumed that the addressed target platforms are already equipped with a trusted boot-loader and a trusted operating system. Moreover, the measurement processes from the boot of the machine up to the measurement of the Java virtual machine (JVM) is supposed to be performed prior to the start-up of the JVM.

Figure 3.2 shows the different modules that are discussed in greater depth in the following sub-sections.

Number (5) in Figure 3.2 shows the TC enabled application that makes use of TC services. Such services might be *remote attestation* (i.e. integrity reporting and integrity verification), or *sealing* and *binding.* The next component (No. (4)) is the TSS library. The library consists of a provider for cryptographic operations and a provider for TC services. No. (3) is the command library that is primarily used by the TSS to send

Figure 3.2 Architecture Overview



and receive commands from the MTM. Nevertheless, an application might require to access the MTM directly, so the low-level command library also offers an API to the top level. Communication between the command library and the MTM is carried out by the abstraction layer (No. (2)).

The concept focuses on the user and user installable applications. Therefore, the used MTM (Figure 3.2 No. (1)) has to provide the functionality of a *MLTM*. A more detailed discussion of each layer component (1)(2)(3)(4) is given in the following sections.

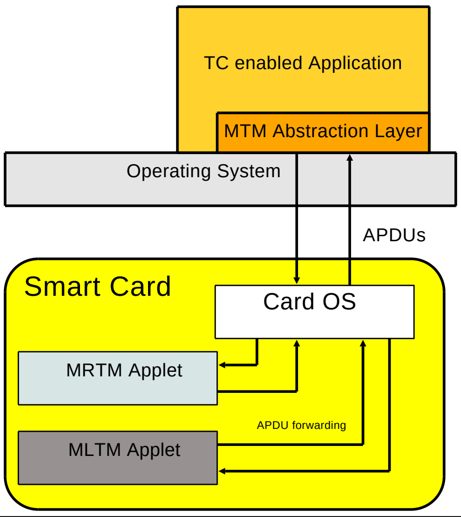
3.2.2 The mobile trusted module

When investigating mobile phones and other embedded systems, the question how to implement an MTM (see Figure arises 3.2, number (1)). The MTM specification [111] does not answer this question - it only describes the required features for compatible MTM implementations in order to give the designers more flexibility for their implementations. When thinking about implementing an MTM on a mobile phone the fact that every extra piece of hardware raises the cost for the whole device has to be considered. For this reason, using already existing hard- and software components in a mobile phone sounds reasonable. One way could be to implement the MTM in software. However, such a solution would require that the underlying operating system or hardware provides process isolation or the typical sandbox model protecting the MTM application and providing shielded locations and protected capabilities as required by the specification. Nevertheless, a detailed investigation of the isolation mechanisms is required in both cases.

Another option involving a special dedicated silicon chip like on desktop systems would provide *protected capabilities* and *shielded locations* as requested for TPMs, however, extra hardware components occupy space within the device and drain power from the battery.

At first glance, the most logical piece of hardware in a mobile phone to host a TPM is undoubtedly the SIM (resp. USIM) card. The Subscriber Identity Module (SIM) is an integral component of a mobile phone. It provides identification of the user as well as support for cryptographic operations. Moreover, SIM cards meet the requirements for *shielded locations* and *protected capabilities*. Being equipped with these features, the SIM card appears to be a good basis for providing MTM functionality. However, the SIM card

Figure 3.3 Software architecture for a Smart Card based trusted mobile platform



is not fixed to the device. A user can easily remove the card.

This is also true for any other removable security device such as SD-cards. Although they provide a high level of security and are used in many security related applications, they can hardly be applied to host MTM functionality.

Another drawback of using the SIM card are the stakeholder requirements. SIM cards are under the sole control of the network providers. Other stakeholders are hardly allowed to install and execute arbitrary applications on these cards. Fortunately, other devices are equipped with on-board smart cards (e.g. Nokia 6131 NFC) which were used in the proposed approach to host the MTM [30] as they provide the required access mechanisms without restrictions. In contrast to a SIM card, the onboard secure element cannot be removed, from the device. All data that is stored on the secure element, remains on the platform and cannot be transferred to another device by just moving the card.

This fact has interesting consequences. On the one hand, all data that is bound to a removable SIM card cannot be unbound unless the correct card is reinserted again. On the other hand, data that is stored inside or bound to a secure element cannot be transferred to a new device. Moreover, if the data is fixed to a certain platform configuration, the new device could be forced to have a specific configuration in order to get access to the stored data.

As a consequence, the proposed architecture focuses on devices that are fixed to the embedded platform like secure elements or secure CPU core extensions [9]. In contrast to SEs, CPU core extensions are part of the CPU. They provide a secure execution environment inside the CPU. Compared to TrustZone, they are not just a CPU extension a separated controller providing a smart card like environment for applications.

JavaCards can host different applications, called applets [101]. Typical applications use such applets to store and manage authorized access e.g. digital purses or authentication credentials. However, the card can also be used to host applets with MTM functionality.

Figure 3.3 shows the basic design of the prototype. The smart card (or secure element) hosts the MTM. The MTM itself is implemented as a Java Card applet that supports the processing of TCG compatible commands [112]. Common TPM functionality also includes cryptographic operations, e.g. the current MTM specification defines RSA operations to be used for asymmetric cryptography. Typically, smart cards are equipped with hardware based cryptographic support. This hardware support enables smart card applications to perform fast cryptographic operations. Fortunately, this support is also available for Java Card applets which enables them to support all required operations [52].

The communication between the host application and the smart card is done by TCG conformant commands that are split to fit into application programming data units (APDUs) as defined in [62]. However, current available cards are subject to strong restrictions concerning available memory (EEPROM and RAM). The free available memory on the reference prototype is about 76 kilo bytes of nonvolatile memory and 8 kilo bytes of RAM. Furthermore, the processing of Java byte code is rather slow, which demands efficient command handling and parsing.

Many cards can host more than one applet. This fact allows the installation of multiple MTM applets and therefore allows to install an MRTM and an MLTM on the same card, providing functionality of both kinds of MTMs.

A major advantage when using smart cards for hosting MTM functionality is that all security properties of a certain smart card can be reused when assessing the security level of the smart card based TPM. Smart cards are well investigated in the sense of security evaluations - various security evaluations and protection profiles [98] exist.

3.2.3 The MTM abstraction layer

An MTM can be implemented in a number of ways, as the TCG only defines requirements and features for the MTM but does not regulate how to implement it. Depending on the implementation and vendor, the communication with the MTM can be realized in many different ways. Hence, the proposed architecture includes an interface called *MTM abstraction layer* as shown in Figure 3.2, No. (2).

Similar to the TCG Device Driver Library (TDDL), the abstraction layer provides a common interface for the command library to the underlying MTM implementation. That means that the abstraction layer receives the encoded commands from the command library and converts them into a format required by the TPM. Furthermore, the abstraction layer handles the sending and receiving of these MTM commands. The implementation of this layer is of course vendor and MTM specific and has to be adapted for any different kind of MTM. Therefore, depending on the type of MTM, the abstraction layer might also require access to the operating system and native libraries.

3.2.4 The MTM command library

An MTM supports a variety of commands and data structures as defined in [111]. Many of these commands stem from the original TPM1.2 specification [53, 112]. The command library provides the MTM related commands and authentication protocols an an API for applications. However, depending on the intended purpose of the MTM, it might not be necessary to implement all commands. Considering a set of commonly used services, it could be reasonable to define a subset of commands required for particular operations.

3.2.5 The mobile trusted software stack

TC services are primarily provided for the applications by the trusted software stack (TSS). The TSS's main components are the TSS Core Service, a crypto service provider and a TSS service provider [113]. All these services and providers can also be part of the mobile TSS.

However, a TSS implementation might become very large so a discussion of specifying a slim mobile TSS might be of interest.

3.2.6 Deployment of the MTM

As the focus of the concept lies on software for mobile devices, the question arises how that software required by the architecture could actually be installed on the device. The term *software* refers to the mobile TSS, the command library and the abstraction layer. In the rest of the Section, these components are referred to as *TC software* or *TC software components*. The application may be installed over-the-air (OTA) via wireless link or could be pre-installed by the network provider. Therefore, a discussion of how these components can be efficiently and securely delivered to the mobile device is given in the following paragraphs.

Why is the question how the components can be installed so important? Although Java is seen as a highly portable development environment, mobile Java applications are developed for a specific environment. This environment consists of the virtual machine plus its' boot libraries. In addition, optional libraries can be added to the environment by the mobile equipment manufacturer. As a consequence, the developer has to know exactly which environment he can expect on the mobile device and if the TC software is already installed on the platform or if it has to be delivered with the application.

Current J2ME implementations assume that all required libraries are installed and uninstalled along with their owning application. This fact forces the developer of the application to include all software components. Consequently, if an application wants to use this architecture, it has to include: the low-level command library, the TSS and the abstraction layer. However, a TSS implementation in Java can have a size of 500 kilo bytes or more [66] For this reason, it would be reasonable to find a solution to share code among applications. This problem could be solved by the upcoming J2ME/MIDP 3.0 specification [85]. This specification allows mobile Java applications to share code in form of libraries. Other options could be to integrate a TSS into the device platform allowing the Java virtual machine access to it or shipping the stack with the JVM as an a optional package in form of a Java library. A mobile TSS API could then provide access for applications to the MTM for both solutions.

Three ways how to provide the TC software components can be identified:

- The TC components are installed along with the J2ME application. Application and TC components are installed together in one package. The drawback here is that all applications have to include the TC components. Remember, a TSS implementation might become very large so sharing the TSS's code is mandatory. On the contrary, a developer might only include components that exactly match the requirements of the application and remove unused TC features and services.

- The TC components are included within the Java environment. This could be realized by either integrating the software within the Java base classes or by adding an *optional* package. The idea would include moving the TSS and/or the low level commands into the Java runtime environment. This step has the advantage that the stack and its components do not have to be installed together with the application as previously discussed. However, for integrating the software into the base classes, the J2ME platform specification would have to be redefined, which is very unlikely to happen.

- The TC components are installed on the platform and come as part of the operating system. In this solution the TSS runs as a service. Consequently, the Java runtime environment has to provide an interface to the application.

All these solutions have in common that no standardized API exists at the moment. Therefore, defining an API is a first logical step.

3.2.7 Design and implementation of the prototype

The prototype implementation of the architecture covers the ownership operations (TPM_TakeOwnership etc.), the authorization protocols OIAP and OSAP and the endorsement key (EK) operations for creating and reading the EK.

The generation and parsing of the MTM commands are implemented on both sides - on the MTM as well as on the J2ME platform. Optional commands are omitted whereas commands for both, for the MRTM and the MLTM, have been implemented in the same MTM prototype for testing purposes. Despite the already implemented features and commands, the primary objective of the implementation is support for *attestation* and *secure boot*.

Figure 3.4 Prototype Design Overview

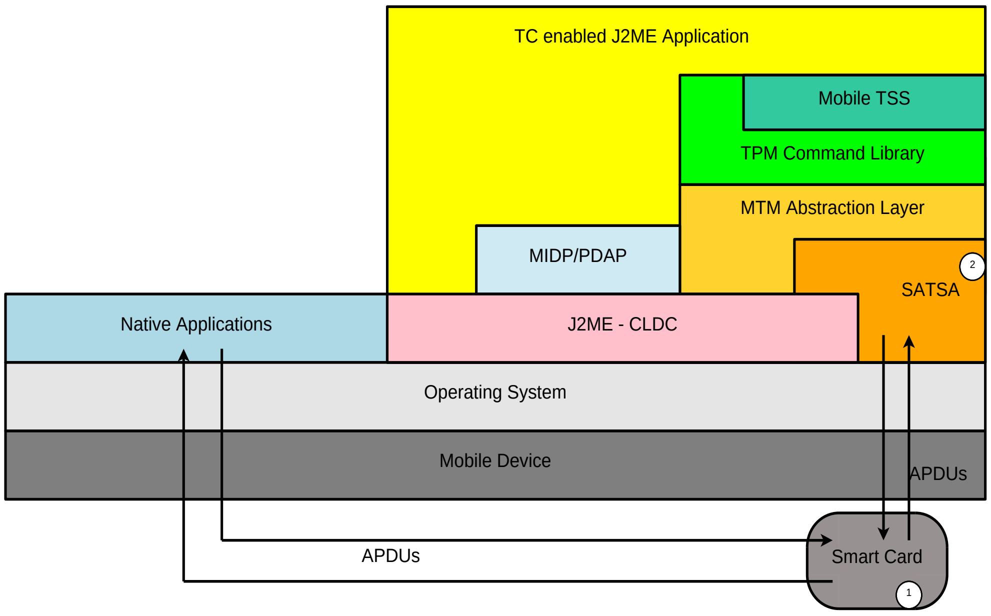


Figure 3.4 shows the prototype implementation of the architecture. The low level commands in the library are implemented in an object oriented fashion, meaning that every command is represented as a Java object. Each of these objects is able to generate *incoming* MTM commands - these are the commands that are sent to the MTM - as well as to parse *outgoing* commands[3]. Although implementations for generating and parsing TPM commands in Java already exist, [90] [113] none of them was designed to meet the requirements of the J2ME/CLDC platform. The prototype closes this gap and provides a J2ME/CLDC compatible command library.

[3]Incoming and outgoing is meant from the point of view of the TPM chip

3.2.8 The mobile trusted module on a secure element

The MTM in this prototype is implemented as a *JavaCard applet* (see Figure 3.4 number (1)). This applet can be installed on a SIM or a on a device internal smart card.

An important question for TPM vendors is the *byte order* of the host system. Commands sent to the TPM have to be sent in the according order either with most significant byte (MSB) first or with least significant byte first (LSB). The same applies to the single bits in the bytes. If the byte order of the host is different to the one used for the TPM, a byte conversion has to be performed. However, there is no conversion required for the proposed architecture. SIM cards shipped today are in fact JavaCards (JC) [101]. JavaCards provide a very small Java runtime environment, supporting only a subset of the standard Java API. The JavaCard runtime environment (JCRE) *is* a Java platform. This fact is especially important as all Java platforms use the *network byte order* [74]. The bytes are presented to the application in this order - no matter which order is used by the host platform. As a matter of fact, there is no byte or bit conversion required when exchanging data between card applet and a J2ME application. Moreover, a boot loader or arbitrary native device application that is able to transmit APDUs to the applet could also make use of this MTM implementation.

Nevertheless, implementing the MTM on a smart card has some drawbacks. For example, the *TPM_Init* command is not a TPM command at all but is rather a physical signal sent over the LPC bus[4] to inform the TPM [112] about a system's reboot. This function is not available on a SIM card or SE. Note that this missing feature also affects the TPM_Startup command as it is always preceded by TPM_Init.

3.2.9 Communication with the mobile trusted module

Microcontroller are usually connected and addressed via a bus or equivalent system. Access via low level programming as done with assembler or C can easily be achieved on common platforms. On the contrary, access to system components via a high-level programming language like Java is a difficult task. Java desktop or enterprise platforms offer an interface called *Java Native Interface* (JNI) [103] to access operating system functions or shared libraries of the host platform. However, this feature is not available on the J2ME/CLDC platform. The J2ME/CLDC [99] specification does not provide access to native functions for applications. Nevertheless, access can be provided by the *Security and Trust Services API* (SATSA) (Figure 3.4 No. (2)) which provides a set of Java classes that provide a standardized API for security relevant operations. Among support for cryptographic and PKI operations, SATSA provides an interface for exchanging APDU commands with the installed SIM or smart card.

As discussed in the previous chapter, the used MTM in this prototype is a JavaCard applet installed on a JavaCard compliant smart card. Communication between applications and smart cards or SIM cards is done by exchanging APDUs. Consequently, the prototype uses the APDU transmission mechanism provided by SATSA for the data exchange with the MTM applet.

The SATSA package is delivered along with the Java VM, but it is declared as an *optional* package which means, its presence depends on the mobile phone manufacturer. Hence, the usage of this interface is limited to devices supporting this API.

[4]on the desktop platform

3.2.10 Other Applications

The presented approach also has other interesting fields of application: As SIM cards are conceptually the same as common smart cards the JavaCard application implementing the TPM may not only be installed on a SIM. It can also be installed on any smart card supporting the JavaCard specification. A banking card could also be used to attest a pay terminal as proposed in [81]. In this way, the cash card could provide feedback of the status of the terminal to the user. The user could then determine whether the terminal was manipulated or not. However, this idea can not work without modification of the MTM or the smart card. The smart card must have a way to present the reported status back to the user. This could be achieved by some kind of indicator on the card like an LCD display or simply an LED. Furthermore, the MTM has to be extended with an attestation application that receives the attestation information from the terminal respectively. In this case, the MTM would not only receive commands, but would in fact receive and process attestation blobs adopting the functionality of a remote attestation application.

3.3 Software-based versus Hardware-based Mobile Trusted Modules

In this Section, the different advantages and disadvantages of the SE and TrustZone approaches are discussed. Both approaches are able to provide MTM features and functionality as defined by the TCG mobile phone specification [52]. All required commands, for MRTMs and MLTMs, can be provided.

On the one hand, software-based MTMs are a very flexible solution and can easily be adapted to certain use-case requirements. However, it is hard to determine whether a pure software or TrustZone enhanced implementation can provide *shielded locations* and *protected capabilities* that form the core requirements of TPMs as required by the TCG [54].

On the other hand, shielded locations are supported by secure elements per se. The design of current TPMs and MTMs originally stems from the designs of smart cards which makes it easy to prove that the requirement for shielded locations and protected capabilities can be achieved by secure elements. Moreover, protected capabilities can be established by implementing the required command and authorization handling services as software components on the card.

Trusted computing platforms have many other security properties especially when taking the properties of MTMs into account. Remember that an MTM may be implemented in a different way. Therefore, the set of security properties and capabilities are defined in an abstract way. For example, the mechanism how a TPM is connected to the rest of the platform may either be by physical connection in form of a data bus or by cryptographic binding of the TPM software module to the specific device.

These security properties and features of MTMs are discussed in the following Sections:

3.3.1 Roots-of-Trust

One important building block of trusted computing enabled platforms are the *roots of trust*. These roots form the basis for any higher level service. The roots are defined for desktop and mobile platforms. Their specification is abstract so that specific implementations vary and depend on the manufacturers characteristics. Not all of these roots are provided by the MTM, they can be provided by the BIOS or a piece of software that does verification operations, for example. Therefore, it is reasonable to investigate possible impacts on these roots by the proposed design and the TrustZone approach.

The following roots are defined:

- Root-of-Trust for Measurement (RTM). This functionality is provided by a runtime agent, typically a software module that is responsible to measure events on a platform an report them to a MTM.
- Root-of-Trust for Reporting (RTR). The RTR provides a secure proof of the configuration of the platform. This mechanism is realized in the remote attestation protocol by the digital signature applied on the PCR values by the MTM. Hence, this service is located in the MTM.
- Root-of-Trust for Storage (RTS). This root defines the capability to securely store configuration information which is located inside the MTM.

- Root-of-Trust for Verification (RTV). Like the RTM, the RTV is a piece of software. It is responsible to validate the integrity of the software images that are loaded on the embedded platform.

- Root-verification authority identifier (RVAI). This identifier is used to validate the creator of the boot integrity protection. In the simplest case, this identifier is a hash of the public-key of the signer of the boot loader of the platform.

It is obvious to see that the SE based approach provides a root-of-trust-for-storage (RTS) and root-of-trust-for-reporting (RTR) per se. All measurement values are stored inside a protected area and all required operations for reporting this values (i.e. quote and identity operations) can be performed in the same protected area. Therefore, it can be assumed that the card provides RTR and RTS. However, can these roots also be applied on the TrustZone approach?

Basically, all these features can be implemented inside the TrustZone execution area. Therefore, the discussion of the security of this approach can be reduced to discussion of the isolation and protection mechanisms of TrustZone. A discussion of the security level of TrustZone is given in section 3.3.6.

In contrast to an SE, critical operations may be executed in the secure environment of the TrustZone. When investigating the SE approach, one can see that the trusted system is constructed from a RichOS application processor component and a small, resource constrained security device. The validation of the software image is done by components of the RichOS. Although it is generally protected by the boot process, manipulations and exploits during runtime cannot be excluded. Hence, software like the measurement or verification agent forming the RTV and RTM may be affected.

On a TrustZone system, these agents may be executed inside TrustZone. The software images that are loaded for execution may be sent to TrustZone and then validated by the agent that is executed inside the protected environment. Consequently, manipulations of these agents are much harder to employ. Due to resource constraints, this approach is not effective with the SE approach.

3.3.2 Validating integrity information

The integrity verification process involves reference integrity measurement (RIM) certificates which contain integrity information of certain software images and information of expected integrity metrics [52]. When loading a software image on a trusted embedded platform, the reference values of the certificate are used to validate the integrity of the software image - either when loading or after extending the hash of the image into a PCR so that they meet an expected configuration. If the result of the image validation fails, further execution of the image is aborted.

The integrity of these certificates themselves are checked by using asymmetric cryptography which can be rather time consuming and slow on mobile devices. In order to address this problem, the MPWG has introduced the concept of binding a RIM certificate to a certain MTM involving just symmetric cryptography with a key that is only known to the MTM. Using secure elements could improve this process greatly. Instead of binding the certificate to the MTM, the certificate could be stored within the MTM. Assuming that only authorized entities can update or store certificates within the element, the certificate's integrity can be seen as assured.

The benefit of this approach is that a verifier only has to create a hash of the RIM certificate and of the image and send it to the secure element. The secure element could then simply compare the hash certificate with the hash of the stored certificate to verify its integrity.

Moreover, the card could compare the corresponding PCR selection values stored in the certificate with the current content of the PCRs detecting aberrations between the expected and the actual platform configuration before extending the PCR, as defined in [52]

The same assumptions and methods apply to the TrustZone approach unless the TrustZone is equipped with a sufficient amount of non-volatile memory in order to store the hash values of the certificates or other material used for integrity protection.

3.3.3 Process separation

Process separation is especially important for isolating security relevant operations. However, mobile handsets and smart cards are highly sophisticated multiprocessing machines allowing different tasks to be executed concurrently. In the proposed architecture different types of MTMs are used on the same device. (see Figure 3.3). Therefore, process isolation between e.g. a MRTM and a MLTM process is essential in order to prevent unauthorized access to data.

In the JavaCard MTM implementation, the processor executing the MTM code is a physically distinct entity to the processor running application code. The interface between the MTM and the application is constrained by the ISO7816 [62] smart-card interface of the secure element. Data exchange between the MTM and the application is limited to an APDU based protocol, there is no mechanism for directly sharing memory between the MTM and other applications on the device. The nature of this smartcard interface automatically forces the MTM and any other applets running inside the secure element into a passive role, with respect to the application processor.

With a TrustZone system, the separation mechanism is created by the duplication of CPU registers and memory protection mechanisms for the two security domains. Communication and data transfers is established through secure monitors.

3.3.4 The role of virtual machines

Virtual machines play a key role in both of the designs discussed in this Chapter.

In the secure element based design, the primitives provided by the JavaCard framework and the Java language are used to realize protected capabilities and shielded locations for the MTM applet. Within the context of the Java environment running on the secure element, applet security and isolation is provided by the design of the JavaCard framework [105].

The JavaCard framework is designed to be usable in environments with extreme constraints on resources like memory and computational power. Today smart cards are often based on very simple 8bit micro controllers such as 8051-derivatives. These controllers mostly lack support for features like memory protection, virtual memory or a distinction between privileged and unprivileged processor modes.

Providing process isolation for applications running natively on such a limited processor becomes next to impossible. The JavaCard VM provides a powerful yet simple solution to remedy this undesirable situation. Instead of allowing applet writers to use the potentially dangerous native instruction set of the smart card processor, it provides a safe virtual machine instruction set. The virtual machine instruction set of the JavaCard VM (cf. [105], [74]) is designed to not expose any direct means for raw pointer or memory operations. In addition the Java virtual machine specification enforces a number of restrictions on valid programs to allow bytecode verification. In the context of current JavaCards, bytecode verification is mostly done outside the card. Since special keys are required to load applets onto the card, it is still possible to guarantee that only verified applets are installed. Once the applets are installed, the card can be locked, disallowing any further applet installations.

Based on bytecode verification and the virtual machine instruction set design of the JavaCard VM, it is possible to overcome the limitations of the underlying native processor with respect to applet isolation. Under the assumption that the virtual machine implementation is correct and secure, JavaCard VMs allow powerful software-isolation without the need for equally powerful underlying hardware isolation.

A separation is also required to protect software based roots-of-trust or other security services based on these roots. Such services that can be used by applications are called *trusted-engines*. They include different services such as remote attestation tasks or key-storage facilities. When discussing the TrustZone based prototype design, the possibility of implementing trusted engines as user-space processes, running in the secure world environment of the TrustZone based platform has already been mentioned. The ARM processors used in the TrustZone based design do not suffer from the same limitations with respect to memory protection and privileged instructions.

Nevertheless, trusted engines implemented as native processes can pose a threat to the entire secure world environment, especially if they have to process input from untrusted sources. Incorrectly implemented native trusted engines can give an adversary the capability of directly executing code in secure world user-space. While this does not necessarily lead to an immediate break of the platform security, it can be a highly significant advantage to an adversary.

A virtual machine based approach to trusted engines offers mechanisms to tightly restrict the low-level operations which can be carried out by software running inside the trusted engine. For example, potentially unsafe low-level operations like direct raw pointer manipulation can be ruled out by appropriate byte code design. Depending on the trade-off between performance and security requirements, virtual machines can implement a significant amount of run-time checks and byte code verification steps.

Examples for candidate VMs include the Java VM (J2ME, JavaCard) or the Lua VM. Especially the latter case of the Lua scripting language appears to be a quite attractive candidate due to the small size and high flexibility of the Lua programming language. It should be pointed out that Lua has already been used in designs with a similar problem setting, as demonstrated in [39].

3.3.5 Platform binding

The TCG specification defines that an essential requirement of TPMs is *platform binding*. On desktop systems, this binding is achieved simply by attaching the TPM to the motherboard of the PC. However, on mobile devices the situation is different. While SEs are attached to the mobile's board similar to desktop TPMs, MTMs on SIM/SD cards or software based MTMs can be removed.

If no binding is established, a malicious user may execute the following attack: He might boot the device into a trusted state and attach the SIM card to another device that is not in a trusted state. This way, he would be able to pretend the device to be in a specific configuration providing applications and external services with false attestation information. Therefore, binding mechanisms must be in place for software based MTMs and MTMs based in removable security devices.

But how can this binding be established? The basic requirement for such a binding is another security mechanism on the device. For example, a simple binding could involve a Diffie-Hellman key-sharing protocol where a key is shared between MTM and mobile device. The MTM functionality is then only available to the device if it can authenticate itself to the MTM.

However, if the authentication information is stored on the device it is prone to theft and manipulation. Therefore, an additional security component that controls the authentication information on the mobile side is required. Such a component could either be an on-board smart card like a SE or the TrustZone security mechanism. In case of the SE solutions, the authentication information is stored inside the security device and the MTM is only activated if the device can be authenticated.

Security enhanced embedded systems such the ARM Cortex devices are equipped with a hardware device unique key (HUK). This key is a symmetric key and is unique to each device, therefore, the MTM software module for the TrustZone solution may be encrypted with this key. Typically, the module is stored on a mass storage device such as a SD card or the main memory of the device. Consequently, before loading the MTM software image into the TrustZone execution environment, it is decrypted. If an attacker manages to copy the module to a different platform, he will be unable to execute it as the target device will not possess the HUK from the original device. During execution, the MTM is protected by the security functions of TrustZone preventing an malicious attacker to copy the software module from the platform.

Another approach might involve only integrity protection of the module. This means that during install the module is signed by device with a key derived from the HUK. Consequently, the MTM is not bound to specific device and may be transferred to another device.

Yet unanswered is the question how the MTM credentials such as the EK or DAA parameters are stored. The preferred approach when storing credentials on memory constrained embedded systems is to store the data outside of the protected environment. This requires cryptographic protection. Therefore, the typical concept is to derive a key based e.g. via a key derivation function defined in [63] on the HUK and encrypt the data with this key. Only the *salt* used during the key derivation process is then stored in the non-volatile memory.

A more secure approach is the combination with a SE, where the TZ executes the MTM and the credentials are stored in the SE. A detailed analysis of this approach in coherence with a concrete use- example is given in Chapter 4.

3.3.6 Security evaluations for embedded security mechanisms

Security evaluations are a wide spread method to estimate and harmonize the levels of security that can be assured by security modules. They are also used to evaluate secure elements, however, on TrustZone like devices, they can only be applied with limitations. The problems when evaluating a TrustZone device are multifaceted. First, the designer of the security extension is typically not the device manufacturer. The exact mechanisms and techniques employed for manufacturing the device are not known to the designer. Hence, an evaluation of the full system cannot be conducted by the original designer. Second, embedded systems like ARM processors are designed to be extended by the manufacturer. Consequently, the original designer is again not in the position to conduct an evaluation as it is not know to him which components are added to the platform. Third, the power drain of the rather powerful TrustZone processors is hard to control. Hence, they are more prone to side channel attacks than common smart cards.

The challenges for security evaluators are to find a cost and time effective way in order to keep the time to market delay of new devices at a minimum. Hence, a full evaluation of every new embedded device that appears on the market is out of question. Therefore, new methods are required in order to solve this gap. In contrast, a secure element can be evaluated by existing methods. In fact, as SEs typically originate from other security devices that already have security evaluations. Consequently, SEs are rather easy to evaluate - if it is necessary at all.

To overcome this situation, different approaches are subject to research at the moment. The concept proposed in this thesis is to use TrustZone and SEs in combination, thereby separating the device into three domains:

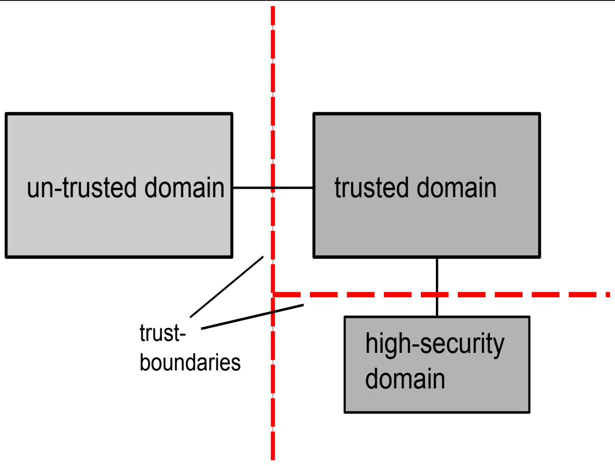
1 Un-trusted domain - in this domain, all user related application that are potentially seen as un-trusted are executed. No confidential material is stored here.

2 Trusted domain - this is the environment protected by TrustZone. In this domain, security related applications that need protection from the untrusted domain are executed. Secret information is not stored here.

3 High security domain - this is the environment provided by the SE. The primary use is to store credentials and key-material in a well protected device.

Figure 3.5 illustrates the principle of the architecture. The benefits of this three-domain separation is that the evaluation may be conducted separately according to different assurance levels. Assuming that the SE is already evaluated, only the TrustZone architecture remains to be evaluated.

Figure 3.5 3 Domain Separation for Security Evaluations



As secret material is only stored in the SE, the level of evaluation for TrustZone may be lower as if keys were stored in it.

3.4 Dynamic Command Loading for Security IC based MTMs

Trusted platform modules are typically monolithic software modules that provide a set of functions and commands following a defined specification. Specifications, however, are not static - they undergo updates and modification over time, thereby changing the defined set of features. Such modification might also include the addition of new commands.

However, once they are deployed in the field it is very hard to update their firmware as the required amount of memory to verify a firmware block inside the TPM in a secure way exceeds the memory of current TPMs.

This raises the question what can be done in order to modify a set of provided MTM commands in a deployed MTM without the requirement of changing the entire firmware. Moreover, how can the supported commands be composed according to the requirements of the current use-case thereby using the resources of TPMs efficiently? How can the TPM be updated accordingly and in a secure way?

To solve this question, the design and implementation of a concrete concept for dynamic command loading for mobile trusted modules is discussed in this Section. Based on the idea of [70], a specific approach is introduced which allows to build flexible and reconfigurable MTMs. The concept may be realized on dedicated micro controllers as well as software based MTMs. The proposed approach takes advantage of security features that are available on many mobile phones i.e. Secure Elements [30].

The new concept aims at achieving different things:

1. First, overcome resource limitations like the memory constraints of the Secure Element's EEPROM by limiting the provided functionality to the requirements of the current application.

2. Second, achieve algorithm flexibility. Current designs are nailed down to certain algorithms e.g. SHA-1 or discrete logarithm based Direct Anonymous Attestation schemes (DAA). By dynamically loading and updating the set of commands supported by an MTM, a higher level of algorithm-flexibility can be achieved. Moreover, the provided functions of an MTM can be switched either to support mobile-local-owner-trusted module (MLTM) features or mobile-remote-owner-trusted (MRTM) modules features.

3. Third, the MTM specification differentiates between *optional* and *mandatory* commands [111]. With dynamic command-loading, optional commands can be supported and installed on MTMs even if they have already been deployed in the field. Moreover, field-updates as defined in the TPM specification can be done securely and in a simple way by using the code authentication frameworks provided by the Java environment.

The proposed architecture is compatible with the current MTM specification from TCG and relies on security features provided by security ICs available on modern mobile platforms. The architecture neither requires modifications of the TSS specification, nor does it need extra hardware.

As previously discussed , mobile trusted modules may also be realized as smart-card applications [30], if the mobile platform is equipped with an adequate device. Such devices can be for example Secure Elements (SEs) as discussed in [30] and applications which are installed on the phone. Hence, from a certain point of view, a TPM may be seen as trusted execution environment (TEE). Instead of providing a fixed set of functions, a TEE can execute arbitrary code.

This approach allows to dissembed functionality of the TPM and let only the function required for specific operations remain in the TPM.

3.4.1 Design of the deployment architecture

The basic architecture of the introduced deployment mechanism works as follows: The application has to select the applet it wants to exchange APDUs with by the applet's application identifier (AID) which is unique for each applet. When this task was successful, the application may send commands to the card that are received by the SE's operating system and forwarded to the selected applet. The SE supports the installation of different applications (applets) which can support different features. For example, one applet may support integrity reporting functions, i.e. *quote* and the corresponding commands and another applet might support *sealing* or *binding* functionality. Each applet contains a set of TPM commands. For the experiments, the categorization defined by the TCG that separates the TPM commands in administrative functions, authorization functions, configuration reporting functions etc. was used. However, this categorization may vary depending on the use-case the MTM is applied. Therefore, the detailed grouping of commands is left to authorized third parties e.g. the MTM vendor.

Figure 3.6 Design Overview

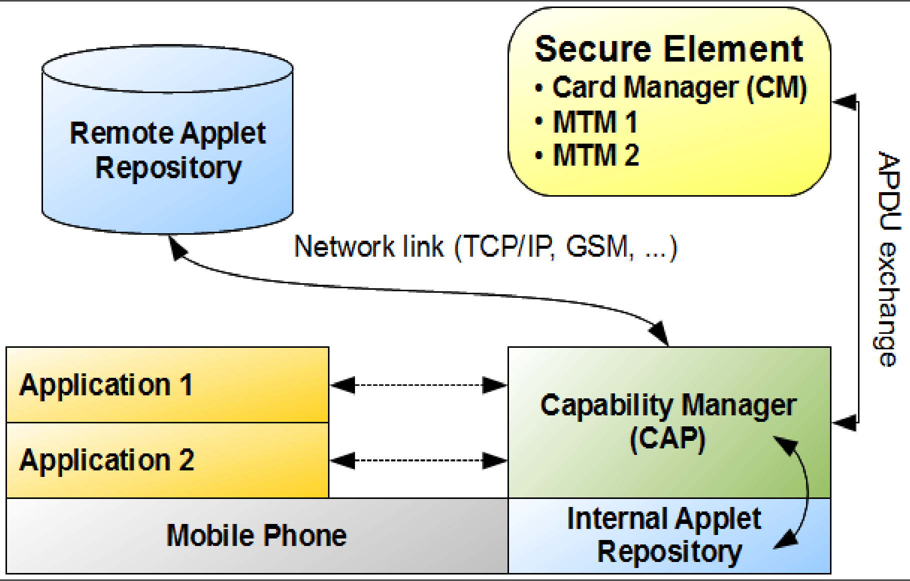


Figure 3.6 gives an overview of the proposed design. In addition, a module that manages the communication of the mobile applications with the SE is required. In the proposed approach, this is done by the *Capability Manager* (CAP). The CAP is responsible for communicating with the CM to instruct the CM to load and delete applets. Moreover, it is responsible for selecting the applet with the requested TPM functionality (a specific functionality can be requested by an application that wants to use the MTM, see discussion in subsection 3.4.3). The CAP has to manage three repositories: first, the repositories that contain the applets that can be installed in the SE. The CAP has to know in which storage it can find (externally or internally) applets with specific features. Second, the repositories that contain the validation information for the applets. And third the SE that

executes the applets as the CAP has to keep hold of which applet, providing a specific functionality is currently loaded in the SE/MTM.

The design (see Figure 3.7) is based on a *master applet* (MA) for each MTM instance. The MA is pre-installed and provides different services to the other applets on the card. It offers, for example, TPM command handling, like integrity check or parsing of the commands and command authorization functions that may be used by other applets belonging to the same MTM. Basically, any common function that is required by other MTM commands.

Moreover, the MA controls access and usage of the endorsement-key (EK) and EK certificate. The actual processing of the TPM command is handled by the specific applet. However, special care has to be taken when sharing cryptographic objects. A cryptographic object can be an instance of e.g. a *java.security.MessageDigest* or *javax.crypto.Cipher* class. Such objects may be initialized by one applet and then shared with another one. Hence, data that has been handed over to the object, e.g. data that should be encrypted, is now available to the other applet as well as it is stored in the cipher object. Typical scenarios require a single MLTM and a single MRTM. All applets that are related to these MTMs are assumed to be allowed to share data. More sophisticated application scenarios might require two or more MTM instances. In this case, care has to be taken when sharing such critical objects among applets and either delete the data stored in the objects or return a new instance of the corresponding object.

In addition, Figure 3.7 shows the layout of the MTM applets. A single MTM contains several applets that are merged into a single applet context. The applet firewall, which is a feature of the JavaCard runtime-environment, provides isolation for the contexts and prevents access from applets of one MTM to objects of another applet.

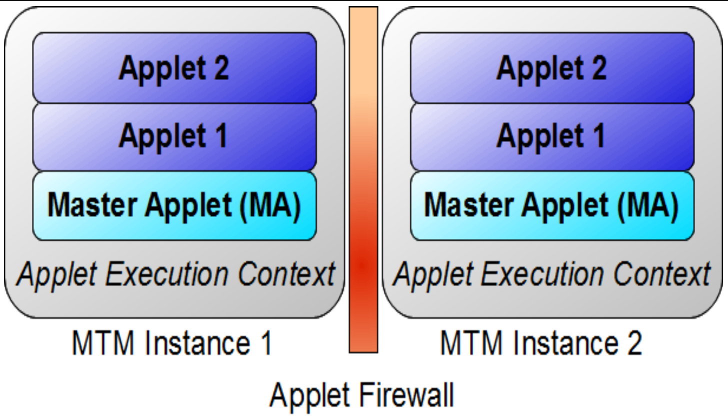
The JavaCard runtime environment provides a strong isolation between the different applets which means that an applet can not access the Java objects or fields from another applet. This isolation is required as the applet entry points are public and other applets might be able to get an object reference which could be used to gain access to protected information. Although the firewall provides an isolated execution environment for each applet, there has been a mechanism for sharing objects between applets since JavaCard version 2.1. of the JavaCard specification. The JC 2.1 environment provides a *shareable interface* where an applet can define a set of methods that are available to other applets. An applet can implement an arbitrary number of shareable interfaces and can extend other applets that implement shareable interfaces. Only the methods defined in the shareable interface can be accessed by other applets. The applets that provide shared objects are *server applets* and the applets that use the shared interface are *client applets*. Hence, the master applet is a server applet.

3.4.2 Installing applets

Using a virtualized environment like a JavaCard, has the advantage that applications on the card can be removed from the card and applets with new or different functions can be downloaded. Moreover, the virtualized environment provides *isolation* between applets allowing them to run in different (card) contexts. The proposed idea of the *TPM Execution context* ideally fits the concept of the (card) context.

The installation process (in the SE) is managed by the card-manager applet. This applet is pre-installed by the card manufacturer and handles the downloading and installation process of the applet. Before the card-manager accepts installation requests, it

Figure 3.7 Applet Isolation in the Secure Element



validates authentication information from the requester. Typically, the authorization is done via shared keys. One single authorization key allows to unlock the card and grants access to it. Hence, the instance that is in possession of the authorization key is able to install applets, thereby modifying the SE which is actually a root-of-trust. In future specifications like JavaCard 3.0 it is possible to use different authorization keys for different applications [75]. On the JavaCard of the experimental platform, the card manager (CM) is not able to verify the integrity and authenticity of the applets that are installed. Hence, this task has to be done on mobile platforms by the capability manager. More sophisticated cards support *Data Authentication Patterns* [46] which enable the card-manger to verify RSA signatures that are put on the applets.

The main task of the card manager is to install and delete applets. When an applet is loaded into the SE by the CM, it is stored in the card's EEPROM memory and the CM creates an instance of the applet, initializing the fields and objects used in the applet.

One requirement is that applets that can be unloaded (i.e. all applets except the MA) do not store permanent information. When unloading a card applet, all data stored in the space of the applet is deleted. Consequently, permanent data structures like TPM_Permanent_Data etc. must only be stored by the MA.

3.4.3 TPM command execution process

For the prototype design, the following procedure for using the SE based MTM functionality is defined: When an application on the mobile phone wants to send TPM commands to the MTM, it contacts the capability manager. The CAP manages the installation and removal of applets and knows which applet (i.e. TPM functionality) is currently installed in the SE. The CAP is also able to request new functionality (i.e. applets) from a trusted source and is able to install applets on demand depending on the resources of the SE. Sources for applets can either be external, remote repositories or SD cards or an internal storage e.g. the phones memory. However, it is important that the CAP or the CM can verify the integrity of the applets before loading it into the card. Hence, the applets have to be signed by a trusted third party. The CAP selects the corresponding applet and returns a session handle to the requesting application which can now send TPM commands to the MTM via session handle and CAP.

The TPM commands that are sent from the application and forwarded by the CAP are received by the applet that was selected by the CAP. The applet processes this request and returns the result. For processing the request, it may access the TPM command handling functionality that is provided by the master applet via the shareable interface.

In the experimental implementation, the communication with the CAP is established via a socket connection. The CM is a Java 2 Micro Edition application that embeds the incoming requests into smart-card APDUs and forwards them to the SE. A requirement that the application is allowed to do that is that it is signed with a Verisign or Thawte code signing certificate. Otherwise, the security policy of the device's on-board Java virtual machine prohibits communication between the application and the SE.

Storage and Protection of the Authentication keys Each time an application requires a new command set (i.e. applet) that is not in the SE, the CAP has to fetch the corresponding applet from one of its storage,s validate its authenticity and write it into the card. The current smart-card standard requires authentication before software images containing such applets may be downloaded to the card. In case of the JavaCard, this is achieved with a single authentication key. Hence, the entity which is in possession of the keys may modify the SE and with it a core-root-of-trust. As a consequence, special protection of the keys is required. The keys may be stored locally or in a remote storage. In both cases, the keys have to be *bound* to the MTM with the TCG's bind feature. Hence, the keys may only be decrypted if the device is in a specific configuration. The binding of the applet has to be done by the instance that publishes the applets for download. This approach has the consequence that for measuring and reporting the configuration of the platform, MTM functionality has to be provided prior to unbinding the keys. This requires an applet that provides integrity measuring, reporting and un-binding functions, to be already installed in the SE. Moreover, this approach requires the *secure boot* [34] feature that is going to be discussed in the following paragraphs.

3.4.4 Security considerations

The SE based MTM is executed in a separated smart-card like environment. Consequently, no further protection of the MTM processes from other programs is required. Access to the MTM is controlled by the master applet. Remember that, although the single applets receive the TPM commands directly, the incoming commands are handed over to the MA that performs the parsing of the commands and the integrity checking. Therefore, *shielded locations* and *protected capabilities* as required in [112] can be guaranteed.

The most critical task is the downloading of the applets into the SE. The proposed approach assumes that the smart-card is not locked (i.e. modification of its memory from outside is forbidden). Smart cards have a feature that allows the conservation of the card and preserving of the current content by preventing updates of the internal state from outside. In smart-cards that are not locked, every entity that knows the cards authorization keys is able to download its own applets. The introduced design requires a software module (the CAP) that handles loading, deleting and communication of the applets. Therefore, the CAP requires authorization keys to unlock the SE. As a consequence, the platform has to be in a trusted state before the CAP can unbind the keys and unlock the SE. This requirement can be achieved by a *secure boot* of the mobile device. Only if the device is in a trusted state, the CAP may unbind the authorization keys in combination with the minimal applet that provides the unbind and configuration measurement functionality. Otherwise, a malicious application may try to get access to the authorization key and modify the SE which results in a compromisation of the root-of-trust. After a new applet or set of applets has been downloaded to the SE, the CAP locks the SE with the keys and deletes it from its memory.

3.4.5 Test environment

For experimental purpose, a Nokia 6131 NFC mobile phone which has a Giesecke & Devrient smart-card on board that plays the role of the SE was used. The G&D smart-card is a Smart@Cafe Expert 3.1 smart-card that can be programmed with the JCOP tools. It provides about 65 kilo bytes of memory for JavaCard applets and is compliant to the Global Platform 2.1.1 [46] specification and Java Card 2.2.1 [101] specification. The SE on the Nokia phone has to be un-locked before it can be used. Therefore, Nokia has published an un-locking service. With this service the authentication keys can be reset to default values and allow developers to install their own applets and un-locking keys. The base size of the MTM command execution applet in our system requires about 12 kilo bytes of memory and contains the MTM constant definitions, permanent data, methods for storing and extending PCR values, AIK loading an activation functionality and command handling and parsing. A size estimation of a single MTM command is hard to give as the size varies according to the command's complexity. However, in order to provide a size estimation, the size of an applet that provides DAA functionality is provided. The applet contains the required DAA commands which are TPM_DAA_Sign and TPM_DAA_Join. The total size of the applet on the card is about 14 kilo bytes. As the DAA command is one of the most complex commands this can be considered as the upper bound for MTM command sizes.

3.4.6 Command set loading via NFC

The Nokia NFC phone provides another interesting feature - the NFC module. While this module is typically used to exchange small amounts of information at short ranges, it is also possible to directly communicate with the SE. A server could, e.g., send TPM commands to the phones SE via a NFC terminal when the phone is in range, bypassing the mobile application processor. This allows for another interesting application as command sets (i.e. applets) could be loaded via the NFC module instead via the CAP. This idea could allow a user to download new command sets from trusted access points, fitting the current requirements. Furthermore, it would avoid the critical un-locking of the SE by an application that is executed on the mobile platform itself.

3.5 Conclusion

In this Chapter, two approaches for building mobile trusted modules based on existing embedded platform features are discussed and compared according to their supported security infrastructure and the security requirements of mobile trusted modules. One design is based on ARM TrustZone and the other one on Secure Elements.

The ARM TrustZone approach, covered in subsection 3.1.2, focuses on using special capabilities of the platform and its main processor for implementing trusted computing building blocks in software. Apart from the requirement for a sufficient processor core with TrustZone support, this approach avoids dependencies on additional dedicated trusted computing hardware. In comparison to the second design approach introduced in this Chapter, the TrustZone approach can not rely on the same security properties and security assumptions which are inherited from smart card environments. Nevertheless it can be argued that it has the potential for matching the security properties of a dedicated hardware based MTM implementation closer than a software MTM implementation on a general purpose processor. Based on the additional memory and process isolation features offered through TrustZone, it can be concluded that the TrustZone approach allows a larger set of possible trust boundaries and domains than the solely use of an integrated smart card. Moreover, it can be concluded that software MTMs running in a secure world environment and exclusively using secure world memory can provide protected capabilities and shielded locations which are potentially stronger than their counterparts in the general purpose processor without TrustZone features.

The second approach discussed in this Chapter is based on a dedicated microcontroller also known as secure element. As mentioned at the beginning of Section 3.2, such secure elements are already deployed in a number of mobile phone platforms. this JavaCard oriented approach again focuses on reusing the existing secure element for hosting trusted computing building blocks, without creating the need for additional special purpose hardware. In this approach, mobile trusted module functionality is implemented in a dedicated smart card environment, sharing some similarity with the TPM modules available in many desktop PC systems. This MTM implementation approach inherits the security properties established by the JavaCard framework and its smart card nature.

It can be concluded that protected capabilities and shielded locations of the smart card based MTM implementation approach closely match their counterparts found in existing TPM modules. A definitive and exhaustive statement on the difference between the security properties of the JavaCard MTM and the TrustZone-based software MTM cannot be given at the moment as detailed security investigations of TrustZone are not available to the public. Properties of ARM TrustZone suggest that the software MTM implementation can achieve characteristics close to the JavaCard MTM's security properties at least for a subset of these properties. The precise limits of the security properties of the TrustZone design Section are part of ongoing research. To ease matters, ARM has initiated the *TrustZone Ready Certification Program* to support equipment manufacturers that want to evaluate their TrustZone based products [7].

Nevertheless, a reference design for estimating and setting boundaries for security assurance is introduced. This design takes advantage of the fact that secure elements have undergone detailed security evaluations. Moreover, the design is based on domain splitting which allows to assign different security assumption to the different domains of a mobile handset. These different domains have further impact on the security assumptions of MTM and other applications executed on the device. The combination of ARM

TrustZone and secure elements allows the introduction of a *semi*trusted environment. This results in a segmentation of security environments into environments with different security assumptions. Therefore, applications may rely on an un-trusted environment supporting a RichOS, a semi-trusted environment providing and a high-security environment. Depending on the application or algorithm, this design allows to split computations between the semi-trusted and high security environment. A concrete use-case that takes advantage from this architectural design facility is discussed in Chapter 4.

Another contribution of this Section is the introduction of a dynamic command loading concept for mobile TPMs that overcomes the resource limitations arising from the security facilities of embedded microcontrollers. The introduced architecture focuses on secure element based MTMs which implement the MTM functionality in applets, small Java applications that can be downloaded into the smart-card like trusted execution environment. The functionality of the MTM may be changed, depending on the functions provided by the applets that are downloaded. With this approach, it is possible to reduce the memory requirements of the underlying security architecture. Furthermore, a high level of flexibility can be provided when defining the actual functions that should be supported by an MTM.

3.5.1 Future work

Currently available smart cards are typically based on an 8 or 16 bit micro controller. Cryptographic operations are executed on co-processors, allowing a high performance when using symmetric or asymmetric cryptography. Future developments in smart-card technology will dramatically change the way smart-cards are used. For example, the specification defines that the cards may host servlets that are embedded in a web service container, allowing TPM functionality to by accessed via HTTP requests instead of APDUs. Such an environment allows applications to access the smart card's services via the HTTP protocol. TPM commands could then be set off via HTTP requests to the smart-card.

Moreover, security evaluations are getting more and more important. Especially, companies that are involved in security reliable products require estimations of the achievable security level of their products.Therefore, more extensive investigation and research on efficient methods for evaluating modern security mechanisms are sought. In addition, with the raise of privacy concerns and support of privacy protection in mobile handsets, the question arises how anonymity protecting products can be evaluated in order to provide reliable statements about the actual protective capability.

Chapter 4

Privacy Enhancing Technologies for Embedded Systems

4.1 Introduction

From the very beginnings of Trusted Computing, research on anonymity and privacy enhancing technologies has been a major issue. The protection of the privacy and anonymity of a trusted platform and its user is especially important when conducting transactions over the Internet or when performing authentication operations that allow to track and identify a specific platform. For example, if two platforms perform a remote attestation as specified by the TCG they require a proof of their current platform configuration. This proof is established by cryptographic means, in detail a digital signature is applied to the values stored in the PCR registers of the TPM.

However, using a Trusted Platform Module and its unique credential - the Endorsement Key (EK) - allows malicious entities to track the activities of specific platforms and eventually identify the owner of this trusted platform. Hence, the EK unambiguously identifies a certain trusted platform and, using this unique key, violates the anonymity of a platform to a great extent. Therefore, the use of the EK must be avoided and technologies protecting the TPM's and the platform's anonymity are required.

In general, using common digital signature schemes for attestation requires complex public-key infrastructures which create a major management effort. In addition, it does not protect the signer platforms' identity as its transactions can be tracked and identified via the signing platform's public-key. Therefore, other solutions have to be researched and applied.

A first approach to tackle this problem is to use a new key for every attestation request. However, using a newly created key alone is not enough - a proof that the involved TPM is genuine and that the key was solely used in this TPM also has to be provided. Consequently, the first approach introduced by the TCG, in order to address this problem, was the concept of the *PrivacyCA* (PCA). For every single transaction, a new asymmetric-key, in detail *attestation identity key* (AIK), is used. Every newly created AIK [112] is then sent to and certified by a PCA. Each AIK is then issued an AIK-certificate. The AIK has the property that it is non-migratable from one TPM to another and that it is created, used and destroyed in a specific TPM. Consequently, the verifier has confidence that the used AIK stems from a genuine TPM and that the associated platform has the properties attested by the PCA.

However, sending a new AIK to a PCA for certification creates the problem that the PCA has to be highly available. Once the PCA is off-line or unreachable, no new AIKs can be certified and the scheme fails or the platform re-uses an AIK, thereby risking to reveal the platform's identity. Moreover, it is unclear how often a new key should be created. How often the same AIK should be used e.g. for one secure channel connection or for a certain period of time such as one day or one week - is not specified, yet. Also unanswered is the question who should host such a PCA. It may be hosted by a company providing anonymity protection for the company's employees or it may be hosted by a third-party outside of a company's domain.

However, for the latter case the question arises how the trust relations to the PCA are defined. Which platforms are allowed to receive certificates form this PCA and basically what is testified by a certificate from this specific PCA? From a certain point of view, the PCA represents a certain group of platforms and with it a certain set of properties of these platforms.

Moreover, the security assumptions of the PCA have to be investigated. Basically, a PCA may either record the issuance of every requested certificate or it may delete all evidence of this process. As a consequence, the PCA is either available to identify the requesting platform by combining the request and the issued certificate or it may simply have no indication of this relation. The consequences are manifold, on the one hand the PCA is a single point of interest for adversaries and the disclosure of the certification issuance information may compromise the privacy of single platforms. On the other hand, revocation of the issued AIK-certificates is virtually impossible.

The validity period of AIKs also proves to be an open issue. The basic question that has to be addressed is how long should the validity period of AIK-certificates be defined? Shorter periods may reduce the need for an efficient revocation mechanism while longer periods might require revocation.

In addition, revocation of issued AIK-certificates may become a complex task especially in cases where the trust relation to the used TPM is broken. This can be the case if a flaw in this specific TPM or set of TPMs has been found. As a consequence, if a certificate of an affected TPM is revoked the platform may choose a different one from a different PCA to hide its identity until all concerned PCAs are informed and all corresponding certificates are revoked.

A more efficient approach would be to certify the keys locally - namely on the platform. The signing key of the PCA could simply be distributed among a set of trusted platforms and the TPM could care for the protection of these keys. However, if one TPM is compromised, all platforms sharing the same signing key are compromised, as well.

In order to overcome the problems that arise from using PCAs or from supplying platforms with shared keys, the TCG introduced the Direct Anonymous Attestation (DAA) scheme [13]. It allows TPMs to sign AIKs on behalf of a group of trusted platforms, each platform being equipped with a unique key. Although, the (DAA) scheme eliminates the requirement of a remote authority, it includes complex mathematical computations. Therefore, the question arises how this protocol can be employed efficiently on modern computing platforms. The term platforms here refers to signing platform as well as verifying platforms.

Even though the TCG states DAA to be *optional* in their mobile TPM specification, there is a high demand for anonymity on mobile and embedded devices. Modern cell phones are able to join up in ad-hoc groups, they exchange data via Bluetooth or near-field-communication (NFC) or support the same or similar applications as desktop platforms do.

Therefore, they are prone to location tracking and behavioral profiling, thereby threatening their owner's privacy. While it is obvious that the network provider is able to identify a certain mobile handset, it does not mean that other cell phones or providers of online services should be able to reveal a certain platform's identity. Consequently, support for anonymity protecting capabilities should be considered *mandatory* on future mobile platforms designs.

However, DAA has strong protection assumptions on the used private credentials. If an adversary manages to steal these credentials from the platform he is able to create DAA signatures on behalf of the group without ever officially entering the group and without validating its true identity. Therefore, strong protection mechanisms have to be applied in order to support this scheme in a secure way.

While on desktop systems this part is covered by the TPM, mobile devices lack such a device. Yet, it is not clear how architectures look like that employ secure implementations of DAA mechanisms on modern embedded systems and which performance can be achieved so that real-world use-cases can be developed. Fortunately, approaches for hosting private credentials such as DAA credentials securely have been introduced. The designs introduced in Chapter 3 form an ideal basis for hosting and processing such credentials.

As a result of the discussions in the previous Chapters, this Chapter focuses on the question how embedded security architectures can be used in order to host DAA functionality on embedded systems. Moreover, it focuses on which performance can be achieved on desktop platforms and mobile devices when creating and verifying DAA signatures. In order to provide a comparison of different platforms, focus is laid on investigations on Java enabled devices as Java provides a high level of portability. The targeted platforms are the Java 2 Standard Edition (J2SE) which is the common virtual machine for desktop platforms and the Java 2 MicroEdition/Connected Limited Device Configuration J2ME/CLDC [84] which is the most widespread virtual machine on mobile phones, nowadays. The investigation focuses on related publications to the DAA scheme defined by Brickell, Camenisch and Chen which is also known under the abbreviation *BCC* scheme. As the scheme was published in 2004, it is also referred to as *BCC04*.

In the proposed approach, the idea from Chapter 3 is extended where TPM functionality is integrated into an on-board smart-card, the *Secure Element* (SE). With the basic Trusted Computing functionalities, namely *shielded locations* and *protected capabilities* ([35], [112]) they form a major building block for secure execution environments. Moreover, the supported cryptographic features of these elements provide an ideal basis for implementing the required cryptographic algorithms.

As the required computations for performing DAA tasks are the same for TPMs and mobile TPMs (MTMs), throughout this Chapter the terms TPM and MTM refer to a trusted module that provides the required DAA functionality .

Furthermore, this Chapter focuses on the basic DAA functions that are DAA signature creation, verification and the join process that allows new clients to enter a group. In order to generate significant measurement values, a reference setup was implemented including a crypto library that provides these basic DAA functions. The implementation relies on the discussion on DAA given in [76] by Brickell, Camenisch and Chen and is the first one addressing this scheme. The library was designed to work on different embedded platforms as well as on desktop systems and is the first implementation of a DAA scheme on a mobile phone. In order to achieve portability, the Java programing language was selected which is also a perfect tool for rapid-prototyping.

The Chapter also includes an investigation of a real-word DAA use-case. For this investigation, the TLS client authentication mechanism discussed in Chapter 2 was chosen. The gathered results offer valuable clues about the practical application of the DAA technology.

Another contribution of this Chapter is the investigation of the DAA scheme for contactless, anonymous authentication mechanisms used in NFC enabled devices, with respect to secure storage and authorized access to the DAA credentials and sufficient performance. It is demonstrated how the DAA protocol has to be extended in order to realize a system which is able to compute authentication information with a reasonable performance that can be applied in real-world scenarios. In addition, off-the-shelf devices are used in order to generate experimental results and to support this statement. The approach can be used to enable privacy protecting technologies on low-cost devices which were previously only available on cost-intensive, high-end devices.

The remainder of this Chapter is organized as follows. An investigation of existing publications and related work on DAA is given in Section 4.1.1 which is followed by an introduction into the DAA scheme and background information on privacy enhancing technologies on trusted platforms in Section 4.1.2.

An analysis of the DAA scheme for TLS client authentication is provided in Section 4.2. This Section is followed by an analysis of the DAA scheme on current smart phone platforms in Section 4.3 as well as a discussion of the use of DAA for NFC authentication in Section 4.4. Furthermore, the reference designs and implementations as well as the integration in common cryptographic frameworks is examined. The different test setups and test devices for the DAA performance measurements are described and a discussion of the measured results including performance values is given. Moreover, implementation details of the DAA library are discussed and finally, the results are summarized and future directions for investigations and improvements are proposed in Section 4.5.

4.1.1 Related Work

Several ideas for integrating anonymous authentication and Trusted Computing technology in TLS have been published. Latze et. al. propose to use the TPM for identity distribution, authentication and session key distribution and have defined an Extensible Authentication Protocol (EAP) extension in order to integrate the Trusted Computing and TPM related information [15]. Although the protocol supports anonymous authentication, it is based on the PCA scheme and not on DAA. Moreover, this document is currently work-in-progress and is tagged to be in an "experimental" status.

The approach from Cesena et. al [38] aims at providing anonymous authentication for trusted platforms and trusted applications in the sense of Trusted Computing. In their work, they define a set of extensions for TLS for transporting DAA related information and propose a design for integrating it into OpenSSL. They use the DAA scheme as defined in the Trusted Software Stack specification [113] which requires them to use a full blown Trusted Software Stack (TSS) [56]. They also provide support for a pairing based crypto variant of the DAA scheme in their architecture which is not supported by existing TPMs. In contrast, the proposal discussed in this Chapter focuses on the integration of existing TPM v1.2 into TLS with all their advantages and disadvantages. Moreover, the proposed modifications are designed following a light-weight design principle so that they can be employed on embedded systems with consideration of memory constraints.

Another interesting publication that is not directly related to this work, but may have interesting implications on further applications, is discussed in [10]. Bichsel et. al. propose an implementation of DAA on a JavaCard. Using this approach, it might be possible to use the anonymous authentication mechanism in combination with smart cards instead of TPMs. This approach could increase the flexibility and area of application of the anonymous TLS authentication as the anonymous credentials can be bound to a specific user instead to a specific TPM and platform.

Furthermore, different approaches have been published to integrate DAA functionality on smart-cards: The basic idea of splitting the computations between a resource limited micro-controller and a powerful host has first been discussed by Brands in [12]. Bichsel et. al. [11] and Sterckx et. al. [93] analyzed implementations of variations of the DAA signature protocol on JavaCards. Both publications give performance results of their implementations which show that a practical use of the DAA scheme requires a powerful host to execute the host side computations of the protocol. This statement is supported by Balasch [64] who implemented a DAA scheme on an AVR micro controller. He concludes that the DAA scheme can only be practically used in combination with a resourceful host. As shown in [11], the computation of an entire signature takes about 16,55 seconds with a modulus length of 1984 bits on a JCOP v2.2/41 JavaCard. Balasch requires 133.5 seconds on an 8 bit micro-controller (with 1024 bit modulus) and Bichsel 450 seconds [11], however, Bichsel and Balasch only take the computations located inside the TPM into account.

In [76] by C. Mitchell, Trusted Computing in general is discussed, but the publication also provides a chapter about the history of DAA. This history includes a discussion of several different approaches and algorithms for anonymous attestation schemes like the Group Signatures with Revocation Evidence (GSRE) scheme or the Boneh, Brickell and Shaham (BSS) scheme. Moreover, it describes a modified variant of the DAA scheme which requires less parameters and - consequently - less modular exponentiations, thereby increasing the performance and reducing the complexity of the scheme [76] (pp. 143-174).

One contribution of this Section is the investigation of anonymous signatures on mobiles using the managed environments for execution of the DAA computations and taking advantage of the protected environments discussed in Chapter 3. The ARM TrustZone CPU extension [8] which supports the separated execution of trusted and un-trusted code in a *secure* and a *non-secure world* and secure elements can be used as a building block for hosting TPM functionality [117] and for storing and using DAA credentials in a secure environment as well. The benefit of this approach is that the software running in the protected world can take advantage of the computing power of the main CPU so that anonymous signatures can be computed in sufficient and user acceptable time [32]. However, this special CPU extension is currently only available on some high-end smart phones using ARM-11 and ARM-Cortex CPUs. For cheaper, low-cost phones, which typically employ ARM-9 or ARM-7 CPUs this technology is not available. Fortunately, these devices may fall back on secure elements which are either attached to their main boards or added in form of external SD-cards.

Furthermore, most of the publications mentioned above omit *rogue tagging*. Rogue tagging is a mechanism to detect malicious TPMs and is, therefore, an important feature when using anonymous credentials. Nevertheless, it has a significant impact on the performance of the entire protocol and mechanisms for pre-computing DAA signatures.

In one of the latest publications, a DAA scheme based on elliptic curve cryptography (ECC) and bilinear maps [14] is proposed. It builds on the Camenisch-Lysyanskaya signature scheme and takes advantage of the much shorter key lengths provided by ECC. However, ECC is not supported by currently deployed TPMs.

The most important publication addressing DAA is [13] in which the general concept of

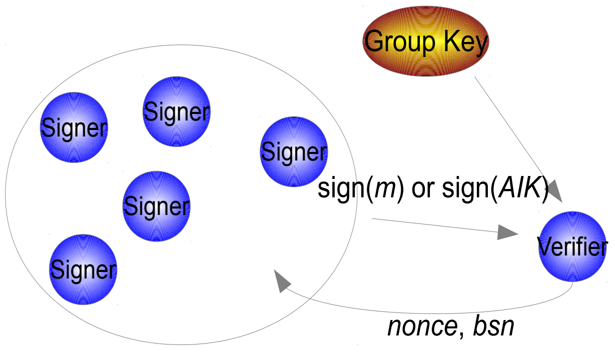
DAA is introduced. Moreover, in this paper, the scheme and the algorithms on which the DAA features of existing trusted computing enabled platforms are based on, are described.

4.1.2 Background on DAA

The model

In the DAA scheme, a platform basically *hides* in a group of other platforms. Figure 4.1 shows the three different parties involved in the DAA scheme: First, the *issuer* or *group manager* that creates and issues the group parameters and the group's public key. Second, a *trusted platform* that wants to create DAA signatures. This platform may be a Trusted Computing enabled desktop PC or a trusted mobile platform. Third, a *verifier* that verifies the DAA signatures created by the signer and which is in possession of the group's public key. The DAA scheme is based on group signatures, therefore, a platform has to obtain the group parameters and credentials for its private DAA keys from the issuer before it is able to compute DAA signatures. This step is called join process as discussed in Section 4.1.2. The signing platform can then compute proofs that it is in possession of the private DAA key and the credentials from the issuer. In context of Trusted Computing, the messages to be signed are AIKs - instead of sending them to a PCA, they are signed on the platform with the platform's unique DAA key. The verifier can then verify the signature and, therewith, the authenticity of the AIK with the group's public-key.

Figure 4.1 The DAA Model



The DAA relies on three protocols: the *join*, the *sign* and the *verify* protocol. However, prior to executing these protocols, the parameters for the group have to be set up by the group manager (see Appendix 6.1). The result of this operation is the private-key (p', q') and the public-key $(n, S, Z, R_0, R_1, \gamma, \Gamma, \rho)$.

In addition, the group-key may have a proof that the parameters of the key were generated correctly. This proof can be used to verify that $Z, R_0, R_1 \in \langle S \rangle$ and $S \in QR_n$ mod n are properly constructed (see Appendix 6.1).

Join protocol

Before a platform may create a DAA signature on behalf a group it has to join the group and obtain special group credentials.

The required steps for the join-process are briefly discussed in the following paragraph. Note that for the hash computations, the SHA-1 algorithm is used.

1. The TPM computes f from a seed and the long term issuer key and splits f into f_0 and f_1, each 104 bits of size. Moreover, the TPM computes a random ν' with length $l_n + l_\phi$ and the commitment $U = R_0^{f_0} R_1^{f_1} S^{\nu'} \mod n$ that is forwarded to the issuer
2. Next, the TPM proves the knowledge of f_0, f_1 and ν' to the issuer. It computes random numbers r_{f_0}, r_{f_1} of size $l_f + l_\phi + l_H$ bits and $r_{\nu'}$ with length $l_n + 2l_\phi + l_H$. Moreover, it computes $\tilde{U} = R_0^{r_{f_0}} R_1^{r_{f_1}} S^{r_{\nu'}} \mod n$.
3. The host computes $c_h = H(n \| R_0 \| R_1 \| S \| \tilde{U} \| n_i)$ where n_i is a nonce from the issuer with length l_H.
4. The final hash is computed inside the TPM via $c = H(c_h \| n_t)$ where n_t is a nonce chosen by the TPM. Furthermore, the TPM computes $s_{f_0} = r_{f_0} + c * f_0$, $s_{f_1} = r_{f_1} + c * f_1$ and $s_{\nu'} = r_{\nu'} + c * \nu'$ which are forwarded to the host and to the issuer.
5. The issuer verifies that $\hat{U} = U^{-c} R_0^{s_{f_0}} R_1^{s_{f_1}} S^{s_{\nu'}} \mod n$ and that $c == H(H(n \| R_0 \| R_1 \| S \| U \| \hat{U} \| n_i) \| n_t)$ and that s_{f_0} and $s_{f_1} \in \{0,1\}^{l_f + l_\phi + l_H + 1}$ and $s_{\nu'}$ lie in $\{0,1\}^{l_n + 2l_\phi + l_H + 1}$
6. The issuer now computes the Camenisch-Lysyanskaya credential (A, e, ν'') and computes a random $\hat{\nu}$ of length $l_\nu - 1$, a prime e with $2^{l_e - 1}$ <e <$2^{l_e - 1} + 2^{l_e' - 1}$ and $\nu'' = \hat{\nu} + 2^{l_e - 1}$
7. For the Camenisch-Lysyanskaya credential, the issuer must also compute the inverse of e: $d = \frac{1}{e}$ via the modulo inverse
8. compute $\phi = (p-1)(q-1)$ and
9. finally compute $d = e^{-1} \mod \phi(n)$.
10. Next, the issuer convinces the host that it computed $A = (\frac{Z}{US^{\nu''}})^{e^{-1}} \mod n$ correctly.
11. The issuer computes $\tilde{A} = (\frac{Z}{US^{\nu''}})^{r_e} \mod n$ and $c' = H(n \| Z \| S \| U \| \nu'' \| A \| \tilde{A} \| n_h)$ and $s_e = r_e - c'd \mod p'q'$ with $r_e \in [0, p'q']$
12. The host can now verify the proof by computing $\hat{A} = A^{c'} (\frac{Z}{US^{\nu''}})^{s_e} \mod n$ and verifying that e is prime and that $2^{l_e - 1}$ <e <$2^{l_e - 1} + 2^{l_e' - 1}$ holds. Moreover, the host verifies that $c' = H(n \| Z \| S \| U \| \nu'' \| A \| \hat{A} \| n_h)$
13. (A, e, ν'') are stored on the host while ν'' is sent to the TPM, where it computes and stores $\nu = \nu'' + \nu'$

After the execution of these steps, the host and the TPM possess the correct parameters to create DAA signatures which can be verified with the issuer's public key. Moreover, the client has obtained a secret key f, ν (stored in the TPM) and the credentials (A, e, ν'') stored on the host platform.

One very important building-block for DAA is the Camenisch-Lysyanskaya (CL) signature scheme. CL-signatures provide the basis for efficient proofs of possession of a certain credential. A detailed discussion of this scheme and CL-credentials is given in [16].

Sign protocol

In the reference implementations, the computation of the DAA signature is done according to the more efficient BCC05 scheme algorithm [76] which works as follows:

1. If revocation is enabled and the verifier requests rogue tagging, the verifier sends *basename* to the host which computes $\zeta = H(basename)^{(\Gamma-1)/\rho} \mod \Gamma$ which is sent to the TPM.
2. The host part of the signature class computes: $T = AS_0^{w_0} S_1^{w_1} \bmod n$ with $w_{0,1} \in \{0,1\}^{l_n+l_\phi}$
3. The TPM computes $N_V = \zeta^{f_0+f_1*2^{104}} \mod \Gamma$. Host and TPM can now compute the "signature of knowledge":
4. The TPM computes: $\tilde{T}_t = R_0^{r_{f_0}} R_1^{r_{f_1}} S_0^{r_{\nu_0}} S_1^{r_{\nu_1}} \bmod n$ with r_{f_0}, r_{f_1} of size $l_f + l_\phi + l_H$ bits and $r_{\nu_{(0,1)}}$ with length $l_n + l_\phi + l_H$. $\tilde{N}_V = \zeta^{\tilde{r}_f} \mod \Gamma$
5. $\tilde{T}_t$ and $\tilde{N}_V$ are returned to the host which computes: $\tilde{T} = \tilde{T}_t T^{r_e} S_0^{r_{\bar{\nu}_0}} S_1^{r_{\bar{\nu}_1}} \bmod n$ where $\tilde{T}_t$ is the input from the computation process that was performed in the TPM. The random parameter r_e is of size $\{0,1\}^{l'_e+l_\phi+l_H}$ whereas $r_{\bar{\nu}_0}$ and $r_{\bar{\nu}_1}$ are of size $\{0,1\}^{(l_e+l_n+2l_\phi+l_H+1)/2}$ bits.
6. Moreover, the host part computes: $c_h = H((n\|R_0\|R_1\|S_0\|S_1\|Z\|\gamma\|\Gamma\|\rho)\|\varsigma\|$ $T\|N_V\|\tilde{T}\|\tilde{N}_V)\|n_v)$
7. The TPM selects a random $n_t \in \{0,1\}^{l_\phi}$, computes $c = H(H(c_h\|n_t)\|b\|m)$ and $s_{\nu_0} = r_{\nu_0} + c * \nu_0$, $s_{\nu_1} = r_{\nu_1} + c * \nu_1$, $s_{f_0} = r_{f_0} + c * f_0$ and $s_{f_1} = r_{f_1} + c * f_1$ where b is a parameter that defines whether the authenticated data is a key that was loaded into the TPM or some arbitrary data.
8. The host part computes $s_e = r_e + c * (e - 2^{l_e-1})$ and $s_{\bar{\nu}_0} = s_{\nu_0} + r_{\bar{\nu}_0} - cw_0e$, $s_{\bar{\nu}_1} = s_{\nu_1} + r_{\bar{\nu}_1} - cw_1e$
9. Finally, the host assembles the signature $\sigma = (\zeta, T, N_v, c, n_t, (s_{\bar{\nu}_0}, s_{\bar{\nu}_1}, s_{f_0}, s_{f_1},$ $s_{f_e}))$. The signature σ can now be verified by the public key $PK_I = (n, R_0, R_1,$ $Z, \gamma, \Gamma, \rho)$.

Note that the parameter S is separated into $S_0 = S$ and $S_1 = S^{2^t}$ and $\nu = \nu_0 + \nu_1 * 2^l$ as the crypto co-processor of the TPM cannot compute the modular exponentiations whether the exponent is larger than the modulus n. The join protocol discussed in the previous section can be used with the BCC04 and BCC05 versions of DAA. BCC05 does not have its own version of a join protocol. The computation of the signature is separated between host and TPM, prohibiting that the credentials (A, e, ν'') that are stored on the platform can be copied and used on a different platform.

Verify protocol

For verification of a DAA signature, the verifier performs the following steps (also based on BCC05):

1. Compute:
$\tilde{T} = Z^{-c} T^{s_e + c2^{l_e-1}} R_0^{s_{f_0}} R_1^{s_{f_1}} S^{s_{\hat{\nu}}} \mod n$

2. Check that $c == H(H(H(n\|R_0\|R_1\|S\|Z\|T\|\hat{T}\|n_\nu)\|n_t)\|m)$ where n_ν is the nonce, previously sent by the verifier to the signer and n_t the nonce generated from the signer's TPM.

3. Moreover, the verifier checks that $s_{f_0}, s_{f_1} \in \{0,1\}^{l_f+l_\phi+l_H+1}$ and $s_e \in \{0,1\}^{l'_e+l_\phi+l_H+1}$

4.2 Anonymous Client Authentication for Transport Layer Security

In this Section, a concrete use-case for for anonymous authentication on desktop systems is investigated. The use-case is based on anonymous authentication for secure channel technology, in detail, focus is laid on anonymous authentication of clients using TLS client authentication as discussed in Chapter 2.1.

Instead of using a full-blown Trusted Software Stack (TSS) which is typically used for trusted applications to access TPM features a simple design for integrating anonymous signatures in existing security frameworks like the Java Cryptography Architecture (JCA) is proposed and used for further investigations. In addition, the design allows easy integration in other widespread embedded security environments like the Global Platform API [23] or the JavaCard API [105].

The changes required in the TLS protocol are also reduced to a minimum. The proposed light-weight approach allows the application on embedded systems and mobile phones which are going to be equipped with TPMs in the near future [111]. The software stack and the modified TLS protocol are combined in a library with focus on portability, size efficiency and simple usability.

This minimal library also contains an implementation of the DAA protocol. The used DAA protocol is based on the DAA scheme which we will address as BCC05 [76], named after its authors (Brickel, Camenish, Chen) and its year of publication.
In contrast to the DAA scheme defined by the TCG (BCC04) in the TPM 1.2 specification, we use the BCC05 scheme which is a performance optimized scheme that requires less parameters and less computations than the BCC04 scheme, however, both can use the DAA features from TPMs without any modifications of the TPM. A discussion about the differences between BCC04 and BCC05 and the security of the BCC05 scheme can be found in [76].

Furthermore, performance measurement values demonstrating the performance that can be achieved using this technique in combination with currently available TPMs are provided. The DAA test library is developed in the Java programming language and is designed to fit into the Java Cryptography Architecture (JCA) [102], allowing easy use of the DAA functions.

4.2.1 TLS Client Authentication

TLS client authentication allows a server to request authentication information from clients that want to connect to it. Figure 2.1 in Section 2.1.2 shows the basic flow of a TLS handshake. The messages marked with (1) are required for client authentication which works as follows: The server sends the client a list with certificate authorities (CAs), which it accepts. The client selects a CA and returns a $Certificate$ message that contains the client's certificate, which then certifies its authentication key. The $CertificateVerify$ message contains a signature on the hash of the handshake messages sent so far. By verifying the signature - the hash can be computed by the server as it knows all messages that have been exchanged with the client - and by verifying the client's certificate plus the corresponding certificate chain, the server can validate the client's authentication information [107].

Instead of sending an X.509 certificate containing an standardized public-key like RSA, ECC, DH etc. the client may now send a certificate that contains the DAA group-key.

Moreover, the hash of the in- and outgoing messages - as required for client authentication - is now signed with an anonymous DAA signature. This may be done in two ways, either the DAA signature is applied directly on the hash data or a temporary key which is typically a RSA or ECC key is used to create the signature. The public RSA key is then certified by the DAA signature and the RSA signature is thereafter applied on the hash values. The RSA or ECC key-pair is used only once.

4.2.2 Test setup

In order to obtain performance results, a test setup was created. The setup uses the features of the library discussed in the previous section. Furthermore, a library that provides the required cryptographic operations for the BBC05 scheme and the DAA commands as defined in [112] has been added.

In addition to TPM specific commands for key management and authorization, the library supports the following minimum set of TPM commands required for DAA operations:

1. TPM_DAA_Sign
2. TPM_DAA_Join
3. TPM_FlushSpecific
4. TPM_OIAP
5. TPM_Terminate_Handle

and the following structures: TPM_DAA_ISSUER, TPM_NONCE.

Support for the modular arithmetic operations, the RSA-OAEP encryption scheme required for loading DAA credentials into the TPM and the DAA protocol operations is provided by the IAIK-JCE-MicroEdition. Details of the implementation of the cryptographic functions and the BCC05 scheme can be found in [31]. In addition, the RSA key-pair generator was modified in order to compute the special prime numbers required for the DAA parameters [13].

The DAA scheme is basically a signature scheme like RSA or ECDSA. Consequently, it can be integrated into existing security or cryptographic software frameworks like the Java Cryptographic Architecture. The development of a JCA provider for the DAA library allows to abstract the complex API definitions in the TSS [113] specification and makes it accessible to developers that are not familiar with Trusted Computing.

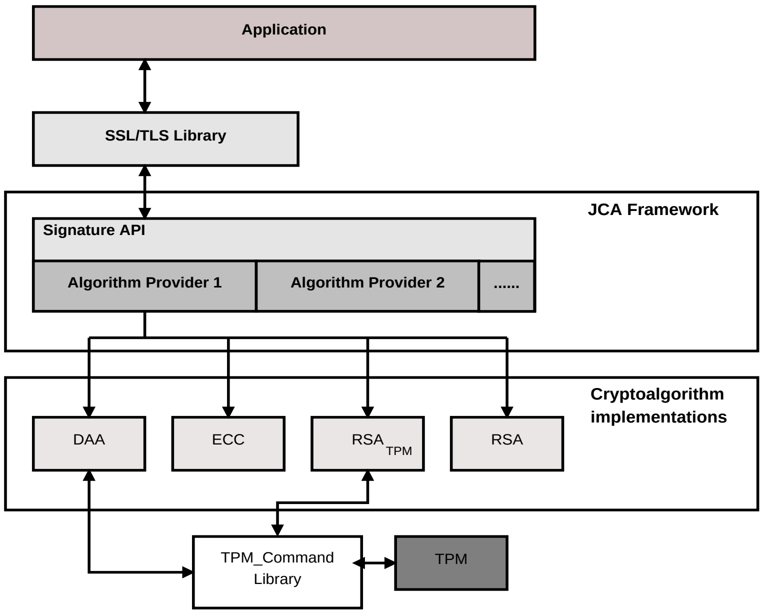
Access to the different TPMs is provided by the Linux kernel module drivers from the specific vendors. The TPM is mapped into the userspace via the `/dev/tpm` device alias. This device can then be accessed by a Java `java.io.RandomAccessFile` object in order to send and receive TPM commands.

4.2.3 Integration into the JCA Architecture

The TCG has specified an API for using the DAA features on trusted platforms. However, this specification is rather complex and is - even for developers that are familiar with Trusted Computing - hardly accessible. As mentioned before, the DAA scheme is basically a signature scheme, hence, it is well suited for integration in the JCA. The advantage of using such a common framework is that for creating signatures, the same API, independent from the signature scheme can be used. Consequently, DAA schemes based on different cryptographic primitives (e.g. discrete logarithm or pairings) can be used with the same commands.

Figure 4.2 shows how our DAA implementation is integrated into the JCA framework. The application uses the iSaSiLk library [94] in order to provide secure channel mechanisms.

Figure 4.2 Integration of the DAA library in the JCA Architecture



The library uses the JCA framework to request a certain algorithm implementation according to the TLS session parameters that have been negotiated during the TLS handshake. The algorithm implementation can be either a software implementation of the algorithm or an interface to some hardware device. The provided DAA signature implementation consists of two parts: the host module and the TPM module. Both modules are abstracted by the signature class API. The host module performs the DAA computations that can be done in software while the TPM module handles the communication with the TPM. For using the DAA sign function of the TPM, the TPM_DAA_Sign [113] command is sent several times in sequence with different parameters to the TPM. From the application's point of view, the algorithm can be initialized via the common Signature API by defining the algorithm and algorithm parameters. The same approach can be applied to use the RSA implementation in the TPM (TPM_{RSA} in Figure 4.2). Moreover, the JCA framework allows different DAA schemes to be used with the same API. For example, the BBC04 scheme could be added to the framework as an alternative to BCC05.

The following listing gives an overview of how a signature object using the DAA algorithm can be initiated and used. Prior to creating a DAA signature, the signature object has to be initialized with several parameters (via the signature.setParams(...) method). Some of the parameters are acquired during the Join phase and are encrypted with the TPM's endorsement key. These parameters include: TPM authentication information, the public part of the EK, issuer parameters, the encrypted values $enc_{EK}(\nu_0)$, $enc_{EK}(\nu_1)$, the encrypted DAA keys $enc_{EK}(daaBlobKey)$ (i.e. group specific credentials that are bound to a specific TPM) and the *basename* and a nonce n_v from the verifier.

```
Signature signature = Signature.getInstance("DAA/BCC05",
"Provider");
signature.initSign();
signature.setParams( daaParams );
signature.update( "Message to be signed".getBytes() );
byte[] sigValue = signature.sign();
```

[Listing 1. Example Code]

The result of the *sign*() method is a DAA signature σ on the message m.

The verification process is handled in a similar way by the same class. However, the signature object is initialized in verification mode and the task does not involve a TPM. The resulting implementation consisting of TPM commands, DAA implementation (Join and Sign protocols) and other cryptographic algorithms (RSA and RSA with OAEP, SHA1-HMAC) is about 150 kilobytes of size. In addition, the TLS implementation is about 120 kilobytes resulting in total of 270 kilobytes which allows efficient usage on mobile platforms. Performance results of our implementation on mobile devices can be found in Section 4.3.

4.2.4 A Note on Specification Compliance

In the TPM specification [112], the commands and structures for using the DAA functions of TPMs are defined. However, the experiments revealed that the TPM vendors have different interpretations of this specification. The library was tested with TPMs from Infineon, Atmel, Winbond, Intel, ST Micro and with the TPM emulator. Each of them has some deviations from the original specification, which result in different DAA implementations for the specific TPMs. For example, the TPM emulator stays close to the specification and uses the definition from the specification which says that parameters can be encoded as parameter length plus parameter. The Infineon TPM, however, requires parameter sizes strictly to be 256 bytes long unless otherwise specified. Moreover, the emulator does not correctly check parameter sizes of the commands. Although that does not appear to be a big problem, the emulator, respectively its code basis, is used in many implementations such as mobile TPMs or virtual TPMs for XEN [116]. Furthermore, the emulator does not close the join session after it has finished. The corresponding session parameters and reserved resources inside the TPM have to be flushed from the TPM manually by invoking a TPM_FlushSpecific command. According to the specification, the Join session and associated resources must be freed after execution of the command. A special case are TPMs from Atmel and Winbond. During execution of the experiments, it was not possible to invoke the DAA functions as these TPMs seem to deviate from the standard when it comes to verifying the issuer settings during the Join process. There are many more deviations that have to be taken into account when working with TPMs, especially when using the DAA functionality. Unfortunately, a continuative detailed discussion is out of scope of this document.

4.2.5 Performance evaluation

The DAA scheme involves complex mathematical computations, hence, it is of interest which performance can be achieved with currently deployed TPMs. Table 5.8 shows the

DAA Join	**Host**	**TPM**	**Host+TPM**	**Issuer**
TPM 1.2$_{INF}$	144,8 ms	56,7 s	56,8 s	712 ms
Emulator	144,8 ms	372,8 ms	517,6 ms	712 ms

Table 4.1: Performance of the Join Protocol with Intel TPMs

DAA Sign	**Host**	**TPM**	**Host + TPM**
TPM 1.2$_{INF}$	300 ms	37,7 s	38,0 s
Emulator	300 ms	67 ms	367 ms

Table 4.2: Performance of DAA signature creation with Intel TPMs

performance of the Join protocol as discussed in Section 4.1.2 on a Intel PC that is equipped with an Infineon TPM 1.2 (rev. 1.2.3.16). The performance values were measured on a HP Compaq dc7900 platform with an Intel Pentium Dual Core CPU E5200 2,5 GHz, a SUN 1.6 Java virtual machine (64 bit) and a Debian Linux operating system with a 2.6.30-1 Kernel (64 bit).

A verification of a DAA signature takes about half the time required for signature creation and does not require a TPM. All results presented in this Section represent the average measurement values of 100 executions of the *Join* and the *Sign* protocol. Performed tests have also been conducted with TPMs from ST Micro and Intel. The results are shown in Table 4.3 which clearly demonstrate the performance advantage of the Intel TPM. This advantage results from the different hardware used to host the TPM functionality. While the ST Micro TPM is basically a common smart card controller that is attached to the PC's motherboard via the LPC bus [60], the Intel TPM is located in the Intel motherboard chip-set (i.e. the Intel 82801JDO Controller Hub (ICH10DO)) itself [61].

Most of the time is consumed by the modular exponentiations and the parameter handling. As the TPM implementors want to save as much TPM resources as possible, the parameters obtained during the Join protocol are stored - encrypted with the EK - outside the TPM. For example, the DAA keys f_0, f_1, ν_0 and ν_1 have to be loaded into the TPM for each single DAA signature operation which takes about 1.5 seconds for each parameter on the ST Micro TPM. Each modular exponentiation requires about 2 to 8 seconds. The exact sequence of operations can be found in [112].

TPM	**ST Micro 1.2 (rev 3.11)**	**Intel 1.2 (rev 5.2)**
DAA Join	44.55 s	7.64 s
DAA Sign	33.38 s	4.66 s
Eval. Board	Intel DQ965GF	Intel DQ45CB
Operating System	Ubuntu v2.6.31-19 32 bit	Ubuntu v2.6.31.12-0.1 64 bit

Table 4.3: Comparison of the DAA Performance of different TPMs

4.3 The DAA Scheme on Mobile Platforms

In this Section, the efficiency and feasibility of the DAA scheme on state-of-the-art mobile handsets is investigated. Therefore, every single operation focusing on the sign, verify and join process is investigated and the performance results for each step of the DAA scheme is analyzed. The setup phase, which includes the computation of the issuer parameters and keys is excluded from the discussion. Before a client can execute the join process, the issuer has to compute and publish its public-key. Furthermore, the client has to obtain the key and a proof of the correct computation of the key as falsely computed key parameters could compromise the client's identity. For further investigations, it is assumed that a client can obtain a DAA issuer public-key and the group parameters from a trusted-third-party, which also performs the proof of correctness of these parameters. The parameters may be distributed via an X.509 certificate where the third party thereby guarantees the correctness of the parameters. A discussion of the issuer's keys and how they can be obtained by clients is given in Section 4.3.2.

For gathering experimental results, off-the-shelf devices, i.e. a Nokia 6131 (6131), a Nokia N96 (N96), a Nokia (E72), a Nokia 5800 and a Sony Ericsson P910 (P910) cell phone as well as an Lenovo T61 (PC) and Sony VAIO Notebook were used. The set of test platforms represents a profile of current mobile devices, ranging from low-cost entry level models to high-end smart phone systems. More details can be found in Sections 4.3.11 and 4.3.1.

The tests were conducted with two different cryptographic providers - the BouncyCastle crypto library [108] (BC) and the IAIK JCE MicroEdition [95] (IAIK). Both are free to use for research and educational purposes. Moreover, both libraries support the J2ME CLDC Java platform which is a typical and widespread embedded Java platform. A discussion of the differences between these two providers and their tremendous performance differences is given in Section 4.3.7.

4.3.1 Parameter setup

All parameters used for evaluation purpose use the parameter sizes suggested in [76], nevertheless, the tests were executed with moduli sizes of 2048 and 1024 bits which also have influence on the parameters (e.g. issuer key-pair) generated during the setup phase. The first step in the setup phase is the key-pair generation for the issuer.

The DAA *issuer's public-key* consists of the parameters n, S, Z, R_0, R_1 that have at maximum the size of the RSA modulus n which is in the test scenario 1024 or 2048 bits. Note that the parameters R_0 and R_1 could either be combined to a single but larger parameter R or distributed into smaller parameters R_0 to R_n, depending on the hardware used for computation. This re-distribution of the parameters also effects the private key parameters f_0 and f_1, which would then be smaller or larger. Discovering the optimum size for these parameters in order to get the most efficient computation performance for certain platforms and implementations, is subject to future investigations. The parameters R_0 and R_1 belong to the group of quadratic residues modulo n (QR_n) and are computed by the issuer as follows:

1. choose a random generator $g \in QR_n$
2. generate ramdom values $x_0, x_1, x_s \in [1, p'q']$
3. obtain the generator S: $S = g^{x_s} \mod n$

4. finally, compute $R_0 = S^{x_0} \mod n$ and $R_1 = S^{x_1} \mod n$

The second part of the issuer's key-pair is the *issuer's private-key*. This key is used to sign the client's DAA signing key-that is stored inside the TPM. The DAA Issuer's private key consists of the parameters *p, q, p', q', p"* where p and q are safe primes as suggested in [76], which have the property that $p = 2p' + 1$ and $q = 2q' + 1$.

The issuer computes the private- and public DAA key parameters as well as a proof that the public key parameters R_0, R_1, Z, S were computed correctly. For mobile clients, it is assumed that the clients do not verify the proofs computed by the issuer themselves. Typical mobile clients are considered to possess not enough computing power to do the complex computations required for verifying this proof. They rather delegate this task to a trusted third party (TTP) which verifies the proof and signs the key, thereby proving the correctness, authenticity and group affiliation of the key. Checking the group affiliation of the key is important as the issuer could generate a unique key for each device and put each device in a single group, therewith revealing the device's identity. This can be prevented by using a TTP which proves the identity and authenticity of the issuer's key. In order to test whether a client is a member of a certain group or not, the client could take a message m and create a signature $\sigma(m)$ with its private DAA signing key. By verifying $\sigma(m)$ with the issuer's public key, the client is able to prove whether it belongs to the group identified by the issuer's public key or not.

As previously discussed, a proof that the parameters of the issuer's public key are generated correctly, should be generated by the issuer according to the protocol specification of DAA. phones has enough computing power to perform these computations, efficiently. The feature was tested on the the mobile phone and a PC and the results are shown in Table 4.4.

Validating this proof is very time consuming and has to be done for a key only once. The typical scenario is to delegate the validation of this proof to a trusted-third party (TTP) which issues a certificate on the key after successfully validation.

In order to prove that $Z, R_0, R_1 \in \langle S \rangle$ are correct $3x160$ calculations are required. Each of these calculations has a modular exponentiation with the power $\in [1, p'q']$ and also a multiplication (if c_i is 1).

The results in Table 4.4 show that the validation timings of this proof on a Nokia 5800 XpressMusic device is equipped with a AEM11 CPU running at 434 Mhz and a Sony Vaio notebook with a Intel Core 2 Duo T7250 CPU running at 2GHz on Windows and SUN Java 1.6.

The performance might be acceptable on a common notebook.

Device	Nokia 5800	Sony Vaio
Timing	2399.237s	99.406 s

Table 4.4: Proof Verification Timings (n=2048)

However, the performance of the Nokia smart phone clearly puts the validation by a TTP into favor.

4.3.2 The Join Process

Although DAA allows a client to create a signature on behalf of a group, the client must *join* the group prior to creating signatures on behalf of that certain group. Within this join phase, all required parameters for the client are created and negotiated with the issuer. This also includes the credentials that are issued by the issuer for the keys in the TPM and the credentials that are stored on the host.

Furthermore, the client and its TPM have to be authenticated in order to prevent arbitrary clients to enter the group without permission. A method to authenticate a TPM during the join phase is given in [76] (p. 143-177).

In order to simplify the operations, the rogue tagging parameters and computations were removed from the implementation. Nevertheless, the implementation is implemented in accordance with the BCC05 protocol as discussed in subsection 4.1.2.

Note that the measurements were made with both crypto libraries where the abbreviation (BC) marks the result obtained from the BouncyCastle library and (IAIK) marks the results from the IAIK JCE-ME library.

Join	6131 (BC)	P910 (BC)	PC (BC)	6131 (IAIK)	PC (IAIK)
n=2048	363.32 s	149.55 s	0.67 s	158.614 s	0.24 s
n=1024	82.48 s	34.07 s	0.18 s	30.57 s	0.05 s

Table 4.5: Join process performance results (client side)

The join step includes very computational intensive modular exponentiations. The results in Table 4.5 show the average computational effort for the clients. The issuer performance is not included in this table as it is assumed that the issuer side computations are executed on a powerful server machine. Nevertheless, the server side computation takes several milliseconds where the creation of the Camenisch-Lysyanskaya signatures proved to be the most time consuming part. A discussion of creating Camenisch-Lysyanskaya signatures is addressed in Section 4.3.3. The tests were not influenced by effects of network latency as, for the tests, issuer and host components were executed on the same platform and only the host specific operations were measured.

4.3.3 Camenisch-Lysyanskaya Signatures

As discussed in the previous section, the issuer computes a blind signature on the private key of the DAA client. For efficiency reasons, this signature is a Camenisch-Lysyanskaya signature and is created during the join process. The performance results for a single CL signature computation can be taken from Table 4.6.

CL Signature	6131* (BC)	P910* (BC)	PC (BC)	PC (IAIK)	N96* (IAIK)	E72* (IAIK)
n=2048	62.29 s - 81.60 s	31.67 s	0.35 s	0.17 s	7.53 s	3.10 s
n=1024	34.23 s	8.21 s	0.14 s	0.07 s	-	-

Table 4.6: Camenisch-Lysyanskaya signature computation performance results

This particular table shows the average performance values on different platforms. However, these values vary strongly - this is due to the fact that for each signature, a new prime e of 368 bits length has to be found. Depending on how fast this prime can be found, the CL signature is computed faster or slower. The values marked with * are the performance values obtained on the mobile devices. These values are added to the table for the interested reader, as the computation of the CL signature is usually done on the serverside and, therefore, only the PC performance is of interest for typical application scenarios of the DAA scheme.

4.3.4 DAA signature creation

The DAA signature computation, compared to an RSA or ECC signature, is a more complex signature. In [76], the authors of the original DAA scheme [13] propose a modified version which reduces the computational efforts on the host side. This sign algorithm also differs from the one discussed in Section 4.1.2.

The computation of the DAA signature in the implementation of the modified signing algorithm works as follows:

1. The host computes: $T = A * S^w$ with $w \in \{0,1\}^{l_n + l_\phi}$
   Host and TPM compute the "signature of knowledge":
2. The TPM computes: $\tilde{T}_t = R_0^{r_{f_0}} R_1^{r_{f_1}} S^{r_\nu} \mod n$ with r_{f_0}, r_{f_1} of size $l_f + l_\phi + l_H$ bits and r_ν with length $l_n + l_\phi + l_H$.
3. The host computes: $\tilde{T} = \tilde{T}_t T^{r_e} S^{r_{\bar{\nu}}}$ where $\tilde{T}_t$ is the input from the computation process that was performed in the TPM. The random parameter r_e is of size $\{0,1\}^{l'_e + l_\phi + l_H}$ whereas $r_{\bar{\nu}}$ is of size $\{0,1\}^{l_e + l_n + 2l_\phi + l_H + 1}$ bits.
4. Moreover, the host computes: $c_h = H(n \| R_0 \| R_1 \| S \| Z \| \| T \| \tilde{T} \| n_\nu)$
5. The TPM selects a random $n_t \in \{0,1\}^{l_\phi}$, computes $c = H(H(c_h \| n_t) \| m)$ and $s_\nu = r_\nu + c * \nu$, $s_{f_0} = r_{f_0} + c * f_0$ and $s_{f_1} = r_{f_1} + c * f_1$
6. The host computes $s_e = r_e + c * (e - 2^{l_e - 1})$ and $s_{\bar{\nu}} = s_\nu + r_{\bar{\nu}} - cwe$
7. Finally, the host assembles the signature $\sigma = (T, c, n_t, s_{\hat{\nu}}, s_{f_0}, s_{f_1}, s_{f_e})$

The computation of the signature is separated between host and TPM and includes three modular exponentiations in the TPM and three plus one multiplications on the host side. In addition, the TPM has to compute s_ν, s_{f_0}, s_{f_1}, which includes three additions and multiplications plus the proofs of the host credentials s_e and $s_{\bar{\nu}}$.

Table 4.7 shows the average performance results of the implementation when creating DAA signatures.

DAA sign	**6131 (BC)**	**P910 (BC)**	**Desktop (BC)**	**6131 (IAIK)**	**P910 (IAIK)**	**PC (IAIK)**	**N96 (IAIK)**	**E72 (IAIK)**
n=2048	239.80 s	81.50 s	0.34 s	130.42 s	52.08 s	0.23 s	6.56 s	3.10 s
n=1024	68.22 s	24.68 s	0.11 s	26.60 s	16.81 s	0.07 s	-	-

Table 4.7: Performance comparison for creating DAA signatures

The computations on the host side in this scheme differ from the original scheme. In BCC'04, the host computes:

1. $T_1 = Ah^w \mod n$ and $T_2 = g^w h^e (g')^r \mod n$ with w, r integers $\in \{0,1\}^{l_n + l_\phi}$ where g, g', h are parameters from the public key.

2. The host picks the random numbers:
   $r_e \in_R \{0,1\}^{l'_e+l_\phi+l_H}$, $r_w, r_r \in_R \{0,1\}^{l_n+2l_\phi+l_H}$ $r_ee \in_R \{0,1\}^{2l_e+l_\phi+l_H+1}$, $r_{ew}, r_{er} \in_R \{0,1\}^{l_e+l_n+2l_\phi+l_h+1}$

3. and computes: $\tilde{T}_1 = \tilde{T}_{1t}T_1^{r_e}h^{-r_{ew}} \mod n$, $\tilde{T}_2 = g^{r_w}h^{r_e}g'^{r_r} \mod n$ $\tilde{T}'_2 = T_2^{-r_e}g^{r_{ew}}h^{r_{ee}}g'^{r_{er}}$ mod n where $\tilde{T}_{1t}$ is computed in the TPM. Note that the computations done in the TPM are the same in BCC'04 as in BCC'05.

4. Moreover, the host has to compute:
   $s_e = r_e + c * (e - 2^{l_e - 1})$, $s_{ee} = r_{ee} + c * e^2$, $s_w = r_w + c * w$
   $s_{ew} = r_{ew} + c * w * e$, $s_r = r_r + c * r$, $s_{er} = r_{er} + c * e * r$

As a result, the signatures of BCC'04 and BCC'05 differ:
$\sigma_{BCC'04} = ((T_1, T_2), c, n_t, s_{\hat{\nu}}, s_{f_0}, s_{f_1}, s_{f_e})$ and $\sigma_{BCC'05} = (T, c, n_t, s_{\hat{\nu}}, s_{f_0}, s_{f_1}, s_{f_e})$

When comparing the two schemes, one can easily see that BCC'05 is the faster scheme. While in BCC'04, the host has to compute $\tilde{T}_1, \tilde{T}_2$ and $\tilde{T}'_2$, in BCC'05 the host is only required to compute $\tilde{T} = \tilde{T}_t T^{r_e} S^{r_{\hat{\nu}}}$ which is a reduction of seven modular exponentiations.

4.3.5 DAA Signature Verification

In this section, the basic steps for verifying DAA signatures are discussed. The verification process includes five modular exponentiations plus reductions (including one inversion) and the hashing of several parameters. It can be performed without the involvement of a TPM.

DAA verify	**6131 (BC)**	**P910 (BC)**	**PC (BC)**	**6131 (IAIK)**	**P910 (IAIK)**	**PC (IAIK)**	**N96 (IAIK)**	**E72 (IAIK)**
n=2048	97.03 s	38.23 s	0.18 s	64.27 s	22.10 s	0.11 s	4.29 s	1.99 s
n=1024	24.91 s	11.72 s	0.05 s	16.96 s	8.24 s	0.04 s	-	-

Table 4.8: Performance comparison for verifying DAA signatures

Table 4.8 gives an overview of the obtained performance results. Special notice should be given to the DAA verification on the desktop platform. The measured 113 ms are not detectable by a human user but could be of interest on server platforms that have to deal with many concurrent verification processes.

4.3.6 Implementation details and discussion

The DAA scheme makes heavy use of modular exponentiations [19]. These modular exponentiations can be reduced to modular multiplications, reductions and squarings, hence, the overall performance depends on the number of executed multiplications, reductions and squarings . In order to determine the average number of operations and their performance, a detailed analysis of a single exponentiation was done. Table 5.1 shows the average number of operations required for calculating S^{r_ν}, where r_ν has the size of 2737 bits, with respect to the proposed parameters in the DAA scheme [76]. This value was chosen because it is the largest coefficient involved in computing a DAA signature - all other exponentiations use smaller values.

	meantime PC	meantime P910	operations
multiply	0.16 ms	19 ms	389
square	0.12 ms	15 ms	2737

Table 4.9: Performance of a single modular operation

With 389 multiplications and 2737 squarings, the overall time for this specific modular exponentiation is about 40 seconds on the P910 device. Still, the single modular operations are over hundred times slower compared to the desktop platform. Hence, improving the performance of these operations is required. A possible improvement could be achieved by moving these basic operations to the native platform, a detailed analysis of this approach is, however, out of scope of this document and is left to future investigations.

The performance was measured with the IAIK-JCE library which is faster than the BC library. Reasons for this are given in the following section. The modular multiplications, reductions and the squaring were implemented according to the algorithm specifications in [5].

4.3.7 Modular arithmetic in BouncyCastle and IAIK JCE-ME

For the analysis, two cryptographic libraries that support the Java 2 MicroEdition platform were used. This Java platform is the most widely spread platform amongst mobile phones. However, there is a tremendous difference according to the performance of modular arithmetic between the two libraries. The speed advantage of the IAIK-JCE library stems from the different implementation of the modular exponentiation. The IAIK library uses the *sliding windows* technique which involves precomputation of certain values used in the multiply and square operations of the modular exponentiation. This technique provides a performance advantage over the basic approach that is used in the BC library where multiply, square and reduce are performed without precomputations [18].

4.3.8 Random number generation on J2ME platforms

The DAA scheme requires a great amount of random numbers for every executed zero-knowledge proof. While generating random numbers on desktop platforms is not a great challenge, generating random numbers or an appropriate seed in a J2ME/CLDC environment is problematic. Desktop platforms have various sources of entropy e.g. user input, disk movements or hardware based true random number generators. The J2ME/CLDC environment does not allow applications to execute native code or access platform features for generating entropy. A common way to seed the random number generator is to use the current date and time. However, this proceedure should be investigated for exploiting attacks on the DAA implementation. The DAA implementation uses a pseudo random function based on SHA-1 to generate the random numbers.

4.3.9 Hashing of bignumber objects

Bignumber types and *bignumber* objects are used to represent the large (prime) numbers used in cryptography e.g. for RSA primes and ECC curve parameters. They can be found as methods in programming languages like C (see Openssl library) or classes in Java (java.math.BigInteger). Basically, they are an abstraction of a byte array, containing the value of such a long integer. The DAA scheme depends on calculating hashes of a high number of such bignumbers. Consequently, an efficient and standardized way for hashing bignumbers should be used in order to avoid bottlenecks and provide interoperability. For this reason, the implementation supports the bignumber hashfunction as defined in P1363 [57] which converts a biginteger into a fixed size output bit-string in big-endian order.

4.3.10 Prime number generation

The Camenisch-Lysyanskaya signature scheme requires prime numbers e that lie in $[2^{l_e-1}, 2^{l_e-1} + 2^{l'_e-1}]$ for each signature operation. The DAA scheme does not specify which prime number generator to use. For the tests, the prime number generator defined in [79] was used. This generator is based on a sieve technique that requires a random starting point. Based on the selection of this starting point, the computation time varies strongly. Consequently, the computation time of the CL-signatures varies.

4.3.11 Test environment

In this section, the basic setup of the test environment and test tools is discussed. In order to answer the question whether anonymous credentials as proposed in the DAA scheme can be efficiently used on Java platforms or not, a library that provides all required features (i.e. parameter setup, join operations on host and server, sign and verify modules) for working with DAA credentials was developed.

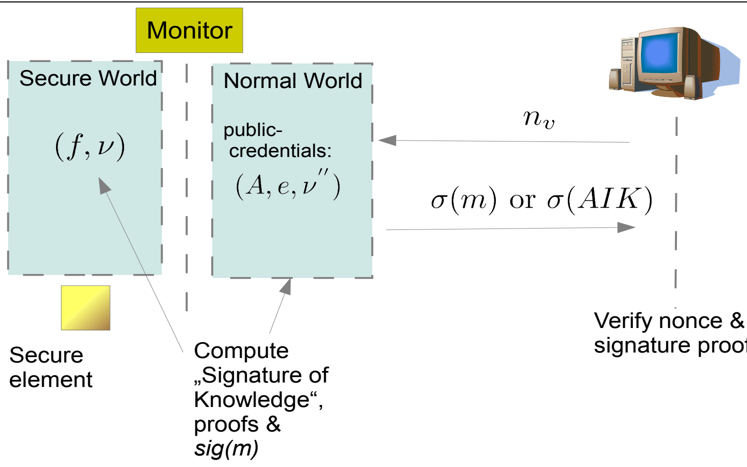
In the test setup, issuer and host are running on the same device. This setup eliminates measurement derivations from network latency but has the drawback that no precise statement about the memory consumption of the implementation can be given. For the tests, a TPM/MTM software emulation that provides the required DAA features with respect to the BCC05 scheme was used. Because of the modifications, it differers from the DAA protocol specified by the TCG. However, this only affects the host side of the protocol so that existing TPMs can be used. The design of the implementation allows the use of different kinds of TPMs. The TPM may be a dedicated micro controller, a software process running in an L4 compartment or isolated TrustZone environment or a smart card based secure element. For the tests, software TPMs and platforms where mobile TPMs may be implemented solely in software as discussed in [111] and [110] were used. Protection of the DAA keys depends on the platform. On the low-cost test devices (e.g. Nokia 6131), protection can only be guaranteed by the security properties of the Java virtual machine (i.e. process isolation, sand-box execution) and the Nokia operating system. On the high-end test devices, protection is achieved by the ARM TrustZone processor extension [6] as discussed in [35]. In this specific case, issuer and DAA host computations are executed in the *non-secure world* and the TPM-DAA computations are executed in the *secure world*. To be more precise, they are executed on a Java VM which is executed inside the ARM TrustZone protected environment. For further details, refer to [35].

The results show the average computational effort for host (plus TPM emulation) and issuer. The single *join*, *sign* and *verify* processes were executed ten times in sequence on each platform in order to eliminate class loading and thread scheduling effects arising from the Java environment.

Figure 4.3 shows the principle design how the TrustZone (TZ) security extension may be used for protecting DAA keys. While the secret keys are stored and processed inside the

TZ, other DAA credentials may be used for computation in the non-secure world. Both, the TZ and the mobile host compute the signature of knowledge. This split basically represents the TPM and PC platform split on desktop systems.

Figure 4.3 Architecture Overview for DAA on TrustZone enabled Devices


Monitor
Secure World
(f, ν)
Normal World
public-credentials:
(A, e, ν'')
n_v
$\sigma(m)$ or $\sigma(AIK)$
Secure element
Compute „Signature of Knowledge“, proofs & *sig(m)*
Verify nonce & signature proofs

Other designs could include a secure element. As a consequence, five different implementations are feasible. First, computations are split between TrustZone and non-secure world as depicted in Figure 4.3. Second, all computations may be done inside the TrustZone environment. Third, computations are split between SE and TZ. Fourth, computations are done solely in the SE and fifth, the computations are split between non-secure world.

4.3.12 Revocation

The revocation mechanism is a crucial part to remove rogue TPMs or MTMs. A revocation check is performed at two stages of the protocol, first during the join protocol in order to prevent revoked platforms to obtain DAA credentials and second during signature creation and verification. The issuer or group manager has a black-list of all revoked devices. As the revocation check depends linear on the number of revoked devices (i.e devices on the list) test were conducted with different black-list sizes.

Test setup

The test setup includes a Vaio VGN-NR11Z/S notebook as group manager and a Nokia 5800 Express Music cell phone as client. Table 4.10 shows the properties of the test devices.

	Device	CPU	Freq.	RAM	OS
Client	Nokia 5800	ARM 11	434 MHz	512 MB	Symbian 60v5
Issuer	Sony Vaio VGN-NR11Z/S	Core 2 Duo T7250	2.0 GHz	3GB	Windows XP

Table 4.10: Rogue Tagging Test Devices

Rogue Detection Performance Results

The join protocol was executed via a wireless data link, hence, the performance values in Table 4.11 include the network communication overhead. However, as the majority of time is used for computation, the network delay only contributes a minimum to the overall time.

Modulus length n [bit]	1024	1536	2048
Join time [s]	10.85	19.56	32.52

Table 4.11: Average Join Protocol Execution Time with empty Blacklist

These results were gained when performing joining with an empty blacklist.

Table 4.12 shows the timing results with various black-list sizes.

Rouge TPMs	100	1000	2000	5000	10000
Time [s]	33.57	46.67	59.76	103.35	187.20

Table 4.12: Rogue Detection Execution Times with varying Black-list sizes (n=2048)

In cases where the mobile takes over the role of the group manager, for example, when defining ad-hoc groups or when simply verifying DAA signatures, the mobile has to do the computations for revocation checking.

Table 4.13 shows the rogue detection performance on the Nokia 5800 device.

Rouge TPMs	200	500	1000	2000
Time	79.95s	179.47s	352.11s	699.06s

Table 4.13: Rogue Detection on the Nokia cell phone

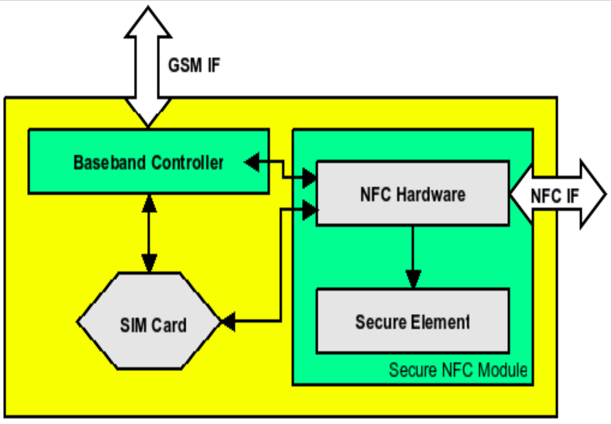
The impact of the revocation check on the overall signature verification time is very high especially with larger lists. The total verification time for $n = 2048$ and 200 rogue TPMs on the list requires 10.314 for signature validation plus 79.95 for revocation checks makes a total of 90.26.

A definite suggestion is to delegate the work of rogue detection to a third-party. A simple solution could be to modify the widely used online certificate status protocol (OCSP). Instead of sending the serial number(s) of X.509 certificates, the protocol could be used to transport the revocation check information of the TPMs.

4.4 Analysis of DAA on NFC enabled devices

The design of NFC enabled devices typically includes a secure element that is connected to the mobile host platform as well as to an NFC communication element (see Figure 4.4).

Figure 4.4 Connection of NFC Module, Secure Element and Mobile Platform



However, all existing approaches ([11], [93], [64]) clearly show that current smart-cards do not provide sufficient performance for computing discrete logarithm based DAA signatures for use in real-world scenarios. Hence, it is inevitable to include a *host* that contributes the computation of such a signature. This can either be done by providing a host with adequate processing capabilities or by providing a host that controls and manages the pre-computation of such signatures. Idle phases of the host or the SE can be used to generate RSA key-pairs - which are addressed as ephemeral authentication keys (EAKs) from now on - that can then be certified by a DAA signature. Mobile phones, equipped with either SIM-cards or secure elements provide the ideal platform for such an approach.

In this Section, two approaches how anonymous signatures can be computed on mobile devices that are equipped with secure elements (SEs) are investigated. In the first approach, the signature is computed entirely in the SE. In this case, the application on the host initiates the pre-computation of the keys and the signature creation. The algorithm listed in subsection 4.3.4 is executed entirely inside the SE. The pre-computation of the ephemeral key-pairs can be executed on the card without further involvement of the host. In the second approach, the signature computation is partially computed inside the TPM and partially on the host. Details of the implementation and performance results can be found in Section 4.4.2.

In addition, the host may be used to store specific DAA group credentials. Similar to the DAA scheme on desktop system where the PC platform is used to store DAA credentials, these credentials may be loaded and un-loaded from the SE and stored on the mobile host.

However, in both approaches, the rogue tagging value cannot be computed in advance as it depends on a *basevalue* created by the verifier. Moreover, when looking at the DAA sign algorithm, it is obvious that the computation of the signature relies on the computation of the pseudonym $\tilde{N}_V$ which is used for the computation of c The consequences of this dependency are that in case of no revocation, the EAKs can be computed and certified in advance. In case of revocation, the EAKs can be computed but the certification may only be pre-processed to the computation of c which further required that all temporary values of the algorithm must be stored until the final computation of the signature.

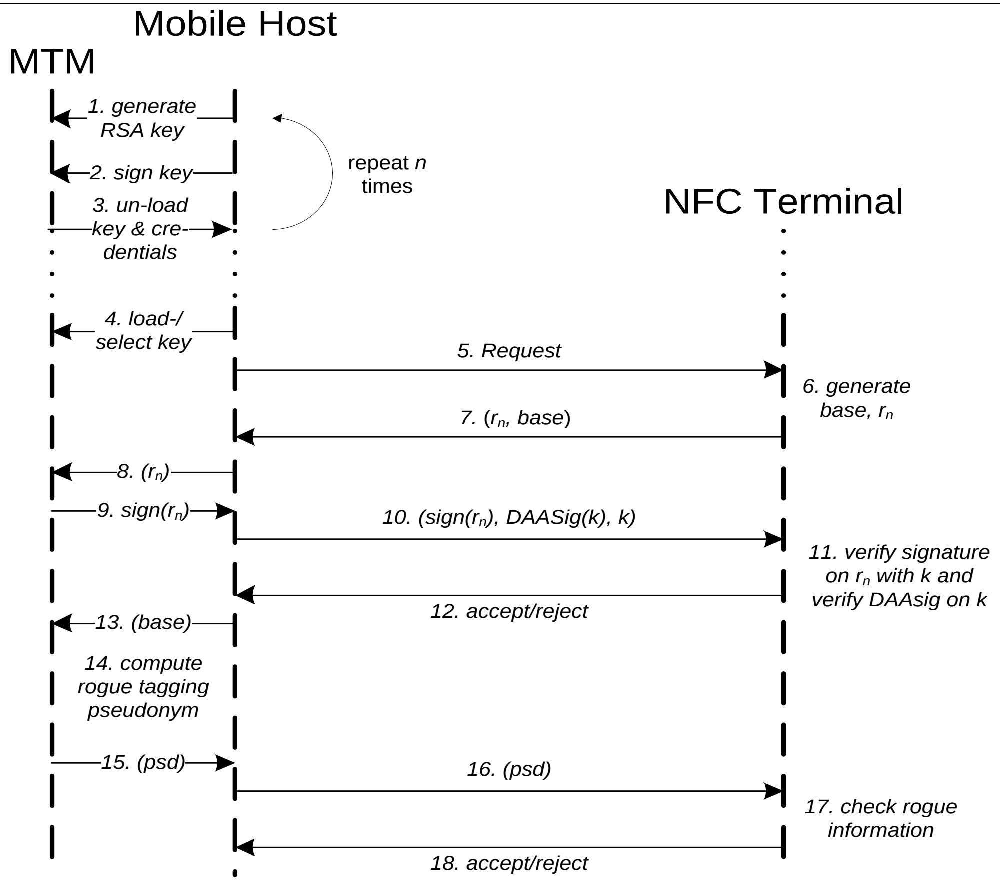
Although a DAA signature, according to the specification [112], contains the actual signature and the computed rogue tagging value, both computations are rather independent cryptographic operations (only the computed hash c - see 4.1.2, step 6 of the algorithm , contains the rogue parameters). Hence, they can be computed separately.

4.4.1 An anonymous authentication scenario

Figure 4.5 shows the application of the approach in a basic authentication scenario.

The mobile platform pre-computes a set of n EAK key-pairs (steps 1-3), certifies the public parts with DAA signatures[1] and stores the private parts of the keys either in the EEPROM of the SE or encrypts it and un-loads it to the host device. The public-keys and their credentials (i.e. the DAA signatures) are stored on the mobile platform. The same is true for the DAA credentials (f, ν_0 and ν_1, R_0, S_0, S_1). By loading different credentials, the TPM may create DAA signatures on behalf of different groups it has joined before.

Figure 4.5 Authentication Protocol Sequence



A user can now use these keys and the NFC module on the phone to prove his authorization against an NFC terminal without revealing his identity. Before sending a request to the terminal, the mobile loads a certified EAK key into the TPM, either from the EEPROM or from the mobile device (steps 4-5). The terminal computes and sends a nonce

[1]In Trusted Computing enabled application scenarios, the standard exponent 65537 is used. Hence, only the RSA modulus is signed and transmitted when required.

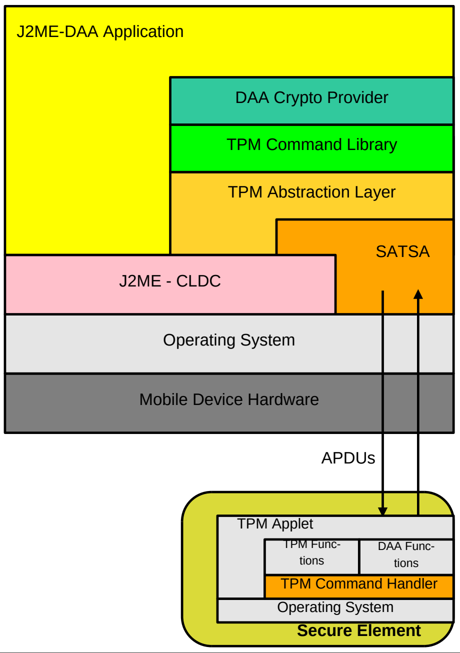
r_n and $base$ for rogue detection to the mobile (steps 6-7). The mobile forwards r_n to the TPM which signs r_n with the previously loaded EAK key. The mobile device forwards the signature $sign(r_n)$ on r_n, the public EAK key k and the DAA signature $DAASig(k)$ on this key to the terminal (step 10). The terminal verifies $DAASig(k)$ with the issuer's public-key (step 11) and continues the protocol if the verification succeeds. In steps 13-15, the TPM computes the pseudonym $psd = H(base)^{\Gamma - 1/\rho} \bmod \Gamma$ which is verified by the terminal as discussed in [76].

If all verifications succeed, the terminal has the information that the requestor is a member of a certain group - namely the group represented by a certain issuer and its public-key - and that the used TPM is not on a list of compromised TPMs. However, the terminal has no information about the identity of the platform or its owner.

4.4.2 Implementation aspects

For the experiments, a Nokia 6212 NFC mobile phone was used. This phone is equipped with a Giesecke & Devrient SmartCafe smart-card as SE. The secure element based TPM uses this smart-card which provides a JCOP41 v2.2.1 runtime environment. The TPM commands and the DAA computations are handled by a JavaCard applet that is installed on the smart-card. Figure 4.6 shows the concept. The host application uses a TPM command library to issue commands which are sent to the SE via application protocol data units (APDUs).

Figure 4.6 Architecture Overview



The host part is implemented as a Java2MicroEdition (J2ME) [84] application that allows the installation of mobile applications on the phone. Moreover, it is taken advantage of the Security and Trust Service API (SATSA) [104] respectively of JSR 257 the *Contactless Communication API* [83] which allows the application to communicate with the card applet via APDUs. This approach, however, requires that the J22ME application is signed with a code signing certificate from Versign or Thawte.

For the DAA support in the TPM, several TPM commands and structures as defined in [112] as well as support for different algorithms are required. The JavaCard 2.2.1 environment does not provide support for implementing cryptographic protocols. The ideas from [93] and [11] concerning algorithm implementations on JavaCard are followed. For example, the modular exponentiation can be computed via the RSA cipher algorithm and modular multiplication via transformation into a binomial form, $((a * b) \bmod n = \frac{(a+b)^2 - a^2 - b^2}{2} \bmod n)$, the *hmac* algorithm has to be implemented in Java, reducing the overall performance when computing the integrity check of incoming TPM commands.

The minimum implementation of the DAA scheme requires the following TPM commands and TPM structures on the host and TPM side:

1. a protocol for authorization: TPM_OIAP plus session handling,
2. the TPM_DAA_Join() command
3. the TPM_DAA_Sign() and TPM_DAA_Sign_Init() commands
4. TPM_FlushSpecific() and TPM_Terminate_handle commands for aborting the computation during one of the stages and freeing the resources inside the TPM.
5. TPM_DAA_Issuer_Struct. This structure holds the issuer parameters.
6. two containers for symmetric keys

For unloading the RSA key-pairs, the corresponding DAA signature and the DAA credentials, the TPM generates two symmetric keys k_0, k_1, one for encrypting the data and one for computing an integrity check on it. The TCG specification allows to use symmetric or asymmetric encryption for this purpose. In the approach, symmetric cryptography for encryption and - this is different to the TCG specification - a symmetric key for integrity protection to detect modifications of the encrypted authentication keys and DAA parameters when they are stored on the device is used.

4.4.3 The pre-computation step

The TCG specifies two commands *TPM_DAA_Join* and *TPM_DAA_Sign* which are executed repeatedly in different stages [112]. For simplicity reasons, these stages are reduced to one single stage.

Table 4.14 shows the measured performance values. The first row shows the values when the computation is split between host and TPM. The first column shows the required time for command handling which includes the computation and verification of *hmac* integrity checks on the command data and its transmission to the TPM. The second column shows the time consumed for computing the host part and the third column shows the time required for computing the TPM part inside the SE. The last column shows the overall result of all single operations.

Table 4.14 shows a slight performance advantage when computing the entire DAA signature in the TPM[2]. For the first approach, the JavaCard applet that includes the TPM command handler, the cryptographic algorithms and the DAA functionality, requires about 5284 bytes in the EEPROM of the card.

[2]For the interested reader, a DAA signature computation on an Infineon 1.2 TPM requires approximately 38 seconds.

Command handling	**Host**	**Secure Element**	**Total**
1,1 s	23,8 s	4,8 s	29,7 s
1,4 s	-	26,0 s	27,4 s

Table 4.14: Performance comparison of the DAA sign approaches

Note that all performance measurements are average values that were estimated by 100 executions of the single operations.

4.4.4 The NFC authentication step

For the actual authentication via the NFC channel, the certified EAK-keys from the precomputation step can be used. As shown in Figure 4.5, the terminal sends a nonce to the mobile/TPM which basically applies an RSA signature according to PKCS#1.5 [73] on the nonce which takes approximately 1 second.

Command Handling	**Nonce singing**	**Rogue Tagging**	**Total**
1,0 s	1,3 s	1,1 s	3.4 s

Table 4.15: Performance of the authentication process

Moreover, the TPM computes the rogue tagging parameter which is basically a single modular exponentiation which also takes approximately 1 second. Hence, the total time required for authentication is 3,4 seconds.

Parameter lengths In the prototype implementation, the following parameter lengths are used:

l_n	l_s	l_e	l_f	l_ν
2048 bits	1024 bits	368 bits	160 bits	2536 bits
l_ϕ	l_{r_w}	l_{r_ν}	l_{r_f}	l_Γ
80 bits	2128 bits	2228 bits	400 bits	2048 bits

Table 4.16: Parameter lengths in number of bits

4.5 Conclusion

This Chapter focuses on the investigation of the DAA scheme on different platforms - especially on mobile platforms with specific security devices. It further investigates how Trusted Computing based technologies and embedded security mechanisms can be used for anonymous authentication for NFC and RFID applications.

Two approaches, the first splitting the computation of such DAA signatures between a resource constrained TPM and a more powerful host platform and the second, using solely the MTM located in a security device to compute the entire DAA signature are analyzed.

In addition, the TLS client authentication use-case was investigated to measure the performance that can be achieved with currently available TPMs. Although the performance of TPMs like the ST Micro or the Infineon TPM is rather slow, the fast Intel TPM allows the use of anonymous credentials in an efficient way.

Fourth, a concrete scheme is provided how an anonymous credential systems based on TPMs can be integrated into the Java cryptography architecture architecture and how they can be used for anonymous authentication. Moreover, a design was proposed that allows the use of the DAA protocol on mobile phones, taking advantage of state-of-the-art security components.

For generating experimental results, off-the-shelf mobile phones were used that are equipped with secure elements to host the TPM/MTM functionality and which are connected to NFC modules, allowing a practical implementation of the proposed architecture.

The current implementation of the library shows that anonymous credentials can be efficiently implemented and used for the Java desktop platform. However, when used on mobile platforms, a pure Java implementation is only reasonable if the Java execution acceleration is supported. This could either be done in hardware, like in the case of the ARM processor extension Jazelle [82] or in software via Just-in-Time (JIT) compilers.

Another solution could be to provide native code support - the J2ME/CDC platform allows native calls from Java applications. Thus, the modular arithmetic operations, which are a great bottleneck to cryptography when computed in pure Java could be accelerated. Moreover, hashing of bignumber objects is done in great numbers as discussed in Section 4.3.9. These hash operations could also be moved to the native operating system and native device to speed up the DAA operations. Therefore, future investigations should include native support for modular arithmetic.

A different way to improve the performance could be to delegate the computation to a dedicated crypto device, for example, the sim card or secure elements like the one discussed in [30].

The performance of the DAA protocol could also be improved by using other cryptographic primitives like a DAA scheme based on elliptic curves or pairing based cryptography as discussed in [14].

Hardware based acceleration, especially for embedded devices, could also be implemented by extending a CPU's core. Extensions to the instruction set as discussed in [49] support fast computation of long numbers, i.e. modular addition, modular exponentiation and multiplication operations.

4.5.1 Future work and improvements

Future investigations should include DAA schemes based on elliptic curves or pairing based cryptography as discussed in ([72], [21]). ECC based schemes clearly show a performance advantage over the RSA based variant. Although support for ECC is provided by JavaCard vendors, adequate support for developing complex DAA protocols based on ECC is not yet available on current JavaCard platforms. Another interesting aspect for mobile devices is power consumption. How much power is drained from a device's battery depends strongly on the executed operations. Consequently, an analysis of the power consumption when computing DAA signatures is of great interest for future investigations.

Also of interest are *secure* implementations of DAA algorithms which are robust against interferences like attacks with laser light from adversaries outside the security devices. While the basic cryptographic operation of addition and multiplication can be implemented in a secure and robust way the question is still unanswered whether this can be done for entire DAA algorithms.

Chapter 5

Improvements and Alternative Approaches for establishing Anonymity Protection for Trusted Platforms

5.1 Introduction

In the previous Chapter 4, a discussion of existing privacy enhancing technologies (PETs) was given. In this Chapter, attention is turned to improvements and alternative approaches how anonymity protection can be established.

One of these improvements addresses the Java based DAA cryptosystem implementation which is bound to the constraints of embedded processors. DAA is based on discrete logarithm operations that involve numerous modular exponentiations. The most popular algorithm for modular exponentiation is the Montgomery exponentiation based on sliding window technology. This technology offers several configuration options in order to get the best trade-off between the amount of pre-computations and multiplications that are required for different exponentiation operands. Consequently, the optimum configuration and best parameters for receiving the highest performance gain are of interest. Therefore, different approaches for improving the performance of modular exponentiation with respect to the DAA scheme on Java enabled platforms are of interest. In particular, the optimal parameter setting for the Montgomery exponentiation has to be identified and furthermore it has to investigated how natively executed modular multiplications and modular reductions, with respect to a minimum of native code involved, can be integrated to improve the performance of mobile Java applications. Experimental results show that the optimal setup of the Montgomery algorithm for a single modular exponentiation differs from the optimal setup used for the combination of all operations and operands used in the Direct Anonymous Attestation scheme.

Modern virtual machines (VMs) move time critical operations to the native part of the host platform. Therefore, it is interesting to know which performance improvement can be gained with native code execution support. In commercial VMs, operations like modular exponentiation used, for example, in the RSA cryptosystem are executed natively which is entirely transparent to the Java application. Unfortunately, these RSA implementations can not be used to compute exponentiations as the apply paddings per default. A "'raw"' mode where only a modular exponentiation/reduction without padding is executed are virtually non-existent on commercial phone-VMs. As a consequence, the test implementations in this Chapter take advantage of the Java Native Interface (JNI). However, a performance gain is not possible for any cases where data is processed natively. In some cases the overhead of shifting the data in and out of the VMs outweighs the performance gain especially when VMs are used that employ Just-in-Time (JIT) or Dynamic-Ahead-Compilers (DAHC) compilers. As there are many parameters to shift in DAA, it is of interest to know whether there is a performance gain or not.

Another topic in this Chapter is the design of alternative anonymisation algorithms. Two different anonymisation schemes for Trusted Computing platforms have been proposed by the Trusted Computing Group - the PrivacyCA scheme and the Direct Anonymous Attestation scheme. These schemes rely on trusted third parties that issue either temporary one-time certificates or group credentials to trusted platforms which enable these platforms to create anonymous signatures on behalf of a group. Moreover, the schemes require trust in these third parties and the platforms have to be part of their groups.

Although it is possible for the TPM vendor to play the role of the DAA issuer, it is practically not possible to do so without providing extra services (i.e. a DAA infrastructure). Such an infrastructure would include the issuer component where the clients can obtain their credentials. Theoretically, it might be possible to ship TPMs with pre-installed credentials, however, in practice this is not the case. Furthermore, a revocation facility and a trusted-third-party service that checks and signs the issuer parameters is required - remember, before loading the signed issuer parameters into the TPM, their integrity and authenticity has to be checked which is done partly by the TPM and partly by the host platform. The signing party which signs the DAA parameters could also be the vendor, however, at the cost of an extra service and additional processing steps on the platform required for validating the parameters.

However, the approaches proposed so-far (i.e. DAA and both approaches rely on a trusted third party which is either a PCA or an issuer or group manager. This idea is acceptable as long as the interacting platforms are part of such a group. The group and the corresponding services could be established by a company or an official government institution. However, if you think of your private PC at home, which of these services would provide protection for your home platform? A company's anonymization service will unlikely provide such a protection for the company's employees' private computers in order to protect their transactions in their spare-time. A private computing platform would have to rely on either paid anonymization services which would add extra cost to the platform's owner in order to receive anonymity protection or it would have to rely on free and open anonymization services where a platform and its user have to trust that the information sent to the service is dealt with correctly. However, the platform owner has no influence and no hold on the correct treatment of the information and the availability of the service. This raises the general question of how two platforms that are not part of one of such groups can establish a connection and stay anonymous at the same time. Hence, it would be reasonable to have an anonymization scheme that does not rely on such a trusted third party like a PCA or DAA issuer and that does not produce extra costs for clients.

However, there are certain use-cases where group affiliation is either not preferred or cannot be established. Hence, these existing schemes cannot be used in all situations where anonymity is needed and a new scheme without a trusted third party would be required. In order to overcome these problems, an anonymity preserving approach that allows trusted platforms to protect their anonymity without involvement of a trusted third party is presented. The scheme can be used with existing Trusted Platform Modules version 1.2 and a detailed discussion of a proof-of-concept prototype implementation is provided.

For example, the approach proposed by Chen et al. [22] could use this approach as basis for computing the ring-signature. Moreover, whistle-blowers that disclose information to *Wikileaks* could use this scheme to authenticate the disclosed data without revealing their identity.

In order to address this problem, attention in this paper is turned to *ring-signatures* which have been introduced by Rivest et al in the year 2001 [87]. This kind of signatures allow a signer to create a signature with respect to a set of public-keys. This way, a verifier who can successfully verify the signature, can be convinced that a private-key corresponding to one of the public-keys in the set was used to create the signature. However, which private/public key-pair was used is not disclosed. Moreover, these signatures provide another interesting property: the ring-signatures are based on ad-hoc formed groups or lists of public-keys which can be chosen arbitrarily by the signer and they do not, in contrast to the PCA scheme or DAA scheme, rely on a third party. This last property is the most interesting one as we want to exploit this property for our purpose.

Nevertheless, such signatures can become large, depending on the number of contributing public-keys. Efficient ring-signature schemes have been proposed in [36] and [20]. Unfortunately, these schemes can not be applied for our purpose as we depend on the involvement of a TPM which we require to compute commitments and proofs for our approach.

A more advanced approach is discussed in Section 5.7. In this section, group signature scheme based on elliptic-curve cryptography (ECC) is discussed.

The Chapter is organized as follows, the Java improvements are analyzed in Section 5.2 followed by the introduction of the ring-signature based approach discussed in Section 5.3. Finally, in Section 5.7 a ECC based anonymous authentication scheme is introduced.

5.1.1 Related work and Contribution

Several publications address the topic of using ring-signatures for Trusted Computing systems: Chen et al proposed to use ring-signatures for hiding platform configurations [22]. Their approach aims at configuration anonymity which means that the signer proves that his platform's configuration is one out-of-n valid configurations. A verifier can check if the signer's configuration is a valid one, but the true configuration is not revealed. However, their paper does not focus on platform anonymity. In order to achieve platform anonymity, in addition to configuration anonymity, their approach still requires an extra anonymization scheme like PCA or DAA.

Tsang et al [114] discuss the application of ring-signatures in Trusted Computing. They investigate how this type of signature could be used to implement a DAA scheme based on linkable ring-signatures. However, they do not provide a detailed discussion of their idea. Moreover, they rely on group managers that set up the scheme and its parameters, thereby reversing the advantage of the ring-signatures which allows to neglect third-parties.

In contrast to these two publications, a scheme for platform anonymization in which trusted platforms do not require one of the above mentioned third-parties is proposed in this Chapter. Furthermore, a detailed discussion of how ring-signatures based on the

Schnorr signature algorithm [91] can be created using the TPM DAA commands is given. Therefore, we show how the Schnorr ring-signing scheme can be modified in order to meet the requirements of the TPM's DAA functionality. Moreover, a protocol is defined that allows a TPM to obtain a credential from a TPM vendor which is further used in the proposed approach.

–

5.2 Improvements for DAA signature creation on Java enables platforms

Today, nearly every mobile phone is equipped with a *Java virtual machine* which allows the device owner to install and run different applications *over-the-air* [80]. However, Java bytecode has the drawback that it runs slower than optimized C or Assembler code. As many important algorithms used for public-key cryptography such as RSA or DAA rely on computation-intensive arithmetic operations such as modular exponentiations, this drawback has enormous effects on the execution speed. The modular exponentiations required for these operations involve very long integers, typically ranging from 512 to 2048 bits. A modular exponentiation is generally realized through a sequence of modular multiplications and consumes the majority of the execution time in inner loops. Improving the performance of these loop operations, therefore, has a significant impact on the total execution time of public-key cryptosystems.

The most prominent representative of a public-key cryptosystem used in Trusted Computing is the DAA scheme introduced in section 4. This scheme involves many modular exponentiations, for example, creating a DAA signature requires the host to compute:

$$T_{host} = T_{tpm} * (A * S^w)^{r_1} * S^{r_{\bar{\nu}}} \mod N \tag{5.1}$$

which involves three modular exponentiations and three multiplication with bases typically ranging up to 2048 bits and exponents ranging from 344 to 2737 bits [[76]]. In addition, the TPM has to compute:

$$T_{tpm} = R_0^{r_{f_0}} * R_1^{r_{f_1}} * S^{r_\nu} \mod N \tag{5.2}$$

while on general Trusted Computing (TC) enabled platforms, these computations (5.2) are executed inside the TPM, in case of virtual TPMs or mobile trusted modules (MTMs) [[111]], [[35]] these exponentiations are done solely in software and are, therefore, performed on the platforms main CPU. According to (5.1) and (5.2), the host has to compute six modular exponentiations in total. Consequently, the most efficient implementation and parameter setting of the long integer modular arithmetic is crucial for the overall computation speed.

The performance of these computations can be improved in different ways. In this chapter, we discuss two of them: the first method addresses the used exponentiation algorithm. Many of these algorithms use windowing techniques that involve pre-computing values that are later required for the actual multiplications. Therefore, it is essential to find the optimal distribution between the number of pre-computated values and the number of modular multiplications in order to get the maximum overall performance. This distribution depends on the used window size - a larger window means more pre-computed values and lesser multiplications. However, the optimal window size depends on the operands and their lengths used for the modular exponentiation and, therefore, varies for different crypto-systems. To be more exact, the window size varies for different modular exponentiations, depending on their operands, which is also the case when creating DAA signatures.

A discussion of the results from the search of finding the optimal parameters for the DAA scheme is given in Section 5.2.5.

The second method addressed focuses on the Java architecture: Java offers the option of Just-in-Time (JIT) compilation [69] or *Dynamic-Ahead-Compilation (DAC)*. However, these options are not available on many embedded virtual machines [84]. Moreover, specialized operations like modular arithmetics have a significant performance advantage when implemented specifically for a certain platform instead of compiled by a JIT compiler. In order to take advantage of these performance gains with respect to the portability of Java applications, only the crucial parts of the arithmetic computations are moved to the native platform. These operations include the modular multiplication, modular reduction and squaring operations. The required modifications to the algorithm implementations and the different options for platform access from Java are discussed in Section 5.2.6.

5.2.1 Long integer arithmetic

Many important public-key cryptosystems, such as RSA or Diffie-Hellman, rely on modular exponentiation, for example, operations of the form $c = m^e \mod n$ where m, e and n are long integer values [5]. Although a variety of algorithms for modular exponentiation in literature exist, all of them can be reduced to modular multiplications, reductions and squarings. Hence, reducing the number of multiplications and reductions is desirable. The modular multiplication can be achieved in different ways. One method for achieving efficient reduction is discussed in the following Sections.

5.2.2 Sliding window exponentiation

Many exponentiation algorithms use a sliding window technique as discussed by Kaya Koc [18].

Algorithm 1 Sliding Window Exponentiation

```
Require: $g, e = (e_t e_{t-1} ... e_1 e_0)_2$ with $e_t = 1$, and an integer $k \geq 1$.
Ensure: $g^e$
 1: Precomputation
 2: $g_1 \Leftarrow g$, $g_2 \Leftarrow g^2$
 3: for $i = 1$ to $(2^{k-1} - 1)$ do
 4:     $g_{2i+1} \Leftarrow g_{2i-1} * g_2$
 5: end for
 6: end Precomputation
 7: $A \Leftarrow 1$, $i \Leftarrow t$
 8: while $i \geq 0$ do
 9:     if $e_i = 0$ then
10:         $A \Leftarrow A^2$,$i \Leftarrow i - 1$
11:     else
12:         find the longest bitstring $e_i e_{i-1} ... e_l$ such that $i - l + 1 \leq k$ and $e_l = 1$, and do the
            following:
13:         $A \Leftarrow A^{2^{i-l+1}} * g_{(e_i e_{i-1} ... e_l)_2}$, $i \Leftarrow l - 1$
14:     end if
15: end while
16: Return $(A)$
```

This technique reduces the average number of multiplications required for exponentiation. It is based on repeated squarings and multiplications but also includes a precomputation step. The number of these precomputations depends on the factor k. This factor denotes the size of our *window* - the larger the size of k the more precomputation have to be done and the possibility of fewer multiplications excluding precomputations is given. However, precomputing values is time consuming and might exceed the performance gain achieved by reducing the number of multiplications. Consequently, experimenting with k in order to find the optimal value for certain exponent lengths is recommended.

We have chosen an algorithm with a fixed window size as this kind of algorithms is one of the most widely used. Moreover, it is one of the fasted algorithm for modular exponentiation, it is easy to implement and provides a good trade-off between codesize and performance [19]. A comparison with algorithms that use variable window sizes is out of scope of this chapter.

5.2.3 Montgomery reduction

The Montgomery Reduction is a very efficient algorithm used for reduction in large modular multiplications. As shown in Algorithm 2, the reduction is done by calculating TR^{-1}. By choosing the so called *Montgomery residual factor* $R = b^n = 2^n$, this can be done very efficiently. Before doing modular multiplications, the factors have to be initialized by multiplying them with R, so that in the end, after multiplying the final result with R^{-1}, the *Montgomery residual factor* cancels out. The Montgomery Reduction also prohibits the intermediate values arising from multiplications and squarings to be large.

Algorithm 2 Montgomery reduction

```
Input: integers $m = (m_{n-1}...m_1m_0)_b$ with $gcd(m,b) = 1$, $R = b^n$,$m' = -m^{-1} \mod b$,
    and $T = (t_{2n-1}...t_1t_0)_b < mR$.
Output: $TR^{-1} \mod m$
 1: $A \Leftarrow T$ (Notation: $A = (a_{2n-1}...a_1a_0)_b$)
 2: for $i = 0$ to $(n-1)$ do
 3:     $u_i \Leftarrow a_im' \mod b$
 4:     $A \Leftarrow A + u_imb^i$
 5: end for
 6: $A \Leftarrow A/b^n$
 7: if $A \geq m$ then
 8:     $A \Leftarrow A - m$
 9: end if
10: $Return(A)$
```

5.2.4 Efficient squaring

For the squaring operations, we have chosen the *separated operand scanning method* as recommended in [[18]]. Although building the square of a long integer has an performance advantage over multiplying the value by itself, it can never be more than twice as fast as a multiplication [[5]].

5.2.5 Results

In this section, an analysis of the performance results of the exponentiation process is given. A first investigation of a single modular exponentiation revealed the average time for performing one single modular multiplication and one single squaring operation, as shown in Table 5.1. The results stem from the observation of the modular exponentiation

Algorithm 3 Multiple-precision squaring

```
Input: positive integer $x = (x_{t-1}x_{t-2}...x_1x_0)_b$
Output: $x * x = x2$ in radix $b$ representation
1: for $i = 0$ to $(2t - 1)$ do
2:     $w_i \Leftarrow 0$
3: end for
4: for $i = 0$ to $(t - 1)$ do
5:     $(uv)_b \Leftarrow w_{2i} + x_i * x_i$, $w_{2i} \Leftarrow v$, $c \Leftarrow u$
6:     for $j = (i + 1)$ to $(t - 1)$ do
7:         $(uv)_b \Leftarrow w_{i+j} + 2x_j * x_i + c$, $w_{i+j} \Leftarrow v$, $c \Leftarrow u$
8:     end for
9:     $w_{i+t} \Leftarrow u$
10: end for
11: $Return((w_{2t-1}w_{2t-2}...w_1w_0)_b)$
```

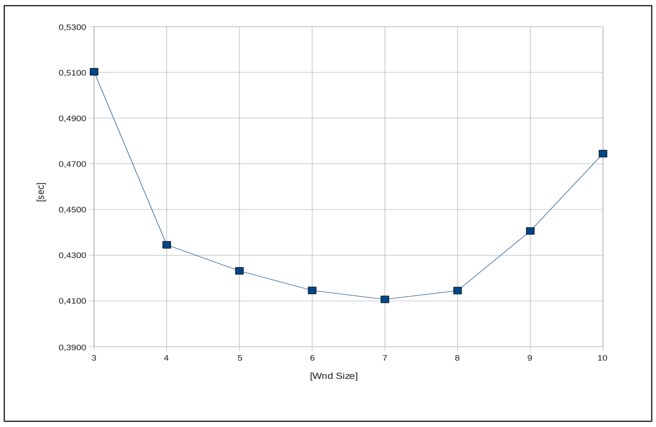
with varying exponents of the coefficient $S^{r_{\bar{\nu}}}$: This exponentiation which requires an exponent ($r_{\bar{\nu}}$) of size 2737 bits and a base (S) of a size up to 2048 bits is therefore one of the largest coefficients in Formula (5.1). The parameters where chosen according to the proposed values in the DAA scheme [76].

	multiply	**square**
operations	389	2737
meantime PC	0.17 ms	0.12 ms
meantime ARM9	19 ms	15 ms

Table 5.1: Performance of a single arithmetic operation

The values in Table 5.1 represent the average results when using a window size of 6. By varying the window size, the average performance values change according to Figure 5.1. The best performance is given at a windowsize of 7 that are $2^7 = 128$ precomputations which requires a total time of 625 ms to complete on the ARM9 platform and about 8.02 ms on the PC platform. (Note that all performance values were created with a modulus length of 2048 bits.)

Figure 5.1 Performance values for different windowsizes - single exponentiation



Combining the pre-computation, the modular multiplications, reductions and squarings together, we get the total time required for one single modular exponention as shown in Table 5.2.

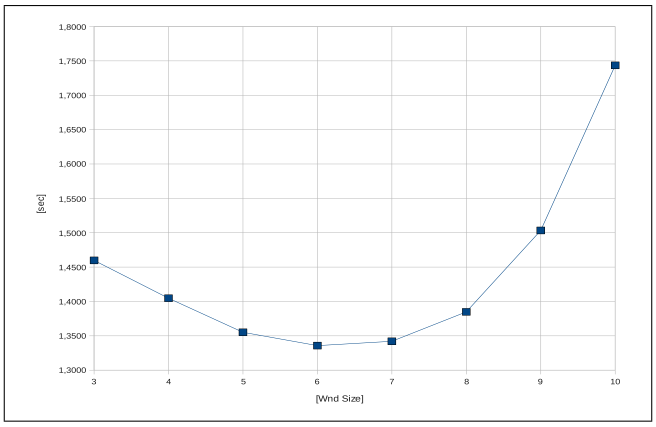
	PC	**ARM9**
modExp	0.41 s	50.14 s

Table 5.2: Performance of a single modular exponentiation

However, the measured values are only valid for this specific modular exponentiation with the given parameters. A sequence of modular exponentiations as used in DAA for creating a signature (5.1) (5.2) that involves operands with different base lengths and different exponent lengths show a different behavior and therefore require, a different average window size.

Figure 5.2 shows the computation speed in relation to the window size for a complete DAA signature. For creating a DAA signature, an optimum average windowsize of 6 is the best choice for the modular exponentiations. This setting results in 64 precomputated values per modular exponentiation.

Figure 5.2 Performance values for different windowsizes - DAA sign



Keep in mind that a complete DAA signature also includes the computation of $s_n = r_n + c * X$ for the private parameters X. These private parameters include the private keys f_0, f_1, ν and credential parameters $e, \bar{\nu}$ as well as the computation of the hash $c = H(H(H(n\|R_0\|R_1\|S\|Z\|\|T\|\tilde{T}\|n_\nu)\|n_t)\|m)$ of the system and signature parameters as well as the the message m. Nevertheless, our investigation revealed that although these computations have an influence on the overall computation time, they are neglible when selecting the optimal window size for the exponentiations.

Testequipment

For the experiments, an ARM9 based micro processor platform was used as it is installed in common mobile phones and a standard PC with a 2.00 GHz Semptorm CPU. The Java Virtual machines used for our investigations were a SUN Java 1.6 on the PC and a proprietary VM from SonyEricsson.

5.2.6 Java native calls

A major performance improvement can be achieved by executing the critical operations natively on the platform. Java applications are running in a sand-box and, therefore, can not execute native code a priori. However, different methods to overcome this constraint exist. In order to access platform specific functions and equipment, Java offers an interface, the Java native interface (JNI) [[103]] that allows to leave the virtual machine's sandbox, execute native code and return to original execution path of the Java application. Moreover, the interface provides a mechanism to transfer data to and from the Java application to the native code for processing.

The calls to the native world are associated with a small amount of time overhead for the native function call itself and the amount of data being transferred. Finding out how large the costs for native calls exactly are, is rather difficult, as it depends on the implementation of the JNI in the underlaying virtual machine. A native function call itself is estimated to be two to three times slower than a pure Java function call on a typical virtual machine [[67]]. Moreover, the transfer overhead depends on how the transfer is done, as various possibilities how to achieve that exist. A good advice is to avoid memcopies. Instead of doing a complete copy of the parameters from the Java heap to the C heap, the native function can also get direct access to the data inside the Java object heap, which results in an substantial performance-boost compared to a native call that uses memcopy for data transfer [[67]] (Chapter 9.2.6). Because of these costs, it is reasonable to reduce the number of native calls as this overhead could lead to a performance loss. However, in case of complex arithmetic operations, the time required for the computations outweighs the time required for data transfer.

Another mechanism to allow Java applications execute native code is to integrate the code directly into the virtual machine [[106]] where the VM offers an interface to the native code for applications executed by itself. This technique is used on many mobile JVMs as it is more lightweight than the JNI framework and easily portable to different platforms. However, the integration of the native code has to be provided by the VM manufacturer or the platform vendor.

As discussed in Section 5.2.1, modular exponentiations can be reduced to modular multiplications, reductions and squarings. These operations are the most time consuming ones when taking the whole exponentiation computation into account. Speeding up these computations results in a remarkable improvement of the whole operation. In our approach, we move these computations to the platform's execution environment. In detail, we now perform multiplications, squarings and reductions on the native platform. Although the overall number of these operations executed, of course, stays the same as when executed in pure Java, the performance gain is tremendous. Table 5.4 gives an overview of the achieved results.

mult.
Secure channel technologies - like Transport Layer Security (TLS) - are crucial components of many today's securit

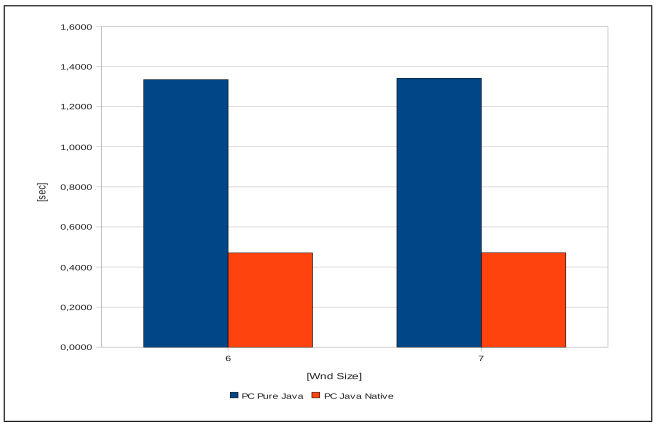
Table 5.3: Average performance values of a single modular exponentiation with windowsize 6

The speed of a single multiplication or a single squaring executed in native code is about 3 times faster compared to the same code executed in Java on the PC platform. On the ARM9 platform, the performance gain is about 5 times. Note that all multiplications and squarings already include the Montgomery reduction step. Keeping in mind that Java on the PC platform uses JIT compilation, the arithmetic operations done in native code even outrun the native code generated by the JIT compiler.

	Java ARM9	**native ARM9**	**JIT Java PC**	**native PC**
Sign	162.63 s	40.36 s	1.34 s	0.47 s
Verify	80.40 s	21.07 s	0.66 s	0.24 s

Table 5.4: DAA Signature Times

Figure 5.3 Pure Java and Java native code performance difference of DAA-sign (PC)

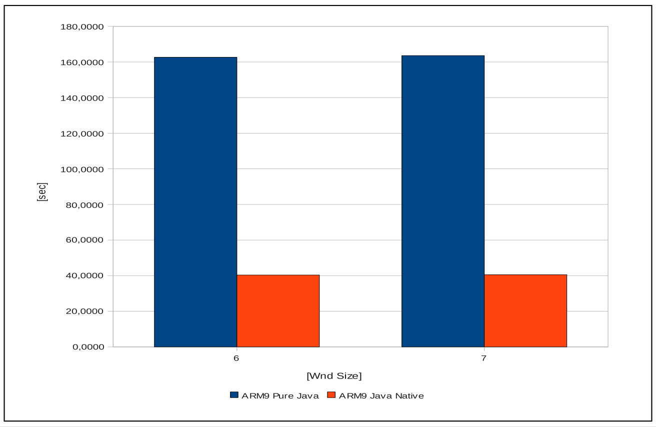


5.2.7 Deployment of the native library

The Java virtualization model allows mobile applications to be downloaded and executed more or less independent of the operating system and platform configuration. However, the involvement of specific platform code violates this assumption and a method for deploying our native code optimizations is required.

Using the JNI interface, the platform dependent code can be distributed by a simple mechanism. Java applications are typically deployed by packing the compiled Java classfiles together in a single archive file, which is then delivered to the target platform. This archive file can also contain the native code library. When starting the application, the library can be extracted and copied to the native filesystem on the target platform where the JVM is searching for dynamic libraries. After the library has been stored on the platform, the application gives a command to the JVM to load the library. For the k-native interface, the situation is different. As the native code is already on the platform - to be more exact, it resides inside the JVM, the application is not required to carry the native code by itself. The here is that this option is only available on specific JVMs. The native optimisations have to be integrated either by the platform integrator or the JVM vendor. In this case, it is reasonable that the application also carries the required arithmetic implementation in pure java and checks the presence of the native speedups before execution. If they are not present, the application has to use the arithmetic implemented in pure Java.

Figure 5.4 Pure Java and Java native code performance difference of DAA-sign (ARM9)



5.3 Issuerless anonymity protection with TPMs

As discussed in the previous Chapter, the two different anonymisation schemes for Trusted Computing platforms have been proposed by the Trusted Computing Group - the PrivacyCA scheme and the Direct Anonymous Attestation scheme. These schemes rely on trusted third parties that issue either temporary one-time certificates or group credentials to trusted platforms which enable these platforms to create anonymous signatures on behalf of a group. Moreover, the schemes require trust in these third parties and the platforms have to be part of their groups. However, there are certain use-cases where group affiliation is either not preferred or cannot be established. Hence, these existing schemes cannot be used in all situations where anonymity is needed and a new scheme without a trusted third party would be required. In order to overcome these problems, an anonymity preserving approach that allows trusted platforms to protect their anonymity without involvement of a trusted third party is presented. This new scheme can be used with existing Trusted Platform Modules version 1.2 which is demonstrated by a detailed discussion of a proof-of-concept prototype implementation.

5.3.1 Highlevel Description of the approach

In this section, a high level *discuss*ion of the proposed issuer-less approach is given. Furthermore, the following assumptions and definitions are introduced:

1. All TPMs are shipped with a unique RSA key-pair, the endorsement-key EK.
2. Moreover, it is assumed that the vendors of the TPMs have issued an endorsement certificate to the TPM's endorsement keys in order to prove the genuineness of the TPM.
3. Both the signing platform H and the verifying platform V have to trust the TPM and the TPM vendor.
4. An *endorsement-key* or EK denotes the endorsement key-pair (public and private part).
5. A *public-endorsement-key* or *public-*EK denotes the public part of an endorsement-key-pair.

6. An *endorsement-key-certificate* or *EK certificate* denotes a certificate that contains the public part of an endorsement key-pair.

7. A *schnorr-key* or *SK* denotes a schnorr key-pair (public and private part).

8. A *public-schnorr-key* or *public-SK* denotes the public part of a Schnorr-key-pair.

9. A *schnorr-key-certificate* or *SK-certificate* denotes a certificate that contains the public part of a Schnorr key-pair.

The proposed approach takes advantage of the fact that each TPM is part of a certain group right from the time of its production, namely the group that is formed from all TPMs of a certain manufacturer.

The approach is based on a *ring-signature* scheme where the ring is formed by a set of public-SKs and closed inside the TPM of the signer. Therefore, the prover has to show a verifier that the public-SK of the signing platform is an element of a group of public-SKs and that the ring was formed inside a genuine TPM. If he can successfully verify the signature, the verifier can trust that the signature was created inside a TPM. An introduction to ring-signatures can be found in [87] and [4] which is used as a basis for the trusted-third-party (TTP) less anonymization scheme.

For creating a signature, the signer chooses a set $S = (SK_0, .., SK_{n-1})$ of n public-SKs, that contribute to the signature. He computes the signature according to the algorithm discussed in Figure 5.5.

The ring is finally formed by computing the closing element inside the TPM. In typical Trusted Computing scenarios where remote attestation is used to provide a proof of the platform's configuration state, the signer generates an attestation-identity-key (AIK) with the TPM. This AIK is an ephemeral-key and can only be used inside the TPM for identity operations. In this scenario, the AIK is signed with the ring-signature which results in the signature σ on the AIK. Nevertheless, it is possible to sign any arbitrary message m with this approach.

The verifier can now validate the signature and knows that the real signer's public-SK is an element of the set S. As a consequence, the verifier knows that the signer was a trusted platform and that the ring was formed inside a TPM. However, the verifier can not reveal the real identity of the signer. How this is achieved in this approach is discussed in Section 5.3.3.

5.3.2 Discussion

A Signer H and verifier V have to trust the TPM and its vendor. The verifier V validates the public certificates of $SK_0, ..., SK_{n-1}$. If all certificates were issued by TPM vendors, the verifier knows that the signer platform is equipped with a genuine TPM from a certain vendor. Otherwise, he rejects the signature.
In contrast to the EKs which are pre-installed in the TPMs and certified by the vendors, SK are created dynamically in the TPM. Consequently, they have to be certified before they can be used for signature creation. How this is achieved and how SKs prove the genuineness of a TPM is discussed in Section 5.3.4.

The endorsement certificate and the SK-certificate cannot be linked to the TPM it belongs to, as it only provides information about the vendor of the TPM. This is true as long as the EK or the EK-certificate is not transmitted from a certain platform e.g. when used in a PrivacyCA or DAA scheme.

A typical Infineon EK-credential contains the following standard entries: the public-EK of the TPM, a serial number, the signature algorithm, the issuer (which is an Infineon intermediate CA), a validity period (typically 10 years), RSA-OAEP parameters and a basic constraint extension [55]. The subject field is left empty. For experimental purpose, SK-test-certificates with according entries were created.

The design of the TPM restricts the usage of the EK which can only be used for decryption and limits its usage to the two aforementioned scenarios. In these schemes, the EK-certificate could be used to track certain TPMs as the PCA might store certification requests and the corresponding TPMs. If the PCA is compromised, an adversary is able to identify which TPM created certain signatures. This is not possible in our scheme, as rings are formed ad-hoc and no requirement for sending the EK from the platform it belongs to, exist.

An SK-certificate might be revoked for some reason. In this case, the signer must realize this fact before creating a signature. Otherwise, the signer could create a signature, including invalid SKs. Assuming that the signer uses a valid SK to create his signature, the verifier would be able to distinguish between valid (the signers) SK and invalid SKs. Consequently, the signer's identity could be narrowed down or in case all other SKs are revoked, clearly revealed.

A time stamp could be used to define the time of signature creation. The validating platform could then check if the certificate was revoked before or after the time of signing. However, this idea requires the signer to use Universal Time Code (UTC) format in order to eliminate the time zone information which could also be used to narrow down the identity of the signer.

One advantage of this approach is that the SKs may be collected from different sources. However, in order to keep the effort for collecting the SKs and managing the repositories low, a centralized location for distributing the SK-certificates could be reasonable. Such a location might be the TPM vendor's website but it is not limited to this location.

The scheme can be applied in various use-cases where it is important to form ad-hoc groups with no dedicated issuer. Aside non-commercial and private usage scenarios, such groups, for example, often occur in peer-to-peer systems. Moreover, the scheme can be used according to Rivest's idea for whistle blowing [87].

5.3.3 Schnorr signature based approach

In this section, the Schnorr signature based approach which is based on a publication by Abe et al. is discussed, who proposed to construct ring-signatures based on Schnorr signatures [4] in order to reduce the size of the overall signature. In contrast to the approach from Rivest [87], the idea of Abe et al does not require a symmetric encryption algorithm for the signature creation and uses a hash function instead. This idea can be used for the approach with a few modifications of the sign and verify protocol. A major advantage of this approach is that it can be used with existing TPM 1.2 functionality to compute this kind of signatures. In order to do so, the DAA *Sign* and *Join* protocol implementation of the TPMs v1.2 can be exploited.

Signature Generation. Let n be the number of public-SKs contributing to the ring-signature and H a hash function $H : \{0,1\}^* \Rightarrow \mathbb{Z}_n$. j is the index of the signer's public-key SK_j consisting of y_j, the modulus N_j and g_j with $N_j = p_j q_j$ and p_j, q_j are prime numbers.

A signer S_j with $j \in (0, ..., n-1)$ has the private-key $f_j \in \{0,1\}^{l_H}$ and the public-key $y_j = g_j^{f_j} \mod N_j$.

The signer can now create a ring-signature on the message m by computing:

1. Compute $r \in \mathbb{Z}_{N_j}$ and $c_{j+1} = H_{j+1}(SK_0, ...SK_{n-1}, m, g_j^r \mod N_j)$
2. For i=j+1..$n-1$ and 0..$j-1$.
3. Compute $s_i \in \mathbb{Z}_{N_i}$ and $c_{i+1} = H_{i+1}(SK_0...SK_{n-1}, m, g_i^{s_i} y_i^{c_i} \mod N_i)$, if $i+1 = n$ then set $c_0 = c_n$.
4. Finally, calculate $s_j = r - f_j c_j \mod N_j$ to close the ring.

The result is a ring of Schnorr signatures $\sigma = (SK_0...SK_{n-1}, c_0, s_0, ..., s_{n-1})$ on the message m where each challenge is taken from the previous step.

Using a TPM 1.2 to Compute Schnorr Signatures

In the proposed scheme, the TPM of the signer is involved in signature generation in order to close the ring by exploiting the TPM's DAA commands. A detailed explanation of the DAA commands and their stages can be found in the following Paragraphs of this Section.

Although the DAA scheme is based on Schnorr signatures, the TPM is not able to compute Schnorr Signatures a-priori. However, the TPM_DAA_Sign and TPM_DAA_Join commands can be used to compute Schnorr signatures for our purpose. Therefore, the algorithm description with the stages that have to be gone through during the execution of the TPM commands has to be extended:

A signature on the message m can be computed as follows: Let (g, N) be public system parameters, $y = g^f \mod N$ the public-key and f the private-key (Note that for computational efficiency, f is split into f_0 and f_1 inside a TPM). For simplicity reasons, a common modulus N is used and a fixed base g for all contributing platform's in the further discussions. M is 20 byte long nonce required for computing a DAA signature inside a TPM.

Computing the Schnorr Ring-Signature. In order to compute a Schnorr signature, the TPM_DAA_Sign command can be exploited. Therefore, the protocol can be started in order to execute stages 0 to 11 as defined in [112], however, for the ring-signature computation, only stages 2 to 5 and 9 to 11 are of interest.

Table 5.5 shows the steps for running the DAA Sign protocol with a TPM. The TPM_DA
A_Sign command is executed in 16 stages by sub-sequent execution of the command.

It is not required to finish the *Sign* protocol and the DAA session can be terminated at stage 11 and leave out stages 12 to 15. Stages 6 to 8 have to be executed but the results can be ignored.

In order to use this approach, the Schnorr signature generation and verification scheme had to be modified: The TPM_DAA_Sign command requires a *nonce* from the verifier to get a proof for the freshness of the signature and computes $H(nonce||M_{T_j})$ where M_{T_j} is a random number generated inside the TPM. This proof is not required in the scheme and $c_{in} = H(g||N||y_0||..||y_{n-1}||e)$ (with $e = g^{r_0} y_j^{c_{j-1}} \mod N$) can be set. However, the resulting value M_{T_j} has to be recorded as it is required to verify the signature. As a result, the TPM computes $c_j = H(H(c_{in})||M_{T_j})||1or0||morAIK)$.

Figure 5.5 Schnorr Ring-Signature creation

1. Let $L = y_i$ with $(i = 0..n-1)$ be a list of n public-keys including the signer's key that contribute to the signature and let j be the index of the signer's public-key y_j.
2. Execute TPM_DAA_Sign to stage 5 and retrieve $T = g^{r_0} \mod N$ from the TPM (see Table 5.5 for the DAA Sign command steps).
3. Compute a random M_{T_i} and $c_{j+1} = H_{j+1}(H(H(g||n||y_0||..||y_{n-1}||T))||M_{T_i})||1or0||m$ or $AIK)$
4. For $i = j+1..n-1$ and $0..j-1$.

   Compute a random M_{T_i}, s_i.

   Compute $c_{i+1} = H_{i+1}(H(H(g||n||y_0||..||y_{n-1}||e_i))||M_{T_i})||1or0||m$ or $AIK)$ with $e_i = g^{s_i} y_i^{c_i} \mod N$.
5. To close the ring, continue to execute the TPM_DAA_Sign command protocol:

   Continue to stage 9 and send $c_{in} = H(g||n||y_0||..||y_{n-1}||e)$ with $e = T * y^{c_i}$ to the TPM which computes $c = H(c_{in}||M_{T_j})$ and outputs M_{T_j}.

   Continue to stage 10 and send either:

   $b = 1$, m is the modulus of a previously loaded AIK

   $b = 0, m = H(message)$ to compute $c = H(c||b||m/AIK)$ (where $c_j = c$)

   Continue at stage 11 and compute $s_j = r_0 + c_j f_0$ via the TPM
6. Abort the DAA protocol with the TPM and output the signature $\sigma = (c_0, s_0, .., s_{n-1}, M_{T_0}, ..., M_{T_{n-1}})$

The rest of the stages may be ignored and the session can be closed by issuing a TPM_Flush_Specific command to the TPM. The resulting signature is $\sigma = (c_0, s_0, ..s_{n-1}, M_{T_0}, ..M_{T_{n-1}})$ plus the list of public-SKs $\{SK_0...SK_{n-1}\}$. The parameter $b = 0$ instructs the TPM either to sign the message m that is sent to the TPM or if $b = 1$ to sign the modulus of an AIK which was previously loaded into the TPM. In this case, m contains the handle to this key which is returned when the key is loaded by a TPM_LoadKey2 command [112]. The latter case is the typical approach for creating AIKs that may be used for remote attestation.

Verifying the Schnorr Ring-Signature. The signature $\sigma = (c_0, s_0, .., s_{n-1}, M_{T_0}, ..., M_{T_{n-1}})$ can now be verified as follows in Figure 5.3.3: The verification of the signature does not

Figure 5.6 Schnorr Ring-Signature verification

1. For i=0..n-1
2. Compute $e_i = g^{s_i} y_i^{c_i} \mod N$ and $c_{i+1} = H(H(H(g||N||y_0||..||y_{n-1}||e_i))||M_{T_i})||1$ or $0||$ $morAIK)$.
3. Accept if $c_0 = H_0(H(H(g||N||y_0||..||y_{n-1}||e_{n-1})||M_{T_0})||1or0||m$ or $AIK)$

involve a TPM.

Parameter Setup. Before executing the Join protocol, the DAA parameters i.e. issuer public-key, issuer long-term public-key [112] have to be generated which are required

Stage	Input0	Input1	Operation	Output
0	DAA_issuerSettings	-	init	DAA_session handle
1	enc(DAA_param)	-	init	-
2	$R_0 = g$	n	$P_1 = R_0^{r_0} \mod N$	-
3	$R_1 = 1$	n	$P_2 = P_1 * R_1^{r_1} \mod N$	-
4	$S_0 = 1$	n	$P_3 = P_2 * S_0^{r_{\nu 1}} \mod N$	-
5	$S_1 = 1$	n	$T = P_3 * S_1^{r_{\nu 2}} \mod N$	T
.	.	.	.	.
.	.	.	.	.
9	c_{in}	-	$c' = H(c_{in}\|\|M_T)$	M_{T_j}
10	b	m or AIK handle	$c_j = H(c'\|\|b\|\|m)$	c_j
11	-	-	$s_0 = r_0 + c_j f_0$	s_0

Table 5.5: TPM_DAA_Sign Command Sequence

during the execution of the protocol to load the signature settings into the TPM. In order to compute the platform's public and private Schnorr key, first a commitment to a value f_0 by computing $y = g_0^f \mod N$ has to be established. This can be done executing the TPM_DAA_Join command: with the parameters: $R_0 = g, R_1 = 1, S_0 = 1, S_1 = 1$, a composite modulus $N = p * q$ where g is a group generator $g \in \mathbb{Z}_n$ and p, q prime values.

Stage	Input0	Input1	Operation	Output
0	DAA_count=0 (repeat stage 1)	-	init session	DAA_session handle
1	n	sig(issuer settings)	verify sig(issuer settings)	-
.	.	.	.	.
4	$R_0 = g$	n	$P_1 = R_0^{f_0} \mod N$	-
5	$R_1 = 1$	n	$P_2 = P_1 * R_1^{f_1} \mod N$	-
6	$S_0 = 1$	n	$P_3 = P_2 * S_0^{s_{\nu 0}} \mod N$	-
7	$S_1 = 1$	n	$y = P_3 * S_1^{s_{\nu 1}} \mod N$	y
.	.	.	.	.
24	-	-	E=enc(DAA_param)	E

Table 5.6: TPM_DAA_Join Command Sequence

After finishing the protocol, the public Schnorr key y and the secret-key f_0 which is stored inside the TPM have been obtained.

The DAA commands (as shown in Table 5.6) are executed in 25 stages by sub-sequently executing the command with different input parameters (*Input0*, *Input1*). Each stage may return a result (*Output*). Parameters that are marked with "-" are either empty input parameters or the operation does not return a result. Column *Stage* shows the stage, *Input0*, *Input1* the input data, column *Operation* the operation that is executed inside the TPM and *Output* shows the result of the operation.

In stage 7 the public-key $y = g^{f_0} \mod N$ can be obtained. Although they do not contribute to the public-key generation, the rest of the stages have to be run through in order to finish the *Join* protocol and to activate the keys inside the TPM.

The DAA_issuerSettings structure contains hashes of the system parameters (i.e. R_0, R_1, S_0, S_1, N) so that the TPM is able to prove whether the parameters that are used for the signing protocol are the same as the ones used during the *Join* protocol. A discussion how the issuer settings are generated is given in Section 5.5.

Security Parameter Sizes. The following list contains a suggestion of sizes for the required parameters:

1. $l_h = 160$ bits, length of the output of the hashfunction H.
2. $l_n = 2048$ bits, a public modulus.
3. $l_f = 160$ bits, size of the secret key in the TPM.
4. $l_r \in \{0,1\}^{l_f + l_h}$ bits, random integers.
5. $l_g < 2048$ bits, public base $g \in \mathbb{Z}_n$ with order n.

5.3.4 Obtaining a Vendor Credential

One issue remains open: while all TPMs are shipped with an endorsement-key and an according vendor certificate, our Schnorr key does not have such a credential. Hence, one can

1. assume that TPM vendors will provide Schnorr credentials and integrated them into TPMs right in the factory.
2. obtain a credential by exploiting the DAA Join protocol.

While the first solution is unlikely to happen, the second one can be achieved with TPMs 1.2. For this approach, one has to use the public RSA-EK and the DAA_Join protocol from the TPM.

The credential issuing protocol runs as follows:

1. The TPM vendor receives a request from the trusted client to issue a new vendor credential
2. The vendor computes a nonce and encrypts it with the client's public-EK
   $EN = enc(nonce_I)_{EK}$
3. The client runs the Join protocol to stage 7 and sends EN to the TPM (see Table 5.6)
4. The TPM decrypts the nonce and computes $E = H(y||nonce_I)$ and returns E
5. The client sends (E, y, N, g) to the vendor who checks if (E, y) is correct.
6. The vendor issues a credential on the public Schnorr key y.

By validating the EK-certificate, the vendor sees that the requesting platform is indeed one of its own genuine TPMs. Moreover, the encrypted nonce can only be decrypted inside the TPM which computes a hash from y and $nonce_I$, therefore, the issuer has proof that U was computed inside the TPM which he issues a certificate to.

5.3.5 Discussion

Experimental results show that the computation of a single sign operation involving a single public-key of the ring signature takes about 27 ms (on average) which is in total 27*(n-1) ms + sig_{TPM}. sig_{TPM} is the signing time of the TPM for the complete signature and n is the number of contributing keys. When computing a ring-signature with 100 public-keys, the overall time is about 3 seconds on average, making this approach feasible for desktop platforms. As a Java implementation was used for the tests, optimized C implementations (e.g. based on OpenSSL [109]) could increase this performance by a few factors. The verification of a ring signature takes about the same time as the signature creation. For details on the implementation see Section 5.5.

For the sake of completeness, the performance values of test TPMs are provided, demonstrating the time required for a full DAA-Join command and the stages 0-11 of the DAA-sign command (see Table 5.7).

Operation	**Infineon**	**ST Micro**	**Intel**
DAA Join:	49,7 s	41,9 s	7,6 s
DAA Sign Stages 0-11:	32,8 s	27,2 s	3,9 s

Table 5.7: `TPM_DAA_Sign` Command Measured Timings

For the DAA *Sign* operation, the stages 0-11 are of interest (see Figure 5.5), hence the computation can be aborted after stage 11. All measurement results are averaged values from 10 test runs. The Intel TPM is a more sophisticated micro controller than the ST Micro and Infineon TPMs and is integrated into the Intel motherboard chips which results in a tremendous performance advantage [61]. Details of the evaluation environment can be found in Section 5.5.

The slower performance of the Infineon TPM can be related to hard- and software side-channel countermeasures integrated in the microcontroller. These countermeasures are required to obtain a high-level Common Criteria certification such as the Infineon TPM has obtained [1].

The TPMs do not perform a detailed check of the *input0* and *input1* parameters, they only check the parameter's size which must be 256 bytes where the trailing bytes maybe be zero. Hence, it is possible to reduce the computation of the commitment from $U = R_0^{f_0} R_1^{f_1} S_0^{\nu_0} S_1^{\nu_1} \mod N$ to $U = R_0^{f_0} \mod N$ where f_0 is the private signing-key by setting $R_0 = g$ and $R_1 = S_0 = S_1 = 1$.

A similar approach is used for the signing process. In stages 2 and 11 from Table 5.5 the signature (c, s) on the message m is computed. The message may be a hash of an arbitrary message or the hash of the modulus n_{AIK} of an AIK that was loaded into the TPM previously.

If a signer includes a certificate other than a SK certificate in his ring, the verifier recognized this when verifying the credentials. If the signer closes the ring with a decrypt operation outside the TPM, the signature cannot be validated as he obviously did not use a valid SK and the assumption that only valid SKs may contribute to a signature is violated.

The originality of the TPM can be proven by the Schnorr SK-credential as the TPM vendor only issues certificates to keys that were created in genuine TPMs manufactured by

[1]http://www.trustedcomputinggroup.org/media_room/news/95

himself. This is proven during the execution of the DAA Join protocol where the vendor sends a nonce to the TPM which he encrypted with the original public-EK.

One could argue that obtaining a new vendor credential for the public part Schnorr key is just another form of joining a group like in the DAA scheme. But remember that all TPMs are part of the group formed by the TPMs of a certain vendor right from the time of manufacturing. Consequently, it is not required and not even possible to join the group again. Hence, our modified join protocol is a way of obtaining a credential for the Schnorr key.

5.4 RSA Signature Based Scheme

This approach is based on the assumption that every TPM is equipped with an endorsement key and that the vendor has issued a corresponding EK-credential to the EK. A concrete example, therefore, are the Infineon TPMs which are equipped with an EK-credential and which can be validated with information from the Infineon public-key infrastructure (PKI). Moreover, Infineon has obtained a certificate from Verisign to certify their TPM Vendor CA allowing Infineon EKs to be validated up to a commonly trusted root [2].

A drawback of this approach is, however, that the reverse operation of a signature with the EK inside the TPM in order to form the ring has to be computed. Existing TPMs do not support this feature the way the scheme requires it. The TPM can decrypt data that was previously encrypted with the public EK. This operation is based on the RSA-OAEP [73] encryption scheme which requires an OAEP padding of the encrypted data. However, a decryption operation is required that does not apply a padding in order to form the ring.

Creating the Signature. Our scheme uses the RSA-EK credentials of the TPMs to create a ring-signatures. A signer can create a ring-signature as follows:

Figure 5.7 RSA ring-signature creation

Let n be a public RSA modulus.

1. Fetch l endorsement key certificates.
2. Compute $r_k \in \{0,1\}^{l_n}$ and $c_{k+1} = H_{k+1}(EK_i, ..EK_{l-1}, m, r_k)$
3. Compute $s_i \in \{0,1\}^{l_n}$ and $c_{i+1} = H_{i+1}(EK_i, ..EK_{l-1}, m, (c_i + s_i)^{e_i} \mod n)$ for i= k+1,..n, l-1 and i = 0..k-1.
4. Send c_k to the TPM.
5. Compute $s_k = (r_k - c_k)^{d_{EK_R}} \mod n$ inside the TPM
6. Output the signature $\sigma = (EK_0, .., EK_{l-1}, c_0, s_0, .., s_{l-1})$

To compute a ring-signature, the TPM would require a functionality to compute a RSA decryption operation for computing $s_k = (r_k - c_k)^{d_k} \mod n$. Unfortunately, it is not possible to do so inside common TPMs. Although the EK can be used for decryption operations, it can not be used for arbitrary decryption operations because the EK is a RSA-OAEP key which applies a special padding on the data before encryption [73]. It is

[2]See http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab692060011a for further details

```
1. For i = 0,.., n-1
2. Compute $r_i = (c_i + s_i)^{e_i} \mod n$ (where $e_i$ is the public EK)
3. Compute $c_{i+1} = H_{i+1}(EK_0, .., EK_{n-1}, m, r_i)$
4. Check if $c_0 = H_0(EK_0..EK_{n-1}, m, r_{n-1})$ and accept if true
5. Verify that $EK_i$ for $i = 0..n-1$ are valid TPM endorsement credentials.
```

not possible to perform a *raw* decryption operation which would be required to close the ring.

In order to overcome this problem, an additional TPM command is introduced which was added, for testing purposes, to the TPM emulator [96]. The TPM_Ring_Close command allows to apply this decryption operation and to form the ring. The exact definition of the command can be found in the Appendix 6.1.

Verifying the Signature The signature $\sigma = (EK_0..EK_{n-1}, c_0, s_0, .., s_{n-1})$ on m can now be verified via:

A single signature operation with this approach takes 8 ms. Real-world performance values for the TPM operation cannot be provided as the TPM emulator was used. However, assuming that a typical RSA operation (e.g. the TPM_Sign command) takes about 1.5 seconds and which is the same operation that is required to close the ring, we get an overall result of 1500+8*n ms where n denotes the number of public-keys involved.

5.4.1 Discussion

This approach works with the endorsement credentials that are shipped together with the TPMs and which are issued by the TPM vendors.

The TPM is assumed to be trusted by host H and verifier V. For platform authentication, the anonymization scheme has to make sure that the verifier cannot be cheated by the signer and it must prevent the verifier from identifying the signer platform.

If a signer includes a certificate other than a EK certificate in his ring, the verifier recognized this when verifying the credentials. If the signer closes the ring with a decrypt operation outside the TPM, the signature cannot be validated as he obviously did not use a valid EK and the assumption that only valid EKs may contribute to a signature is violated.

It is not required that the list of EK credentials is transmitted along with the signature. The signer could, e.g., publish the list with a web-service and send only the URL pointing to the certificates

5.4.2 Comparison of both Approaches

In this Section, we give a short comparison of both approaches. While the RSA-based approach is faster and has smaller signatures than the Schnorr-based approach, the latter is the more interesting one. It does not involve the real EKs directly in the computations, but rather uses a revokable and renewable certificate issued by the vendor, thereby providing more flexibility as one single TPM may obtain more than one of these certificates.

Although they can be replaced by other certificates, they provide a proof of the originality of the TPM.

The poor performance of the DAA feature in many TPMs is vendor dependent (see Table 5.7) and might improve when the DAA feature receives greater attention.

The RSA-based scheme and the Schnorr-based scheme could be used in a "mixed" mode, allowing trusted platforms to use these different kinds of credentials to contribute to a single signature. However, we do not elaborate this idea in detail here and leave it open for further investigations.

5.5 Implementation Notes

In order to obtain experimental results, a Java library was implemented exposing the required set of TPM commands to use the TPM's DAA feature (`TPM_DAA_Sign`, `TPM_DAA_Join`, `TPM_FlushSpecific`, `TPM_OIAP`). On top of these primitives we provide Schnorr ring-signatures. The implementation was done in Java 1.6 as the runtime environment supports the required cryptographic operations like RSA-OAEP encryption that is used for the EK operations and modular exponentiations [5] which are required for computing the Schnorr signatures. The OAEP encryption is required for EK operations which encrypt the DAA parameters that are created and unloaded from the TPM during the *Join* protocol. The parameters are loaded into the TPM and again decrypted during the *Sign* protocol. Note that before executing the *Join* protocol, the public-EK of the TPM has to be extracted from the TPM for example by using the TPM tools from [56].
Our test platforms (Intel DQ965GF, Intel DQ45CB and HP dc7900) were equipped with 2.6 GHz Intel Core 2 Duo CPUs running a 64bit Linux v2.6.31 kernel and a SUN 1.6 Java virtual machine. Communication with the TPM is established directly via the file-system interface exposed by the Linux kernel's TPM driver. Our tests were performed with v1.2 TPMs from ST Micro (rev. 3.11), Intel (rev. 5.2) and Infineon (rev. 1.2.3.16).

5.5.1 Signature Sizes

In a straight forward implementation the size of Schnorr ring signatures can grow relatively large. For self-contained Schnorr ring-signatures which do not require any online interactions on behalf of the verifier, the overall signature size can be given as $l_h+(l_{SK}+l_{M_T}+l_s)*n$ with n being the number of public keys in the ring.
We assume that a verifier demands to see the entire SK-certificates instead of just the SK- public-key. Assuming approximately 1.3 kilobyte per SK-certificate [3] and the security parameters given at the end of Section 5.3.3 this yields an overall signature size of $20 + (1300 + 20 + 256) * n = 20 + 1576 * n$ bytes.
The relatively large signature size can be reduced if the burden of fetching the SK-certificates is shifted to the verifier. We have investigated two simple strategies which can reduce the effective signature size to reasonable values, assuming that the verifier has online access to a SK-certificate repository.
An obvious size optimization is to embed unique SK-certificate labels, like certificate hashes, instead of the SK-certificates themselves into the ring signature. When using 20-byte certificate hashes as SK-certificate labels, the overall signature size can be reduced to $20 + (20 + 20 + 256) * n = 20 + 296 * n$ bytes. The downside of this optimization is that

[3]This is the size of a typical ASN.1 [37] encoded EK-certificate from Infineon which we used as a template.

the verifier has to fetch all SK-certificates individually when verifying a signature. Further reduction of the signature size is possible by embedding a label representing the ring itself instead of its underlying SK-certificates in the signature. When using this strategy, the signature size can be reduced to $20 + (20 + 256) * n + l_{label} = 20 + 276 * n + l_{label}$ bytes where l_{label} denotes the size of the label.

5.6 RSA Signature Based Scheme

The approach discussed in this section is based on the assumption that every TPM is equipped with an endorsement key and that the vendor has issued a corresponding EK-credential to the EK. A concrete example, therefore, are the Infineon TPMs which are equipped with an EK-credential and which can be validated with information from the Infineon public-key infrastructure (PKI). Moreover, Infineon has obtained a certificate from Verisign to certify their TPM Vendor CA allowing EKs certified by Infineon to be validated up to a commonly trusted root CA [4].

A drawback of this approach is, however, that the reverse operation of a signature with the EK inside the TPM is required in order to form the ring. Existing TPMs do not support this feature the way it is required for this approach. The TPM can decrypt data that was previously encrypted with the public EK. This operation is based on the RSA-OAEP [73] encryption scheme which requires an OAEP padding of the encrypted data. However, in order to form the ring a decryption operation that does not apply or remove a padding is required.

Creating the Signature. The proposed scheme uses the RSA-EK credentials of the TPMs to create a ring-signatures. A signer can create a ring-signature as follows:

Figure 5.8 RSA ring-signature creation

Let n be a public RSA modulus.

1. Fetch l endorsement key certificates.
2. Compute $r_k \in \{0,1\}^{l_n}$ and $c_{k+1} = H_{k+1}(EK_i, ..EK_{l-1}, m, r_k)$
3. Compute $s_i \in \{0,1\}^{l_n}$ and $c_{i+1} = H_{i+1}(EK_i, ..EK_{l-1}, m, (c_i + s_i)^{e_i} \mod n)$ for i= k+1,..n, l-1 and i = 0..k-1.
4. Send c_k to the TPM.
5. Compute $s_k = (r_k - c_k)^{d_{EK_R}} \mod n$ inside the TPM
6. Output the signature $\sigma = (EK_0, .., EK_{l-1}, c_0, s_0, .., s_{l-1})$

To compute a ring-signature, the TPM would require a functionality to compute a RSA decryption operation for computing $s_k = (r_k - c_k)^{d_k} \mod n$. Unfortunately, it is not possible to do so inside common TPMs. Although the EK can be used for decryption operations, it can not be used for arbitrary decryption operations because the EK is a RSA-OAEP key which applies a special padding on the data before encryption [73]. It is not possible to perform a *raw* decryption operation which we would require to close the ring.

In order to overcome this problem, an additional TPM command is proposed which was added, for testing purposes, to the TPM emulator [96]. The TPM_Ring_Close command allows us to apply this decryption operation and to form the ring. The exact definition of the command can be found in the Appendix 6.1.

[4]See http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab692060011a for further details

```
1. For i = 0,.., n-1
2. Compute $r_i = (c_i + s_i)^{e_i} \mod n$ (where $e_i$ is the public EK)leave it
3. Compute $c_{i+1} = H_{i+1}(EK_0, .., EK_{n-1}, m, r_i)$
4. Check if $c_0 = H_0(EK_0..EK_{n-1}, m, r_{n-1})$ and accept if true
5. Verify that $EK_i$ for $i = 0..n-1$ are valid TPM endorsement credentials.
```

Verifying the Signature The signature $\sigma = (EK_0..EK_{n-1}, c_0, s_0, .., s_{n-1})$ on m can now be verified via:

A single signature operation with this approach takes 8 ms. Real-world performance values for the TPM operation cannot be provided as the TPM emulator was used. However, assuming that a typical RSA operation (e.g. the TPM_Sign command) takes about 1.5 seconds and which is the same operation that we require to close the ring, we get an overall result of 1500+8*n ms where n denotes the number of public-keys involved.

5.6.1 Discussion

This approach works with the endorsement credentials that are shipped together with the TPMs and which are issued by the TPM vendors.

The TPM is assumed to be trusted by host H and verifier V. For platform authentication, the anonymization scheme has to make sure that the verifier cannot be cheated by the signer and it must prevent the verifier from identifying the signer platform.

If a signer includes a certificate other than a EK certificate in his ring, the verifier recognized this when verifying the credentials. If the signer closes the ring with a decrypt operation outside the TPM, the signature cannot be validated as he obviously did not use a valid EK and the assumption that only valid EKs may contribute to a signature is violated.

It is not required that the list of EK credentials is transmitted along with the signature. The signer could, e.g., publish the list with a web-service and send only the URL pointing to the certificates

5.6.2 Comparison of both Approaches

In this Section, we give a short comparison of both approaches. While the RSA-based approach is faster and has smaller signatures than the Schnorr-based approach, the latter is the more interesting one. It does not involve the real EKs directly in the computations. It rather uses a revocable and renewable certificate issued by the vendor thereby providing more flexibility as one single TPM may obtain more than one of these certificates. Although they can be replaced by other certificates, they provide a proof of the originality of the TPM.

The slow performance of the DAA feature in many TPM's is vendor dependent (see Table 5.7) and might improve when the DAA feature receives greater attention.

The RSA-based scheme and the Schnorr-based scheme could be used in a "mixed" mode, allowing trusted platforms to use these different kinds of credentials to contribute to a single signature. However, this idea is not elaborated on in detail here and is left open for further investigations.

5.7 An anonymous authentication scheme

Anonymous authentication schemes have been proposed in various forms and based on different cryptographic primitives. Nevertheless, this section focuses on a special use-case where location based privacy is required. The use-case includes a set of requirements which stem from different sources like support of cryptographic functions on mobile devices or constrains of the verifying platform.

When investigation anonymous authentication scenarios for mobile devices like cell phones one can easily see the different requirements to scenarios involving, for example, smart-cards. While in the smart-card scenario the typical set-up includes a resource constraint client and a - at least - equally powerful verifier, in the mobile handset scenario the situation is different. On the smart-card the challenge lies on the signers side to compute the authentication information in *acceptable* time. Acceptable means that the human user has the subjective feeling that the authentication transaction is executed within acceptable time limits for him. Typically, this values lies between 300..800ms. The industry defines acceptable so that a single mobile transaction shall not exceed 300ms, for example, in the transportation sector. While a smart-card is resource constraint, the verifier is considered to have unlimited computational powers which is not true for every use-case.

Modern mobiles are equipped with powerful application processors which may have up to four CPU cores with a clock frequency of 1.6 GHz and giga bytes of memory, the coming generation will even employ 64 bit CPUs. The verifier side typically consists of a contactless card terminal with a 8 or 16 bit CPU and limited memory support. Consequently, the challenge lies not in gaining performance for creating the authentication information but in gaining performance for the verification step which also includes a revocation check.

The scheme is based on the publication of HeGe et al. [44].

5.7.1 Requirements

The basic use-case investigated in this section focuses on location based privacy. A user (i.e. the owner of a mobile device) wants to authenticate himself in order to pass a specific gate without the drawback of being traced.

The system requirements include:

1. a mobile hand-set that is capable of NFC-connectivity. The handset generates the authentication information and transmits it via the RFID interface to a RFID terminal.

2. a reader which acts as communication module for the gate lock. The reader may either have a constant connection to a back-end service available or it may have periodical connections to update locally stored information.

3. certificate validity check. In order to prevent revocation lists to grow indefinitely, the authentication credentials should be bound to a expiry date.

4. cryptographic support on the device. Therefore, focus is laid on ECC based algorithms for several reasons: First, industry requirements. Although, industry has put much effort in developing and publishing ECC based products support for ECC based algorithms is still not broadly available at the moment. Support for pairing-based

cryptography is virtually impossible to find in commercial products. Second, maturity of the used cryptographic primitives. While pairing based approaches provide shorter, more elegant and more compact algorithms [72], the E-Crypt II report states this technology as not widely deployed and "‘not mature"’ [58] or standadisation is in progress [59].

5. a fast revocation mechanism. The reader shall be able to locally verify authentication information. This information also includes revocation data.
6. the credentials must be non-transferable i.e. they are issued to specific device and may not be transferred to another one.
7. access to the credentials and their use has to be authenticated.
8. unforgeability. An adversary must not be able to forge the private-key f or the certificate U on f in order to create a valid signature.
9. selectable linkability. The user should be able to select whether his authentication operations are linkable or not.

5.7.2 The scheme

In this Section, the ECC based anonymous authentication scheme is introduced and discussed. The scheme is designed to meet the requirements from subsection 5.7.1.

5.7.3 Model

The model of our scheme includes three parties, the issuer (I) who constructs the domain parameters for the group he manages, the prover (P) who computes the anonymous signature and the verifier V.

5.8 Prerequisites & Requirements

- A collision resistant hash function $H : \{0,1\}^* \rightarrow \{0,1\}^{l_h}$
- A prime number generator according to P-1363 - A random number generator according to P-1363 - ECC domain parameters for specific curves, targeted curves are listed in NIST Suite-B - All computations must be done in sufficient time inside a *secure environment* - no computations must be done outside of this environment. A secure environment may be a dedicated *secure element* or TrustZone protected environment.

Parameter setup

In the setup protocol, the group-public-key and all required parameters for setting up the domain parameters for executing the join protocol are generated on by the issuer.

Issuer setup The Issuer Computes the following parameters:

1. a random private-key $\iota \in 0, 1^{l_\iota}$
2. $\Psi_I = \iota \cdot \Psi_g$ where $\Psi_g (\in < G >))$ is a group generator and point on the elliptic curve and ι is the issuer's private-key.

As a result, the following domain parameters are published by the issuer: $\Psi_g, n, r, \Psi_i, Psi_g$, the elliptic curve EC, and the generators $G_1..G_a$ plus the system parameters X, Y. (where r is the cardinality of the underlying prime field).

Client Setup The client computes a private key f and executes the following steps:

1. generate random $f \in 0, 1^{l_n}$
2. compute random $\tilde{w} \in 0, 1^{l_c}$
3. compute $\tilde{Y} = \hat{w} \cdot \Psi_I$
4. compute $\tilde{w} = f \cdot \hat{w} \mod n$

f is the clients private-key and is stored in a secure environment while $\tilde{Y}$ and $\tilde{w}$ are sent to the issuer.

Join protocol During the Join protocol, the client receives a credential U for his private key f. The protocol is based on the publication from Camenish et al. [17].

Before the Join protocol is executed, the client has to establish a secure and authenticated connection to the issuer. How this connection is established is not discussed here as many different mechanisms can be applied. A mechanism with a fixed authentication key (e.g. the EK) is used for authenticating TPM in a DAA scenario.

1. the client sends $\tilde{\omega}, \tilde{Y}$ to the issuer
2. the issuer computes: $\hat{r} \in 0, 1^l$ and $U = \tilde{\omega}^{-1}(\iota^{-1} + \hat{r})\tilde{Y} + \Psi_I$ and $P = \hat{r} \cdot \Psi_I$
3. and the proof: $c = H(\Psi_I \| \Psi_g \| r_I \cdot \Psi_g \| U)$ and $s = r_I + c \cdot \iota$ with $r_I \in 0, 1^{l_{r_I}}$
4. the issuer returns (c, s) and U, P to the client
5. the client verifies the proof by computing: $c' = H(\Psi_I \| \Psi_g \| s \cdot \Psi_g - c \cdot \Psi_I \| U)$ and accepts U if $c! = c'$ and $f \cdot U \equiv \Psi_g$

If the Join protocol was finished successfully, the client has the secret key f and the group credential U. Moreover, the issuer knows the authentication information (e.g. the EK) and the issued credential from the client (EK, U). The parameter $\hat{r}$ ensures the U differs in every join process even if the client re-uses $\tilde{Y}$ and $\tilde{w}$.

Sign protocol The signing protocol is executed in the secure environment of the client. There is no such splitting of computations as used in the DAA scheme

For generating a signature on the message m, the client executes the following steps:

1. compute random $r, r', r'' \in \{0, 1\}^l$.
2. compute $\Upsilon = r \cdot U, \Phi = r \cdot \Psi_g, \Gamma = r' \cdot \Upsilon, \Delta = r'' \cdot \Psi_g$,
3. compute $P' = r \cdot P$, $\overline{r} = r \cdot f$,
4. compute $c = H(\Psi_I \| \Psi_g \| \Upsilon \| \Phi \| \Gamma \| \Delta \| P' \| m)$ where m is the data to be signed
5. compute $\nu' = r' + c \cdot (f - X) \mod n$, $\nu'' = r'' + c \cdot (r - Y) \mod n$
6. assemble signature: $\sigma = (c, \nu', \nu'', \overline{r}, \Upsilon, \Phi, P')$

Verify Protocol

1. the verifier computes: $c' = H(\Psi_I \| \Psi_g \| (\nu' + cX) \cdot \Upsilon - c \cdot \Phi - c \cdot \overline{r} \cdot \Psi_I - c \cdot P' \| (\nu'' + cY) \cdot \Psi_g - c \cdot \Phi \| m)$ and checks if $c \overset{!}{=} c'$
2. check if $\nu' \in \pm 0, 1^{\alpha(l_f + l_c)+1}$ and $\nu'' \pm 0, 1^{\alpha(l_r + l_c)+1}$ range.

The protocol allows basic authentication of the device. In order to add information into the signature, i.e. which gates may be entered and the validity period of U, the scheme has to be modified.

Join protocol with attributes The issuer validates the attributes and computes $\overline{x} = H(x_1 \cdot G_1 \| ... \| x_a \cdot G_a)$are the indices of the attributes. The issuer computes $U = \tilde{\omega}^{-1}(\iota^{-1} + \overline{x} \cdot \hat{r}) \cdot \tilde{Y} + \Psi_I$ and $P = \hat{r} \cdot \Psi_I$

Sign protocol with attributes

1. compute $r, r', r'' \in \{0, 1\}^{l_r}$.
2. compute $\Upsilon = r \cdot U, \Phi = r \cdot \Psi_g, \Gamma = r' \cdot \Upsilon, \Delta = r'' \cdot \Psi_g$,
3. compute $\tilde{x}_{show} = H(x_a \cdot G_b \| x_b \cdot G_b \| ... \| x_g \cdot G_g)$ with all attributes that should be shown.
4. compute $P' = r \cdot \overline{x} \cdot \tilde{x}_{show}^{-1}) \cdot P$, $\overline{r} = r \cdot f \mod n$,
5. compute $\nu' = r' + c \cdot (f - X) \mod n$, $\nu'' = r'' + c \cdot (r - Y) \mod n$
6. compute: $c = H(\Psi_I \| \Psi_g \| \Upsilon \| \Phi \Gamma \| \Delta \| P' | \ m)$.
7. assemble signature: $\sigma = (c, \nu', \nu'', \Upsilon, \Phi, P', \overline{r}, SET(x_{i \in show}))$

The resulting signature now additionally contains the set of attributes that should be presented to the verifier.

The signature can then be verified via:

Verify Protocol

1. compute $\tilde{x}_{show} = H(x_a \cdot G_b \| x_b \cdot G_b \| ... \| x_g \cdot G_g)$ where $a, b, ..g$ are the indices of the shown attributes.
2. the verifier computes: $c' = H(\Psi_I \| \Psi_g \| \Upsilon \| \Phi \| (\nu' + cX) \cdot \Upsilon - c \cdot \Phi - c \cdot \overline{r} \cdot \Psi_I - c \cdot \tilde{x}_{show} \cdot P' \| (\nu'' + cY) \cdot \Psi_g - c \cdot \Phi \| P' \| m)$ and checks if $c \overset{!}{=} c'$
3. check if $\nu' \in \pm 0, 1^{\alpha(l_f + l_c)+1}$ and $\nu'' \pm 0, 1^{\alpha(l_r + l_c)+1}$ range.
4. V verifies all presented attributes $x \in show$.

5.8.1 Revocation

A second topic that receives attention in this Section is *revocation.* An efficient revocation mechanisms is required in order to manage large infrastructures with many different clients. Only if it is possible to remove clients and members of a certain group which have compromised devices or expired credentials this technology will obtain broad acceptance by industry and public.

However, efficient mechanisms are missing. Most publications and research in this are focus on fast execution of the signature creation on embedded devices or smart-cards and often neglect the revocation mechanism. Nevertheless, a positive authentication is only then finished when also the revocation check is finished, therefore the entire protocol has to be taken into account when providing performance estimations.

Traditional revocation is based on the idea that f becomes public and the

We assume that it is more likely that it becomes publicly known which platform and witjit which TPM has been compromised it is more reasonable to build revocation information based on the certificate issued to the TPM and its EK (i.e. on the pair (U, EK) as this relation is known to the issuer.

In our scenario, we employ *verifier-local-revocation* (VLR) i.e. all verifying entities hold a list with revoked credentials.

In the following paragraph, we discuss different revocation mechanisms that can be used with our scheme. Nevertheless, we put the revocation mechanism based on symmetric cryptography into favor.

T raditional revocation: The traditional approach to compute a pseudonym pn from a generator Ψ_g

Symmetric rogue tagging For computing the revocation information, the prover P may compute a symmetric signature with f on U.

1. A verifier (V) sends r_V which is a randomly generated number to P
2. P computes r_P and $PSN_P = HMAC(f, U \| r_P \| r_V)$ and sends both values to V
3. V computes $PSN_{V(i)} = HMAC(f_i, U_i \| r_P \| r_V)$ with $i = 0..n-1$ for all entries on the revocation list RL and checks if $PSN_P \in PSN_{V(i)}$.

r_P allows the prover to control the likability parameter by including a random value into every new revocation check. If $r_P = \{\}$ the check and with it the signature is linkable as the only other randomizing parameter is r_V.

An additional benefit of this approach is that this approach may be extended with respect to backward secrecy. Instead of publishing the actual signing keys and credentials f_i, U_i, only the hashes $H(f_i), H(U_i)$ of these parameters would have to be published. When producing a revocation list RL, the issuer computes $H(U)$ resp. $H(f)$, and puts these values on RL and signs it.

The modified protocol works as follows:

1. the verifier sends: r_V to P
2. P computes: $PSN = HMAC(H(f), H(U)||r_V||r_P)$
3. P sends PSN and r_P to verifier
4. V checks if $PSN \in HMAC(H(f_i), H(U_i)||r_V||r_P)$ with $i = 1..n.$ for all f_i, U_i on the revocation list RL.

The benefit of publishing $H(U)$ and $H(f)$ instead of U and f is the following: If an adversary manages to read a revocation list and gets a (revoked) pair $[f, U]$ that were newly added to a revocation list, before a certain verifier is able to get the updated list, he could manage to compute fake authorization information and exploit the delay in time that stems from the deployment of new lists. He could deceive the verifier as the verifier would still believe that the authorization information is valid (as he does not have the updated revocation list yet).

	Setup	Join	Sign
Montgomery auth. only	0,872s	0,854 s	4,279 s
Comba auth. only	0,194s	0,181s	0,894s
Montgomery w. attributes	0,869 s	0,856 s	5,267s
Comba w. attributes	144,8 ms	372,8 ms	517,6 ms

Table 5.8: Performance of the Join Protocol with Intel TPMs

Simple mechanism A more simpler approach may involve just the credential U. As the issuer knows which U belongs to which platform, he may employ revocation based on the credential U instead of f which is not known to him. This might be the case when it is publicly known that a certain platform acts rogue without that f is known. This approach is more like the approach used in common PKI systems.

So, alternatively, one could also check U only for revocation and exchange the $HMAC$ against a hash function H.

1. V sends r_V to P
2. P computes: $PSN = H(H(U)||r_V||r_P)$ and sends PSN, r_P to the verifier
3. V computes and checks if $H(H(U_i)||r_V||r_P)! = PSN$ with $i = 1..n$ for all U_i on the revocation table.

Connection to the signature Still, the revocation parameter PSN has to be connected to the signature so that a client may not send the verifier a signature and non-related PSN from valid verification check. Therefore, the signature creation process must be modified and the pseudonym PSN must be included in the computation of c:

The verification process is updated accordingly:

$c' = H(\Psi_I\|\Psi_g\|\Upsilon\|\Phi\|(\nu' + cX)\cdot\Upsilon - c\cdot\Phi - c\cdot\overline{r}\cdot\Psi_I - c\cdot\tilde{x}_{show}\cdot P'\|(\nu'' + cY)\cdot\Psi_g - c\cdot\Phi\|P'\|PSN\|m)$

With σ and PSN the verifier is now able to verify the authentication information, perform a revocation check and validate the provided attributes.

5.8.2 Results

The experiments were conducted on a MCB2130 MCU evaluation board [1] which was equipped with an ARM7TDMI microcontroller, The CPU clock was set to a frequency of 60 MHz. Furthermore, the board is equipped with an on-Chip RAM of 32kB and flash ROM with 512kB of memory. Connectivity is provided by Two serial ports and a JTAG Connector. Cryptographic operations are provided by the Miracl library [92] which provides MONTGOMERY modular arithmetic and COMBA multiplication. The system parameters include the elliptic curve P-192 standardized by NIST.

Note, the experimental implementation does neither take any implementation countermeasures like encrypted key storage or randomized byte array coping into account nor are any side.channel countermeasures applied. Such security features add an additional overhead to the overall performance of the algorithms. In addition, the used MCB 2130 board does not provide security mechanisms like crypto co-processors or encrypted memory.

The performance values were generated with a reference implementation done by Michael Kapfenberger. The show the execution speed of the authentication step plus two attributes.

5.8.3 Discussion

The presented scheme in this Section is designed to work with NIST or SECG defined elliptic curves. Although it is not as performant as the pairing based approaches it is faster than the discrete logarithm based counterparts. Nevertheless, one important thing demonstrated here is the computation of the pseudonym based on a hash function instead of a multiplication. Although the effect of performance gain on the client side is not that big as - in the best case - only a single multiplication for pseudonym computation can be spared the performance gain on the server side is tremendous. Especially for large revocation lists, the pseudonym computation via hashing is much more efficient than via multiplications. In addition, similar pseudonym computations based on hash functions has been used in the Austrian e-Government for years. The "bereichsspezifisches Personenkennzeichen" (bPK) or the "‘ wirtschaftsbereichsspezifisches Personenkennzeichen"’ (wbPK) are used for identification and authentication in the governmental and commercial sector.

Furthermore, the client has an additional mean to influence the pseudonym computation and eventually its traceability. By including the random value R_S in the revocation computation, the client can actively prevent the signer from re-using the same computation values for different authentication processes.

5.9 Conclusion

In this Chapter, two different approaches for optimizing and increasing the performance of the DAA scheme on Java enabled platforms are analyzed. The first approach addressed improvements of the modular exponentiation by finding the optimal window size for sliding window exponentiation algorithms with a fixed windows size.

The investigation has shown that the optimal window size, for the used sliding window technology, is six for the combination of all modular exponentiations in the DAA signature and seven for a single modular exponentiation. Although it would be possible to estimate the optimal window size for each single modular exponentiation of (5.1), the management-overhead would increase because one would have to keep track of which exponentiation operation is currently executed. Consequently, the suggestion is to use the obtained mean-value.

The second approach addressed special features provided by Java. Moving the critical operations of a modular exponentiation i.e. modular multiplications, squarings and reductions to native functions, results in a performance-gain, thereby providing a good balance between native code and pure Java code.

With a factor of three on the PC and a factor of four on an ARM9 processor, a major performance improvement could be achieved. However, the ARM9 is relatively slow compared to the PC, so, the factor of four brings us from 80 seconds for a verification down to approximately 20 seconds on this platform, which is already close for a possible practical use.

Unfortunately, this benefit is limited to platforms and JVMs with native support. The 40 seconds, which are required for a signature on an ARM9 with native functions, are still far away from practical use although they are already much better than the 160 seconds required for a signature computed in pure Java. The advantages of the native functions can not only be used for the DAA-signature, but all cryptography algorithms based on large number exponentiation should be able to reach a remarkable performance-gain when using native functions for their math operations.

Performance improvements could also be achieved by using dedicated security hardware for the computations which is typically much faster than software implementations. There are different approaches to investigate because the security hardware could be the SIM card that is attached to every mobile phone, it could be specific instruction set extensions [[49]] that are integrated in the main CPU or it could be Secure elements either fix attached to the device or removable as part of an SD card.

Another idea could be to change the cryptographic primitives of the DAA scheme towards pairing-based cryptography as suggested in [[42]].

Consequently, further research in the area of performance improvements for DAA should focus on the integration of dedicated cryptographic micro controllers and improved cryptographic algorithms.

In this section, an anonymization scheme was proposed for trusted platforms that does not rely on specialized trusted third parties. The approach is based on Schnorr ring-signatures which can be used with existing TPMs v1.2 without modifications of the TPM by well-thought exploitation of the TPM's DAA functionality.
The proposed theme is feasible for desktop platforms and that even large signatures can be created and verified in acceptable time. The performance is only limited by the performance of available TPM technology which differs strongly between the various TPM vendors.

Future investigations could include approaches using the ECC based variants of the Schnorr algorithm. This will be of interest as soon as TPMs support elliptic curve cryptography. Moreover, an investigation of the approach whether it is feasible for mobile platforms or not could be done in the future.

Chapter 6

Conclusions

In this thesis, different aspects of Trusted Computing were investigated. This endeavor is of special significance, as Trusted Computing is one of the most important security technologies employed nowadays. The new security paradigms introduced by Trusted Computing not only provide novel security mechanisms, but also allow major improvements of existing security concepts. In particular, Trusted Computing has caught attention in the embedded domain. Modern applications for digital computing are shifting more and more computational tasks to resource constrained devices like smart-cards, smart-phones, or RFID tags. With the potential advantages and risks of such a shift, the inclusion of sound and strong security measures becomes imperative. With the variety of new devices like smart meters, sensor nodes and smart cams the need for security support has risen. This is particularly true for devices that are managed remotely and which are exposed to a hostile environment. As a response to this strong demand from industry, the TCG has installed two major groups namely the mobile phone working group and the embedded systems working group which are exclusively focusing to deal with security matters in the embedded area. Nevertheless, input from science is required to further improve the security mechanisms and to explore new techniques.

The work presented here has focused on different security concepts used in Trusted Computing. The first concept analyzed was remote attestation, which as one of the new key concepts has earned much attention. The basic idea of reporting a platform's configuration allows a dynamic trust decision based on the actual state of a machine instead of a static third-party certification as done in conventional systems. The remote attestation concept has been investigated and possible improvements proposed. The binding of configuration change information to a TLS channel and the transported data allows active reporting of configuration changes. Furthermore, the gap between a configuration change and the notification of the remote platform can be closed.

The second part of the thesis addresses mobile Trusted Computing. While Trusted Computing is widely supported on desktop systems, mobile and embedded systems are still missing support. As different security extensions are available for embedded systems an analysis of these extensions has to be done. The different extensions also allow different design options for mobile TPMs which have been investigated according to their security requirements.

Moreover, the handling of mobile TPM implementations on resource constrained security devices has been investigated. Small embedded security devices suffer not only from limited processing powers, but also from memory constraints. To complicate these resources have to be shared among different stakeholders, consequently, functionality has to be sourced out. Embedded TPM functionality may be customized to a certain set of features defined by the use-cases they are currently applied in. In addition, updates can be applied much easier as not the entire firmware of the TPM has to be replaced. Embedded platforms offer an ideal basis for this approach as the out sourced functionality can be stored on the host platform like the the platform's non-volatile memory. A concept for achieving this goal in a secure way has been introduced and analyzed.

In the third part, anonymization techniques, in detail the Direct Anonymous Attestation protocol has been examined. Due to the versatility of modern smart phones, anonymity protection has become an asset that has to be protected. The used cell phones are typical representatives of available devices and are equipped with state-of-the art security mechanisms. These different security mechanisms allow different methods of implementing DAA. Furthermore, the impact and feasibility of anonymous authentication in combination with the raising RFID technology has been examined and problem areas identified.

Finally, the last chapter deals with improvements to anonymous authentication schemes. Optimizations to the DAA protocol have been proposed. As Java - with the help of Android - has become the dominant platform in the mobile phone sector, focus on this specific environment has been laid for improvements of the DAA signature creation process. In addition, different alternative models to the DAA scheme have been presented and investigated. One of those schemes is based on ring-signatures. The model shows how trusted platforms with the involvement of TPMs can be used to compute this kind of signature. Moreover, an ECC based anonymous authentication scheme has been introduced which is designed to meet the requirements of a specific mobile use-case. Furthermore, the proposed revocation mechanism is designed to cope with the limitations of resource constrained devices and to vastly improve the overall authentication procedure. In any case, Trusted Computing and its concepts have earned great attention by researchers as well as hackers. This fact makes it imperative to constantly investigate and improve the security protocols and security architectures employed. Moreover, existing security solutions should be reassessed according to the new possibilities introduced by inventions like remote attestation or DAA. Although Trusted Computing features are already widely deployed and supported by various platforms, it is foreseeable that this technology will get even larger attention. This is supported by the exploitation of new areas of application which go beyond the scope intended by the TCG. Especially, anonymization technologies are in the focus of research and industry as they are key enablers for new use- and business cases.

6.1 Further Work

Although thoroughly investigated, remote attestation still offers areas for research and improvement. For example, it still suffers from the complexity that stems from the rich amount of different configurations which makes the configuration check a costly task. As the load of validating is shifted to the verifier, he is responsible for checking the reported configurations. It is important to find new and more efficient methods for improving this tasks in order to increase the acceptance of Trusted Computing and to further spread this technology. In case of embedded systems, the situation is somewhat different. As the number of configurations a device - especially, mobile handsets - is able to manage, the actual work load of the verifier when validating the configuration is kept in limits. Consequently, at the current stage this technology is well suited for embedded systems -

better than for desktop systems.

Mobile TPMs are still not widely available. Although the lax specification offers many implementation options, these options also lead to different trust assumptions. Which trust assumption can be brought to a TrustZone based solution or to a secure element based solution? Although investigations have been done in this direction, a definite answer cannot be given at the moment. Also important to mention here is that TPMs are required to have a Common Criteria certification of at least EAL4. The best know TrustZone certification is aiming at EAL3+ and it is still work in progress. In contrast, secure elements are available with EAL5+ certifications - EAL6+ evaluated products are soon to come which definitely puts this solution into favor.

Nevertheless, there are many application scenarios where mobile TPMs of any kind can improve security. There is a is great request by industry, for example, to protect critical infrastructures, improve security of smart meters or to prevent manipulation of mileage meters in cars. However, desktop TPMs are equipped with a rich set of features and functions. This comes with the price of high complexity. It has to be reassessed if this complexity is really required for the embedded domain and its use-cases.

Finding efficient anonymizing technologies is one of the great challenges of the coming decade. With the growth of control and monitoring means, protection of privacy has become an important issue. Integration and application of these technologies into smartcards and passports is one of the future areas of research. Furthermore, the possible impact on in these areas for other upcoming technologies like RFID has to be explored. Moreover, as RFID devices are often battery powered or supplied with energy from an energy field, especially the power consumption when computing anonymous signatures is of interest.

However, efficient revocation mechanisms are still missing. Fast revocation mechanisms are essential, especially for use in applications with a large number of clients and members, e.g. in the public sector where many citizens are involved. For this reason, integrating the proposed revocation mechanism from this thesis in other anonymous authentication algorithms could be a future path for research. Another important step is standardization. For broader acceptance and deployment by industry, these technologies need to undergo a standardization processes.

Summing up all the open issues discussed in the previous paragraphs, it can be concluded+ that Trusted Computing still offers a wide range of future research areas.

Issuer key generation

1. The issuer chooses a modulus n with length l_n and primes p, q, p', q' such that:
$$n = pq, p = 2p' + 1, q = 2q' + 1$$
2. Next, the issuer chooses random integers $x_0, x_1, x_z \in [1, p'q']$ and $x \in [1, n]$ which will be used to generate the proof and computes
$$S = x^2 \bmod n \qquad Z = S^{x_z} \bmod n \qquad R_0 = S^{x_0} \bmod n \qquad R_1 = S^{x_1} \bmod n$$
3. The issuer produces a non interactive proof that S, Z, R_0 and R_1 are computed correctly.
4. It generates rouge tagging parameters by choosing random primes ρ, Γ and $\gamma' \in_R Z^*_\Gamma$, satisfying $\Gamma = r\rho + 1$ such that r is an integer, $\rho \nmid r$, $2^{l_\Gamma - 1} < \Gamma < 2^{l_\Gamma}$, $2^{l_\rho - 1} < \rho < 2^{l_\rho}$ and $\gamma'^{(\Gamma-1)/\rho} \not\equiv 1 mod(\Gamma)$. Furthermore, the issuer calculates
$$\gamma = \gamma'^{(\Gamma-1)/\rho} mod(\Gamma)$$
5. The private-key of the issuer is (p', q'), and the public-key is $(n, S, Z, R_0, R_1, \gamma, \Gamma, \rho)$.

Issuer-Key Proof Generation

1. The issuer generates random values
$$\tilde{x}_{(z,1)}, \ldots, \tilde{x}_{(z,l_H)}, \tilde{x}_{(0,1)}, ..., \tilde{x}_{(0,l_H)}, \tilde{x}_{(1,1)}, ..., \tilde{x}_{(1,l_H)} \in_R [1, p'q']$$
and for each $\tilde{x}_{(z,i)}$, $\tilde{x}_{(0,i)}$, $\tilde{x}_{(0,i)}$ computes
$$\tilde{Z}_{(z,i)} = S^{\tilde{x}_{(z,i)}} \qquad \tilde{R}_{0(0,i)} = S^{\tilde{x}_{(0,i)}} \qquad \tilde{R}_{1(1,i)} = S^{\tilde{x}_{(1,i)}}$$
2. The issuer then computes the hash value
$$c = H(n, S, Z, R_0, R_1, \tilde{Z}_{(z,1)}, ..., \tilde{Z}_{(z,l_H)}, \tilde{R}_{(0,1)}, ..., \tilde{R}_{(0,l_H)}, \tilde{R}_{(1,1)}, ..., \tilde{R}_{(1,l_H)})$$
3. Furthermore, for each bit c_i of c where $i \in [1, l_H]$ it computes
$$\begin{aligned} \tilde{x}_{(z,1)} &= \tilde{x}_{(z,1)} - c_i x_z \bmod p'q' \\ \tilde{x}_{(0,1)} &= \tilde{x}_{(0,1)} - c_i x_0 \bmod p'q' \\ \tilde{x}_{(1,1)} &= \tilde{x}_{(1,1)} - c_i x_1 \bmod p'q' \end{aligned}$$
4. Finally, the issuer publishes the proof
$$(c, X, \tilde{x}_{(z,1)}, ..., \tilde{x}_{(z,l_H)}, \tilde{x}_{(0,1)}, ..., \tilde{x}_{(0,l_H)}, \tilde{x}_{(1,1)}, ..., \tilde{x}_{(1,l_H)})$$

Issuer-Key Proof Verification In order to prove that the $Z, R_0, R_1 \in \langle S \rangle$ and that $S \in QR_n \bmod n$, the verifier (which in our case is the TTP) has to obtain the proof data structure, which contains:
$$(c, X, \tilde{x}_{(z,1)}, ..., \tilde{x}_{(z,l_H)}, \tilde{x}_{(0,1)}, ..., \tilde{x}_{(0,l_H)}, \tilde{x}_{(1,1)}, ..., \tilde{x}_{(1,l_H)})$$
To verify the proof we also require the public-key parameters (n, S, Z, R_0, R_1, γ, Γ, ρ). Proof verification proceeds as follows:

1. First, the verifier checks if S is QR_n mod n simply by asserting

$$S = X^2 \bmod n$$

2. Then for each bit c_i of c where $i \in [1, l_H]$ the verifier computes:

$$\tilde{Z}_{(z,i)} = Z^{c_i} S^{\tilde{x}_{(z,i)}} \qquad \tilde{R}_{0(0,i)} = R_0^{c_i} S^{\tilde{x}_{(0,i)}} \qquad \tilde{R}_{1(1,i)} = R_1^{c_i} S^{\tilde{x}_{(1,i)}}$$

3. Finally it computes the hash c' as:

$$c' = H\left(n, S, Z, R_0, R_1, \tilde{Z}_{(z,1)}, ..., \tilde{Z}_{(z,l_H)}, \tilde{R}_{(0,1)}, ..., \tilde{R}_{(0,l_H)}, \tilde{R}_{(1,1)}, ..., \tilde{R}_{(1,l_H)}\right)$$

4. Verification of the proof succeeds if $c' = c$ holds.

Appendix A

Definitions

A.1 Abbreviations

AES	Advanced Encryption Standard
AIK	Attestation Identity Key
CRHF	Collision Resistant Hash Function
DAA	Direct Anonymous Attestation
DRM	Digital Rights Management
ECC	Elliptic Curve Cryptography
GPS	Global Positioning System
HMAC	Hashed Message Authentication Code
HTTP	Hyper Text Transfer Protocol
HUK	Hardware Unique-Key
IPSec	Internet Protocol Security
JCA	Java Cryptographic Architecture
JIT	Just-in-Time
LPC	Low-Pin-Count
MAC	Message Authentication Code
ME	Mobile Equipment
MH	Mobile Handset
MLTM	Mobile Local-Owner Trusted Module
MRTM	Mobile Remote-Owner Trusted Module
MTM	Mobile Trusted Module
NFC	Near-Field-Communication
OCSP	Online Certificate Status Protocol
OTA	Over-the-Air
OWHF	One Way Hash Function
PCA	Privacy Certification Authority
PCR	Platform Configuration Register
PET	Privacy Enhancing Technology
PKI	Public-key Infrastructure
PNG	Portable Network Graphic
PRF	Pseudo Random Function
PS	Post Script
RA	Remote Attestation
RIM	Reference Integrity Measurement

RTE	Root-of-Trust-for-Enforcement
RTM	Root-of-Trust-for-Measurement
RTR	Root-of-Trust-for-Reporting
RTS	Root-of-Trust-for-Storage
RTV	Root-of-Trust-for-Verification
SATSA	Security and Trust Services API
SE	Secure Element
TC	Trusted Computing
TCG	Trusted Computing Group
TEE	Trusted Execution Environment
TIS	TPM interface specification
TLS	Transport Layer Security
TPM	Trusted Platform Module
TSS	Trusted Software Stack

A.2 Used Symbols

GND	Common Ground Identifier
V_{DD}	Supply Voltage Identifier

Bibliography

[1] *MCB2130 Evaluation Board - Technical Specifications*, 2012.

[2] Comprehensive TEX Archive Network (CTAN). `http://www.ctan.org/`.

[3] J. C. P. .-J. 218. *Connected Device Configuration (CDC) 1.1.* Specification available at: `http://jcp.org/en/jsr/detail?id=218`, 19 August 2005.

[4] M. Abe, M. Ohkubo, and K. Suzuki. 1-out-of-n signatures from a variety of keys. In *ASIACRYPT '02: Proceedings of the 8th International Conference on the Theory and Application of Cryptology and Information Security*, pages 415–432, London, UK, 2002. Springer-Verlag.

[5] S. Alfred J. Menezes, Paul C. Van Oorschot. *Handbook of applied cryptography.* CRC Press series on discrete mathematics and its applications. CRC Press, Boca Raton, c1997. Includes bibliographical references (p. 703-754) and index.

[6] T. Alves and D. Felton. *TrustZone: Integrated Hardware and Software Security - Enabling Trusted Computing in Embedded Systems.* Available online at: `http://www.arm.com/pdfs/TZ_Whitepaper.pdf`, July 2004.

[7] ARM. *TrustZone Ready Program.* `http://cc.arm.com/products/secure-services/trustzone-ready/index.php`, February 2012.

[8] ARM Ltd. *TrustZone Technology Overview.* Introduction available at: `http://www.arm.com/products/esd/trustzone_home.html`.

[9] ARM Ltd. *SecurCore SC200.* Overview available at: `http://www.arm.com/products/CPUs/SecurCore_SC200.html`, 19 August 2005.

[10] P. Bichsel, J. Camenisch, T. Groß, and V. Shoup. Anonymous credentials on a standard java card. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 600–610, New York, NY, USA, 2009. ACM.

[11] P. Bichsel, J. Camenisch, T. Groß, and V. Shoup. Anonymous credentials on a standard java card. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 600–610, New York, NY, USA, 2009. ACM.

[12] S. A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy.* MIT Press, Cambridge, MA, USA, 2000.

[13] E. Brickell, J. Camenisch, and L. Chen. *Direct Anonymous Attestation.* In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, New York, NY, USA, 2004. ACM.

[14] E. Brickell, L. Chen, and J. Li. *A New Direct Anonymous Attestation Scheme from Bilinear Maps.* In *TRUST*, pages p. 166–178, 2008.

[15] F. B. C. Latze, U. Ultes-Nitsche. Extensible Authentication Protocol Method for Trusted Computing Groups (TCG) Trusted Platform Modules. Internet-Draft, July 2009.

[16] J. Camenisch and A. Lysyanskaya. *A Signature Scheme with Efficient Protocols.* In *In SCN 2002, volume 2576 of LNCS*, pages p. 268–289. Springer, 2002.

[17] J. Camenisch and M. Michels. Proving in zero-knowledge that a number is the product of two safe primes, 1998.

[18] Çetin Kaya Koc. *Analysis of Sliding Window Techniques for Exponentiation. Computers and Mathematics with Applications*, vol. 30:p. 17–24, 1995.

[19] Çetin Kaya Koc and T. Acar. *Analyzing and Comparing Montgomery Multiplication Algorithms. IEEE Micro*, vol. 16:26–33, 1996.

[20] N. Ch, J. Groth, and A. Sahai. Ring signatures of sub-linear size without random oracles. In *In ICALP07, LNCS.* Springer, 2007.

[21] L. Chen. A daa scheme requiring less tpm resources. Cryptology ePrint Archive, Report 2010/008, 2010. `http://eprint.iacr.org/`.

[22] L. Chen, H. Löhr, M. Manulis, and A.-R. Sadeghi. Property-based attestation without a trusted third party. In *ISC '08: Proceedings of the 11th international conference on Information Security*, pages 31–46, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] G. Consortium. *GlobalPlatform Card Specification.* GlobalPlatform Card Specification v2.2.1, January 2011.

[24] V. Costan, L. F. Sarmenta, M. Dijk, and S. Devadas. The trusted execution module: Commodity general-purpose trusted computing. In *CARDIS '08: Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, pages 133–148, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] H. D. Eastlake. *Transport Layer Security (TLS) Extensions: Extension Definitions.* RFC 6066, jan 2011.

[26] J. Daemen and V. Rijmen. *The Block Cipher Rijndael.* In *Proceedings of the The International Conference on Smart Card Research and Applications*, pages 277–284, London, UK, 2000. Springer-Verlag.

[27] T. Dierks and C. Allen. *The TLS Protocol Version 1.0.* RFC 2246 (Proposed Standard), jan 1999. Obsoleted by RFC 4346, updated by RFC 3546.

[28] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006.

[29] K. Dietrich. *Secure Signature Creation using the Java 2 Micro Edition on Mobile Devices.* Master's thesis, Institute for Applied Information Processing and Communications, Technical University Graz, Austria, Inffeldgasse 16a, 8010 Graz, Austria, October 2003.

[30] K. Dietrich. *An integrated architecture for trusted computing for java enabled embedded devices.* In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 2–6, New York, NY, USA, 2007. ACM.

[31] K. Dietrich. Anonymous credentials for java enabled platforms. In L. Chen and M. Yung, editors, *INTRUST 2009*, pages p. 101 –p. 116, 2009.

[32] K. Dietrich. *Anonymous Credentials for Java Enabled Platforms.* In M. Y. L. Chen, editor, *Proceedings of the International Conference on Trusted Systems (INTRUST 2009)*, Heidelberg, 2010. Springer LNCS.

[33] K. Dietrich, M. Pirker, T. Vejda, R. Toegl, T. Winkler, and P. Lipp. A practical approach for establishing trust relationships between remote platforms using trusted computing. In G. Barthe and C. Fournet, editors, *Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 156–168. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78663-4-12.

[34] K. Dietrich and J. Winter. Secure boot revisited. In *ICYCS '08: Proceedings of the 2008 The 9th International Conference for Young Computer Scientists*, pages 2360–2365, Washington, DC, USA, 2008. IEEE Computer Society.

[35] K. Dietrich and J. Winter. Implementation aspects of mobile and embedded trusted computing. In L. Chen, C. J. Mitchell, and A. Martin, editors, *TRUST*, volume 5471 of *Lecture Notes in Computer Science*, pages p. 29–44. Springer, 2009.

[36] Y. Dodis, A. Kiayias, A. Nicolosi, and V. Shoup. Anonymous identification in ad hoc groups. In *IN EUROCRYPT 2004, VOLUME 3027 OF LNCS*, pages 609–626. Springer-Verlag, 2004.

[37] O. Dubuisson and P. Fouquart. *ASN.1: communication between heterogeneous systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[38] D. V. E. Cesena, G. Ramunno. D03c.3 ssl/tls daa-enhancement specification. Technical report, Politecnico Di Torino, May 2009.

[39] J.-E. Ekberg, N. Asokan, K. Kostiainen, and A. Rantala. Scheduling execution of credentials in constrained secure environments. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 61–70, New York, NY, USA, 2008. ACM.

[40] P. England and T. Tariq. Towards a programmable tpm. In *TRUST*, pages 1–13, 2009.

[41] F. S. Foundation. *The GNU Transport Layer Security Library.*

[42] S. D. Galbraith and K. G. Paterson, editors. *Pairing-Based Cryptography - Pairing 2008, Second International Conference, Egham, UK, September 1-3, 2008. Proceedings*, volume 5209 of *Lecture Notes in Computer Science*. Springer, 2008.

[43] Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, and N. Asokan. Beyond secure channels. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 30–40, New York, NY, USA, 2007. ACM.

[44] H. Ge and S. R. Tate. A direct anonymous attestation scheme for embedded devices. In *Proceedings of the 10th international conference on Practice and theory in public-key cryptography*, PKC'07, pages 16–30, Berlin, Heidelberg, 2007. Springer-Verlag.

[45] GlobalPlatform. *Trusted Execution Environment (TEE) Specifications.*

[46] GlobalPlatform. GlobalPlatform Card Specification v2.2, March 2006.

[47] K. Goldman, R. Perez, and R. Sailer. Linking remote attestation to secure tunnel endpoints. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 21–24, New York, NY, USA, 2006. ACM.

[48] Google. *Googlet Wallet.*

[49] J. Grosschadl, S. Tillich, and A. Szekely. *Performance Evaluation of Instruction Set Extensions for Long Integer Modular Arithmetic on a SPARC V8 Processor. Euromicro Symposium on Digital Systems Design*, pages p. 680–689, 2007.

[50] M. P. W. Group. *Selected Use Case Analyses v1.0*, Sept. 2009.

[51] M. P. W. Group. *Mobile Trusted Module 2.0 Use Cases v1.0*, March 2011.

[52] T. C. G. .-M. W. Group. *TCG Mobile Trusted Module Sepecification Version 1 rev. 7.02.* Specification available online at: `https://www.trustedcomputinggroup.org/specs/mobilephone/tcg-mobile-trusted-module-1.0.pdf`, 29 April 2010.

[53] T. C. G. .-T. W. Group. *TPM Main Part 2 Structures.* Specification available at: `http://www.trustedcomputinggroup.org/files/resource_files/8D3D6571-1D09-3519-AD22EA2911D4E9D0/mainP2Structrev103.pdf`, 9 July 2007. Specification version 1.2 Level 2 Revision 103.

[54] T. C. G. .-T. W. Group. *TPM Main Part 1 Design Principles.* Specification available online at: `http://www.trustedcomputinggroup.org/files/resource_files/ACD19914-1D09-3519-ADA64741A1A15795/mainP1DPrev103.zip`, 1 March 2011 2011. Specification version 1.2 Level 2 Revision 116.

[55] R. Housley, R. Laboratories, W. Polk, NIST, W. Ford, VeriSign, D. Solo, and Citigroup. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile - rfc 3280, April 2002.

[56] IBM. *TrouSerS The opensource TCG Software Stack*, 2 November 2007.

[57] IEEE. Ieee standard specifications for public-key cryptography, 2000.

[58] E. II. *ECRYPT II Yearly Report on Algorithms and Keysizes (2009-2010)*, March 2010.

[59] E. II. *ECRYPT II Yearly Report on Algorithms and Keysizes (2010-2011)*, June 2011.

[60] Intel. *Intel Desktop Board DQ965GF Technical Product Specification.* Specification available at: `downloadmirror.intel.com/15033/eng/DQ965GF_TechProdSpec.pdf`, September 2006.

[61] Intel. *Intel Desktop Board DQ45CB Technical Product Specification.* Specification available at: `http://downloadmirror.intel.com/16958/eng/DQ45CB_TechProdSpec.pdf`, September 2008.

[62] International Organisation for Standardisation. *ISO/IEC 7816-4*, 2005. Part 4: Interindustry commands for interchange.

[63] International Organisation for Standardisation. *ISO/IEC Standard 18033-2, Information technology – Security techniques – Encryption algorithms*, 2006. Part 2: Asymmetric ciphers.

[64] J. M. Balasch Masoliver. *Smart Card Implementation of Anonymous Credentials.* Master's thesis, K.U.Leuven, Belgium, 2008.

[65] M. K. Jan-Erik Ekberg. *Mobile Trusted Module (MTM) - an introduction.* Available online at:
`http://research.nokia.com/files/NRCTR2007015.pdf`, November 14 2007.

[66] *Trusted Computing for Java.* Available online at: `http://trustedjava.sourceforge.net/`.

[67] S. J. Kesselman. *Java Platform Performance: Strategies and Tactics.* Addison Wesley, 2000.

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

[69] A. Krall. *Efficient JavaVM Just-in-Time Compilation.* In *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, 1998.

[70] K. Kursawe and D. Schellekens. *Flexible μTPMs through disembedding.* In *ASIACCS '09*, pages 116–124, New York, NY, USA, 2009. ACM.

[71] K. Kursawe, D. Schellekens, and B. Preneel. Analyzing trusted platform communication. In *In: ECRYPT Workshop, CRASH - CRyptographic Advances in Secure Hardware*, page 8, 2005.

[72] D. P. L. Chen and N. Smart. On the design and implementation of an efficient daa scheme. Cryptology ePrint Archive, Report 2009/598, 2009. `http://eprint.iacr.org/`.

[73] R. Labs. *PKCS1 v2.1: RSA Cryptography Standard*, 2001.

[74] Y. F. Lindholm Tim. *The Java Virtual Machine Specification Second Edition.* Available online at: `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`.

[75] S. Microsystems. *Java Card 3.0.1 Platform Specification.* Overview available at: `http://java.sun.com/javacard/3.0.1/specs.jsp`.

[76] C. Mitchell. *Direct Anonymous Attestation in Context.* In *Trusted Computing (Professional Applications of Computing)*, pages p. 143–p. 174, Piscataway, NJ, USA, 2005. IEEE Press.

[77] M. S. C. T. Mourad Debbabi and S. Zhioua. *Security Evaluation of J2ME CLDC Embedded Java Platform. Journal of Object Technlogy*, 5(2):125-154, March-April 2006.

[78] National Institute of Standards and Technology. *Secure Hash Standard.* available online at: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf, 1 August 2002.

[79] National Institute of Standards and Technology - NIST. *Digital Signature Standard (DSS) FIPS-186-3.* Technical report, National Institute of Standards and Technology (NIST), June 2009.

[80] E. C. Ortiz. *Introduction to OTA Application Provisioning.* Technical report, SUN Developer Network, November 2002. Article available at: `http://developers.sun.com/mobility/midp/articles/ota/`.

[81] S. Pearson. *Trusted Computing Platforms, the Next Security Solution.* Technical report, Trusted E-Services Laboratory, HP Laboratories Bristol HPL-2002-221, 5 November 2002.

[82] C. Porthouse. *High performance Java on embedded devices.* Technical report, ARM Ltd., October 2005.

[83] S. C. process. *Java Specification Request (JSR-257): Contactless Communication API.* Specification available at: `http://jcp.org/en/jsr/detail?id=257`, October 2004.

[84] S. C. process JSR 139. *J2ME(TM) Connected Limited Device Configuration (CLDC) Specification 1.1 Final Release.* Specification available at: `http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html`, 4 March 2004.

[85] S. C. process JSR 271. *J2ME(TM) Mobile Information Device Profile (MIDP) 3.0.* Draft available at: `http://www.jcp.org/en/jsr/detail?id=271`. Work in progress.

[86] E. Rescorla. *SSL and TLS: designing and building secure systems.* Addison-Wesley, Boston, c2001. Includes bibliographical references (p. 465-474) and index.

[87] R. L. Rivest, A. Shamir, and Y. Tauman. How to leak a secret. In *ASIACRYPT '01: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security*, pages 552–565, London, UK, 2001. Springer-Verlag.

[88] J. Salowey, A. Choudhury, and D. McGrew. *AES Galois Counter Mode (GCM) Cipher Suites for TLS.* RFC 5288 (Proposed Standard), August 2008.

[89] J. Salowey, H. Zhou, C. Systems, P. Eronen, Nokia, H. Tschofenig, and N. S. Networks. *Transport Layer Security (TLS) Session Resumption without Server-Side State.* RFC 5077 (Proposed Standard), jan 2008. Obsoletes RFC 4507.

[90] M. T. Sarmenta L., Rhodes J. TPM/J Java-based API for the Trusted Platform Module (TPM). Available online at: `http://projects.csail.mit.edu/tc/tpmj/`, Arpil 2007.

[91] C. P. Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 239–252, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[92] M. Scott. Miracl library. http://www.shamus.ie/, 2011.

[93] M. Sterckx, B. Gierlichs, B. Preneel, and I. Verbauwhede. *Efficient Implementation of Anonymous Credentials on Java Card Smart Cards.* In *1st IEEE International Workshop on Information Forensics and Security (WIFS 2009)*, pages 106–110, London,UK, 2009. IEEE.

[94] Stiftung SIC. *The IAIK JCE iSaSiLk v4.4 TLS Library.* Specification available at: http://jce.iaik.tugraz.at/index.php/sic/Products/Communication-Messaging-Security/iSaSiLk.

[95] Stiftung SIC. *The IAIK JCE MicroEdition Crypto Library - J2ME SDK v3.04.* http://jce.iaik.tugraz.at/sic/products/core_crypto_toolkits/jce_me/version.

[96] M. Strasser. *TPM Emulator.* Software package available at: `http://tpm-emulator.berlios.de/`.

[97] F. Stumpf, O. Tafreschi, P. Röder, and C. Eckert. A robust integrity reporting protocol for remote attestation. In *Second Workshop on Advances in Trusted Computing (WATC '06 Fall)*, Tokyo, Japan, November 2006.

[98] SUN. Javacard protection profile, May 2006.

[99] SUN Community process - JSR 139. *J2ME(TM) Connected Limited Device Configuration (CLDC) Specification 1.1 Final Release.* Specification available at: `http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html`, 4 March 2004.

[100] SUN Developer Network. *Java ME at a Glance.* Specifications and Articles availablbe at: `http://java.sun.com/javame/index.jsp`.

[101] Sun Microsystems. *Java Card Technology.* Overview available at: `http://java.sun.com/products/javacard/`.

[102] SUN Microsystems. *Java Cryptography Architecture (JCA) Reference Guide for JavaTM Platform Standard Edition 6.* Specification available at: `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[103] SUN Microsystems. *Java Native Interface Specification.* Available online at: `http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html`.

[104] SUN Microsystems. *JSR 177: Securtiy and Trust Services API.* Specification available at: `http://java.sun.com/products/satsa/`.

[105] SUN Microsystems. *Java Card Platform Specification 2.2.2.* Specification available at: `http://java.sun.com/products/javacard/specs.html`, March 2006.

[106] I. Sun Microsystems. K native interface (kni). Technical report, 4150 Network Circle Santa Clara, California 95054, December 2002.

[107] R. I. T. Dierks, E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[108] The Legion of BouncyCastle. *Bouncy Castle Crypto APIs for Java - Lightweightcrypto j2me 1.43.* http://www.bouncycastle.org/java.html.

[109] The OpenSSL Project. OpenSSL. Programa de computador, December 1998.

[110] Trusted Computing Group - Mobile Phone Working Group. *TCG Mobile Reference Architecture.* Specification available online at: `https://www.trustedcomputinggroup.org/specs/mobilephone/tcg-mobile-reference-architecture-1.0.pdf`, 12 June 2007. Specification version 1.0 Revision 1.

[111] Trusted Computing Group - Mobile Phone Working Group. *TCG Mobile Trusted Module Sepecification Version 1 rev. 1.0.* Specification available online at: `https://www.trustedcomputinggroup.org/specs/mobilephone/tcg-mobile-trusted-module-1.0.pdf`, 12 June 2007.

[112] Trusted Computing Group - TPM Working Group. *TPM Main Part 3 Commands.* Specification available online at: `http://www.trustedcomputinggroup.org/files/static_page_files/72C33D71-1A4B-B294-D02C7DF86630BE7C/TPM%20Main-Part%203%20Commands_v1.2_rev116_01032011.pdf`, 1 March 2011. Specification version 1.2 Level 2 Revision 116.

[113] Trusted-Computing-Group-TSS-Working-Group. *TCG Software Stack (TSS) Specification Version 1.2 Level 1.* Specification available online at: `https://www.trustedcomputinggroup.org/specs/TSS/TSS_Version_1.2_Level_1_FINAL.pdf`, 6 January 2006. Part1: Commands and Structures.

[114] P. P. Tsang and V. K. Wei. Short linkable ring signatures for e-voting, e-cash and attestation. In *In ISPEC 2005, volume 3439 of LNCS*, pages 48–60. Springer, 2004.

[115] J. Uusilehto. *How to establish mobile security.* Available online at: `http://www.mobilehandsetdesignline.com/showArticle.jhtml?printableArticle=true&articleId=196701831`, 24 December 2006.

[116] D. E. Williams and J. R. Garcia. *Virtualization with Xen: including XenEnterprise, XenServer, and XenExpress.* Syngress, Burlington, MA, c2007. Includes index.

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

[118] X. Zhang, O. Aciicmez, and J.-P. Seifert. A trusted mobile phone reference architecturevia secure kernel. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 7–14, New York, NY, USA, 2007. ACM.