FP7-ICT-217069

SEPIA

Secure Embedded Platform with Advanced Process Isolation and Anonymity capabilities

Instrument: Specific Targeted Research Project

Thematic Priority: Information and Communication Technologies

Cryptography and Privacy Protecting Technologies for Embedded Systems (Deliverable D2.1)

Due date of deliverable: October 31, 2011
Actual submission date: November 16, 2011

Start date of project: June 1, 2010 Duration: Three years

Organisation name of lead contractor for this deliverable: Graz University of Technology

Revision 0.6

Project co-funded by the European Commission within the Seventh Framework Programme (2010-2012)		
Dissemination Level		
PU	Public	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Notices For information, contact Kurt Dietrich, e-mail: Kurt.Dietrich@iaik.tugraz.at.

This document is intended to fulfill the obligations of the SEPIA project concerning deliverable D2.1, described in contract number 217069.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright SEPIA 2011. All rights reserved.

Table of Revisions

Version	Date	Description and reason	Author	Affected sections
0.1	14.09.2011	initial draft	Kurt Dietrich	all
0.2	04.10.2011	added input from IFX	Kurt Dietrich	Section 5
0.3	05.10.2011	added input from ARM	Kurt Dietrich	Section 4
0.4	06.10.2011	added new latex layout	Kurt Dietrich	all
0.5	06.10.2011	fixed table layout	Kurt Dietrich	all
0.6	06.10.2011	incorporated comments from G&D	Kurt Dietrich	all

Authors, Beneficiary

Kurt Dietrich (TUG)
Matthias Junk (IFX)
Gregory Conti (ARM)

List of Abbreviations

BCC: Brickell-Camenisch-Chen
BBS: Boneh-Boyen-Shacham
DAA: Direct Anonymous Attestation
DSA: Digital Signature Algorithm
ECC: Elliptic Curve Cryptography
EK: Endorsement Key
HUK: Hardware Unique Key
MMU: Memory Management Unit
NV: Non volatile
PBC: Pairing-based cryptography
PCA: Privacy CA
PKCS: Public Key Cryptography Standard
PSS: Probabilistic Signature Scheme
RSA: Rivest Shamir Adleman
SoC: System on chip
SCP: System Control and Power sub-system
TCG: Trusted Computing Group
TEE: Trusted Execution Environment
TLS: Transport layer security
UEFI: Unified Extensible Firmware Interface
URL: Uniform resource Locator

Table of Contents

Table of Revisions iii
Authors, Beneficiary iii
List of Abbreviations iii
Table of Contents iv
Table of Figures vi
List of Tables vii
Executive Summary 1
Purpose 1
Intended Audience 1
Background 1
1 Introduction 1
2 Privacy Enhancing Technologies for Embedded Systems 5
2.1 Analysis of existing privacy enhancing technologies 5
2.2 The Boneh-Boyen-Shacham scheme 13
3 Improvements of existing PETs 25
3.1 Revocation 30
4 Hardware Primitives for Cryptography on mobile handsets 33
4.1 The need for cryptographic instruction extension 33
4.2 ARM Cryptographic extension performance target 33
4.3 ARM cryptographic instruction set extension overview 42
5 Hardware Improvements for PETs on Security ICs 45
5.1 Introduction and Background 45
5.2 Security IC Optimisation for the SEPIA Reference Platform 45
5.3 Security IC Optimisation for the SEPIA PET Application 48
6 New developments of PETs for embedded systems 51
6.1 A model for issuer less anonymisation 51
6.2 Schnorr Signature based Approach 55
6.3 RSA Signature Based Scheme 61
6.4 Comparison of both approaches 62
7 Implementation Notes 65
8 References 67

Table of Figures

1 The TLS Handshake Protocol ... 6
2 Integration of the DAA library in the JCA Architecture ... 11
3 Key-revocation and Key-update Process ... 15
4 Computation times in relation to number of keys ... 22
5 Computation Times for 2 Keys ... 22
6 NFC Authentication Protocol Sequence ... 27
7 Architecture Overview ... 28
8 Hash-based revocation check ... 31
9 Architecture Overview ... 34
10 ARM Performance targets for 64Bytes AES128 Encryption from DDR ... 35
11 AES128 Performance expectation for IPSEC on smallest packets ... 37
12 ARM Performance targets for 64Bytes SHA256 Hashing from DDR 39
13 SHA256 Performance expectation for IPSEC on smallest packets ... 41
14 AES symmetric encryption instruction usage overview ... 43
15 SHA256 round and schedule operations overview ... 44
16 SEPIA SE Block Diagram ... 48
17 SEPIA SE Cryptographic Library ... 49
18 SEPIA ECC-based PET library ... 50
19 Schnorr Ring-Signature creation ... 56
20 Schnorr Ring-Signature verification ... 57
21 RSA ring-signature creation ... 61

List of Tables

1 Performance of the Join Protocol with Intel TPMs 9

2 Performance of DAA signature creation with Intel TPMs 10

3 Comparison of the DAA Performance of different TPMs 10

4 Signature components and length 16

5 Computation Times 21

6 Computation Times for 2 Keys 22

7 Performance comparison of the DAA sign approaches 30

8 Performance of the authentication process 30

9 Parameter lengths in number of bits 30

10 Wikipedia Internet MIX packets size distribution 33

11 Security IC functionality 47

12 Required cryptographic functionality 49

13 TPM_DAA_Sign Command Sequence 57

14 TPM_DAA_Join Command Sequence 58

15 TPM_DAA_Sign Command Measured Timings 60

Executive Summary

Anonymity and privacy protecting mechanisms are becoming more and more important. The anonymity of platforms and the privacy of users operating in the Internet are major concerns of current research activities. In this deliverable, we analyse how privacy protecting technologies and anonymous credential systems can be efficiently used on mobile devices. To achieve this goal, we investigate existing protocols, in analyse their efficiency in different use-cases. Furthermore, we analyse how these technologies can be ported to mobile environments and propose optimizations in order to increase their efficiency. Finally, we propose new paradigms for privacy enhancing technologies that include new models for anonymity protection respectively new schemes based on advanced cryptographic primitives.

Purpose

The aim of deliverable D2.1 is to provide an overview of existing privacy enhancing technologies that are based on anonymous signature schemes. The main focus lies on the Direct Anonymous Attestation (DAA) scheme which is the first and till now only standardised anonymisation scheme. Moreover, it is implemented in the current Trusted Platform Module (TPM) and is therefore, available on many computing platforms. However, in-depth analysis of the applicability of these scheme in real word scenarios and further, on mobile devices are missing. Consequently, deliverable D1.2 focuses on investigation and optimization of the DAA scheme and the underlying cryptographic building blocks.

Intended Audience

This document is intended to explain the fundamentals of privacy enhancing technologies and the role of security ICs and cryptographic hardware accelerators for privacy protection. It explains the interaction of these components and provides an overview of the requirements the privacy enhancing technologies lay on the security components of the SEPIA platform.

Background

Privacy enhancing technologies are an emerging technology. The SEPIA platform is the first mobile platform that actually takes these technologies into account - right from the start in its design phase.

1 Introduction

Anonymity and privacy protecting mechanisms are becoming more and more important. The anonymity of platforms and the privacy of users operating in the Internet are major concerns of current research activities. Although different techniques for protecting anonymity exist, mobile platforms are still missing adequate support for these technologies. In D2.1, we analyze how privacy enhancing technologies (PETs) and anonymous credential systems can be used on mobile systems.

As a starting point, we focus on the Direct Anonymous Attestation (DAA) scheme proposed by the Trusted Computing Group (TCG) as it is the first and only scheme that has gone through a standardization process so-far. Anonymity has been a major topic in Trusted Computing since its beginnings. In order to achieve anonymity protection for trusted platforms, two different concepts have been introduced by the TCG: the PrivacyCA (PCA) scheme and the Direct Anonymous Attestation scheme, both allowing trusted platforms to hide their identity when operating over the Internet. Both schemes address the problem that arises when performing public key operations. When doing such operations, a platform can always be tracked and identified by public keys and certificates that are associated with it. Both schemes address this problem, however, with different methods.

The first scheme is based on remote certification of public keys. The platform creates a temporary key-pair for each new transaction. Prior to the transaction, the public part of the key-pair has to be sent to the PCA for certification. A verifier who receives information that was signed with such a temporary key is able to verify the signature and the authenticity of the key, but he only sees the certificate from the PCA and cannot link the signature to the originating platform.

However, this approach has some severe drawbacks. Every newly created key pair has to be sent to the PCA, thereby requiring the PCA to be permanently available. Moreover, the PCA could record and store all certification requests from the single platforms and the PCA would be able to link the issued certificates to the requesting platform. This fact opens a big security leak in case the PCA gets compromised. An adversary could use this information to link transactions and signatures to single platforms. Furthermore, it is not specified how often such a certified key may be used. If the certified key is re-used for different operations, an adversary that is able to track the operations that are performed with this key could link the different signatures to the originating platform. In addition, the revocation of such credentials is still an unsolved problem. There are no mechanisms specified for revoking the certificates and if the PCA decides not to store the certification information, there is no way of revealing the real identity of the corresponding platform in case of fraud.

In order to overcome these problems, the TCG introduced another scheme. The Direct Anonymous Attestation scheme relies on local certification, thereby omitting the need for a remote party to certify the keys. It is based on a group signature scheme and Zero-Knowledge proofs, allowing each platform to create signatures on behalf of a group which can be verified by a single group-public key. The advantage of this scheme over the previously discussed one is that no on-line connection to a third party is required. Moreover, different signatures created by the same platform cannot be linked to this certain platform - not even by the group manager

which is responsible for issuing group credentials to each platform of the group. In case the issuer becomes compromised, the adversary only gets knowledge that a certain platform is part of the group managed by the issuer, other information is not stored by the group manager. With only this information, it is not possible to link any signature that has been created before the compromise happened or any signature that will be created afterwards, to a single platform. Another advantage of this scheme is that it supports the unmasking of a platform's real identity based on certain events and conditions in case of misuse. However, this scheme is based on complex cryptographic computations, making it hard to use on resource constrained devices. Moreover, the scheme requires a tamper resistant storage device for protection of the group credentials and DAA keys. These credentials must not be copied or moved to a different platform as they are issued to a specific platform. A more detailed discussion of the DAA scheme and its revocation mechanism is given in Section 2.1.

The document is organized as follows: In Section 2 , we provide a discussion of the efficiency of the standardized DAA scheme and analyze possible improvements of the protocol for mobile phones. Section 3 provides a discussion of cryptographic hardware acceleration on for mobile handsets CPUs. In Section 4 , the discussion is continued and extended to security ICs and optimizations for PETs. Finally, in Sections 5 and 6 , we discuss novel PET schemes involving elliptic-curve and pairing-based cryptography and give an outlook on future developments in PET research.

2 Privacy Enhancing Technologies for Embedded Systems

2.1 Analysis of existing privacy enhancing technologies

The support for DAA is an integral part of all TPMs since version 1.2. As TPMs are available on many desktop and notebook platforms, the question arises whether these scheme can be applied for applications and technologies other than trusted computing such as mobile-banking, mobile-government or location-based privacy. In order to analyze the efficiency of the currently standardized DAA protocol, we focus on the authentication use-case. Basically, this use-case involves the authentication of a device or person (aka. entity) against a service or service provider while hiding the real identity.
One such technology where privacy protection can effectively be applied is the Transport Layer Security protocol (RFC5246) which is used in many systems to establish secure and authenticated connections.

Related Work Several ideas for integrating anonymity protection into transport layer security 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 [7]. 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.

Test Environment For our investigations, we used several TPM Spec. 1.2 compliant TPMs, a TLS library and a crypto library. We focus on the DAA scheme for providing anonymity in secure channels and apply modifications to the crypto- and TLS library in order to provide support for DAA. Moreover, we developed

a communications library that contains the required commands and structure for invoking the DAA features of the TPM.
This approach allows us also to use our test setup on embedded systems and mobile phones which are going to be equipped with TPMs in the near future [32].
The DAA library is based on the DAA scheme which we will address as BCC05 [20], named after its authors (Brickel, Camenish, Chen) and its year of publication. In contrast to the DAA scheme defined by the TCG (which we will refer to as BCC04) in the TPM 1.2 specification, we use the BCC05 scheme because it is a performance optimized scheme that requires less parameters and less computations than the BCC04 scheme. Nevertheless, both schemes can be used with 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 [20]. Furthermore, we give performance measurement values demonstrating the performance that can be achieved using this technique in combination with currently available TPMs. Our DAA library is developed in the Java programming language and is designed to fit into the Java Cryptography Architecture (JCA), allowing easy use of the DAA scheme for developers that are not familiar with DAA and Trusted Computing. The design of the libraries and the over-all architecture of our test set-up can also be seen a novel approach for integrating anonymous digital signatures into standard cryptographic frameworks.

TLS Client Authentication TLS provides a feature that allows a server to request authentication information from clients that want to connect to it. This feature is called client authentication. Figure 1 shows the basic flow of a TLS handshake.

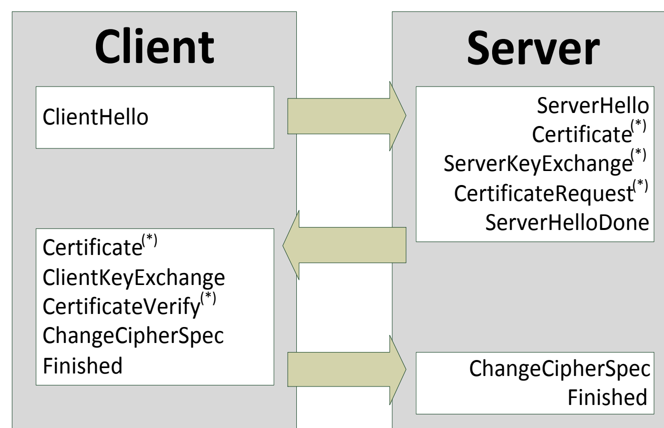


Figure 1: The TLS Handshake Protocol

The messages marked with $^{(*)}$ 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 (RFC5246).

Direct Anonymous Attestation The DAA protocol is basically a group signature scheme based on Zero-Knowledge proofs. Instead of showing a credential to a verifier like in common PKI systems, the creator of a DAA signature computes a proof that it is in possession of certain group credentials. A detailed discussion is out of scope of this document, hence, we focus on a high-level discussion of the basic protocols Join and Sign.
Before a platform can create anonymous signatures on behalf of a group, it has to join the group and obtain credentials from the group manager. Moreover, the TPM contains the secret keys $_f0$, f_1, ν' (the key f is separated in two parts for performance reasons, ν is split into ν‘ and ν“ in order to provide a shared value to do the split computation between host and TPM) that are created during the Join phase. During the join process, the client proves knowledge of f_0, f_1 to the group manager. The group manager computes the credentials (A, e, ν'') where e is a random prime and ν'' a random integer and computes a proof that A was generated correctly. The client verifies the proof on A and verifies that e is a prime in a certain interval.
After successful execution of the Join protocol, the client has obtained the credentials $(A, e, \nu = \nu' + \nu'')$ which represent a signature on the keys f_0, f_1 that are stored in the TPM. Moreover, the TPM has obtained a value ν and the platform has obtained a value ν'' which allows the TPM to create a DAA signature σ together with the platform. Details how this is achieved can be found in Section 2.1.
However, before a client may enter a group, the client and its TPM might have to be authenticated depending on the issuer policy, unless arbitrary clients are allowed to enter the group. Generally, the issuer could have a list of endorsement keys (EK) from the platforms that are allowed to join. The basic approach to identify which TPM the issuer is talking to and whether the TPM is genuine TPM or not, is to encrypt a challenge with the TPM's EK. The TPM then computes the hash of the challenge and a previously committed value. The issuer can now, by verifying the computed hash, determine if the TPM he is communicating with is a registered platform. Moreover, with the EK's certificate, the issuer can determine the vendor of the TPM and whether the TPM is a genuine one or not. This step is executed during the Join phase. The Join protocol is not part of the TLS protocol and has to be executed before a client can use anonymous TLS authentication. The issuer does not have to be the target TLS server, it can be any server, however, the client and the TLS server have to trust the issuer and its public key. When the platform has obtained all required credentials, it is ready to create DAA signatures. A signature is computed by host and TPM, allowing tooutsource most of the computations to the host platform and relieving the resource limited TPMs.
The DAA scheme can be used in two different modes of anonymity: total anonymity and pseudonymity. When using total anonymity, the creator of a DAA signature cannot be identified. This has impact on the revocation check, as malicious platforms and compromised
TPMs can also not be detected. When using pseudonymity, a verifier can detect if different signatures stem from the same platform, however, he is not able to identify the platform. How the pseudonymity mode can be used for credential revocation is discussed in the following Paragraph.

Credential Revocation

Revocation of credentials is an essential feature of common public key infrastructures. Credentials may be revoked for different reasons e.g. loss of the private key, change of the owner's name etc. The DAA scheme also provides a revocation mechanism called rogue tagging. If a TPM is compromised and the DAA credentials f_0 and f_1 become public, a verifier can detect the malicious TPM according to its pseudonym. The pseudonym depends on a basename that is defined by the verifier.

With this basename, a signer can compute its pseudonym which can be compared by the verifier with a list of pseudonyms that is computed from the list of compromised DAA keys. However, such lists can get very large as every TPM may generate an arbitrary number of different keys. A more sophisticated revocation mechanism can be establish by involving an *Attribute Revocation Authority* (ARA) [34]. The ARA is a trusted third party that can revoke the anonymity of platforms. The client encrypts its identity with the public key of the ARA. A verifier can then forward this encrypted identity which it received from a signer. By decrypting the identity, the ARA, and only the ARA, can see the signer's identity and reveal it to the verifier.

The Authentication Scheme In order to keep the implementation for our demonstrator small and simple, we do not implement the TLS extension framework. We, therefore, modify the handshake messages directly as we also want to use the demonstrator on resource constrained mobile devices.

For example, the basename for rogue tagging that is sent by the verifier is appended to the ServerHello message. Further revisions of the implementation might include support for the TLS extension framework. As discussed in Section "'TLS Client Authentication"' the TLS client, if requested by the server, sends its certificate together with a signature $h_m = H(messages)$ on the handshake messages sent so far. Thus, the server can authenticate the client and the client's key. Typically, the credential is a X.509 certificate signed by a certification authority (CA). With the DAA scheme and a TPM, it is possible to create and certify a temporary key locally on the client by signing it with a DAA signature instead of using a certificate from a CA to authenticate it. In our prototype, we only sign the key and send the key plus the DAA signature to the host instead of a X.509 certificate and the according chain. The certificate has no value in this case as it must not contain any information (e.g. subject or serial number) that might compromise the platform's anonymity. The signature on h_m is done by the temporary key, as discussed in the introduction Section.

Another approach could be to sign h_m directly with a DAA signature, however, the results from Table 2 clearly show that this is not a good approach with common TPMs (except the Intel TPM). The low performance of TPMs prohibits the direct use of the DAA scheme for signing information that is used imminently. A better approach is to create temporary keys and sign and certify them prior to opening a TLS channel. The authentication key can be a public key of any common signature scheme - it may be a RSA key, an ECC or DH key parameters. For extra security - note that when using the discussed pre-certification, the certified key might be stolen from the platform before it can be used - the temporary authentication key

could be created and secured by the TPM. The TPM also provides support for RSA signatures and RSA key-pair generation. These features can be used to create temporary keys and to sign h_m. In this case, the TPM provides additional security as it can create, store and operate with the temporary key in a secure and tamper resistant environment. The speed of this sign operation is between one and two seconds which is sufficient for the signature of the client authentication.

Implementation Aspects

In order to demonstrate our results, we developed a library that provides the cryptographic operations for the BBC05 scheme and the DAA commands as defined in the TPM command specification. In detail, we implemented the following TPM commands: TPM_DAA_Sign, TPM_DAA_Join, TPM_FlushSpecific, TPM_OIAP, TPM_Terminate_Handle and the following structures: TPM_DAA_ISSUER, TPM_NONCE.

Support for the modular arithmetic operations, the RSA-OAEP encryption scheme and the DAA protocol operations is provided by the IAIK-JCE-MicroEdition [27]. Details of our implementation of the cryptographic functions and the BCC05 scheme can be found in [10]. 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 our DAA library allows to abstract the complex API definitions in the TSS 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 user space 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.

Performance Evaluation

The DAA scheme involves complex mathematical computations, hence, it is of interest which performance can be achieved with currently deployed TPMs. Table 1 shows the performance of the Join protocol 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).

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 1: Performance of the Join Protocol with Intel TPMs

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. We have also performed tests with TPMs from ST Micro and Intel. The results

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 2: Performance of DAA signature creation with Intel TPMs

are shown in Table 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 smart card controller that is attached to the PC's motherboard via the LPC bus [14], the Intel TPM is located in the Intel motherboard chipset (i.e. the Intel 82801JDO Controller Hub (ICH10DO)) itself [15].

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 3: Comparison of the DAA Performance of different TPMs

Most of the time is consumed by the modular exponentiations and the paramter 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 [33].

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 Java Cryptography Architecture Framework (JCA) [29]. The advantage of using such a common framework is that for creating signatures, the same API, independent from the signature scheme can be used. Figure 2 shows how our DAA implementation is integrated into the JCA framework. The application uses the iSaSiLk library [26] in order to open a secure channel.

The library uses the JCA framework to request a certain algorithm implementation according to the TLS session parameters that have been negotiated during the

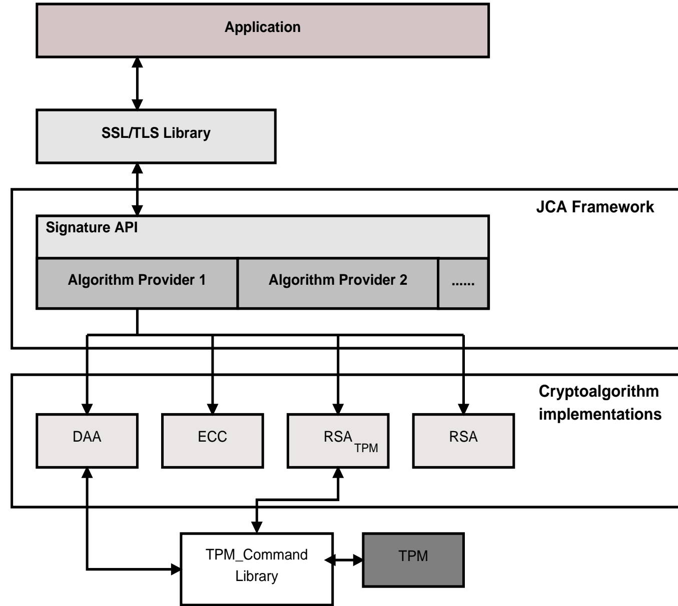


Figure 2: Integration of the DAA library in the JCA Architecture

TLS handshake. The algorithm implementation can be either a software implementation of the algorithm or an interface to some hardware device. Our DAA signature implementation consists of two parts: the host part and the TPM part. Both parts are abstracted by the signature class API. The host part performs the DAA computations that can be done in software while the TPM part handles the communication with the TPM. For using the DAA sign function of the TPM, the TPM_DAA_Sign [34] 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 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. In our implementation, the computation of the DAA signature is done according to the BCC05 scheme algorithm which works as follows:

1. If a *basename* is included in the ServerHello request, the verifier requests rogue tagging and the host computes $\zeta = H(basename)^{(\Gamma-1)/\rho} \mod \Gamma$ which is sent to the TPM.
2. The host part of the signature class computes: $T = AS_0^{w_0}S_1^{w_1} \mod n$ 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: $\tilde{T}_t = R_0^{r_{f_0}}R_1^{r_{f_1}}S_0^{r_{v_0}}S_1^{r_{v_1}} \mod n$ with r_{f_0}, r_{f_1} of size $l_f + l_\phi + l_H$ bits and $r_{v_{(0,1)}}$ 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: $\tilde{T} = \tilde{T}_t T^{r_e} S_0^{r_{\bar{v}_0}} S_1^{r_{\bar{v}_1}} \mod n$ 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 verfied 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 $S_1 = S^{2^l}$ and $\nu = \nu_0 + \nu_1 * 2^l$ as the crypto co-processor of the TPM cannot compute the modular exponentiations if the exponent is larger than the modulus n.
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. The following listing gives an overview on 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. The 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( "Test data".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 kByte of size. In addition, the TLS implementation is about 120 kBytes resulting in total of 270 kBytes which allows efficient usage on mobile platforms. Performance results of our implementation on mobile devices can be found in [10].

A Note on Specification Compliance

In the TPM specification [33], the commands and structures for using the DAA functions of TPMs are defined. However, during our experiments we realized that the TPM vendors have different interpretations of this specification. We have tested our library 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 TPMs from specific vendors. 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 a big problem, the emulator, respectively its code basis, is used in many implementations such as mobile TPMs or virtual TPMs for XEN [35]. 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 our experiments, we failed 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.

Conclusion and Future Work

In this paper, we elaborate on different topics: we discuss how a anonymous credential systems based on TPMs can be integrated into the JCA architecture and how they can be used for anonymous TLS client authentication. Moreover, we show which performance 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.

2.2 The Boneh-Boyen-Shacham scheme

In addition to the previously investigated DAA scheme, we investigated another anonymisation scheme invented by Boneh-Boyen-Shacham (BBS) that is based on elliptic curve cryptography. Although it is not standardized, the scheme provides

an interesting revocation mewchanisms that is, in contrast to the DAA's pseudonym scheme based on key re-issuance. Therefore, we present the findings obtained from our proof-of-concept implementations of the Boney-Sacham (BBS) Signature Scheme [5]. We give an overview over the problems which arise when implementing this scheme on various platforms. Furthermore we show which cryptographic libraries can be put to use for which platform and give an insight into the signature length and the computation times of the protocol.

Background

We implemented the BBS Protocol [5] in J2SE, on a Gumstix XL6P running a Linux v.2.6.11 kernel and on a Philips LPC2138 controller. Due to the fact that the BBS Protocol requires pairings on elliptical curves only a few libraries suited our needs. For our J2SE reference implementation we made use of the jpbc open-source library [8] which is a java port of the pairing based cryptography library [17] which was the base for our implementation on a Gumstix verdex pro XL6P. For the Gumstix implementation we also needed to cross compile the gnu multiple precision arithmethic library [1] and the pbc_sig library [19]. The implementation on the LPC2138 is based on the Miracl library [24].

Setup

For our proof-of-concept implementations we use a type A pairing as proposed by Lynn [18]. The generator point $g_1 \in G1$ is randomly chosen and $g_2 \in G2$ is set equally to it, as $G1$ and $G2$ are the same set for type A pairings. The points u, v and h should satisfy equation 1 (with $\xi_1 \in \mathbb{Z}_p$ and $\xi_2 \in \mathbb{Z}_p$ representing the secret key components of the group manager).

$$u^{\xi_1} = v^{\xi_2} = h \tag{1}$$

In order to get the needed points h is randomly chosen and taken to the power of ξ_1^{-1} for u and ξ_2^{-1} for v. Then the group manager randomly chooses a $\gamma \in \mathbb{Z}_p$ and computes $w = g_2^{\gamma}$ which makes the group setup complete. For each private key a x is randomly chosen out of $\mathbb{Z}_p$. With this x and with the γ above the second private key component can be computed: $A = g_1^{(\gamma+x)^{-1}}$.

Key Revocation

In this section we want to outline some important details about the revocation step. Figure 3 shows the fundamental design of the revocation process. When the certification authority wants to revoke a private-key it - in case the key becomes public - stores it on a revocation list (RL). In case the key to be revoked is not up to date at revocation time the CA has to update it before inserting it into the RL.

The authors of the original protocol [5] propose to keep the tuple $(\psi(A), x)$, with ψ being a computable isomorphism from $G2$ to $G1$ ($\psi(g2) = g1$), on the revocation list instead of (A, x). Due to the fact that we have a type A [18] setup ($\psi(g2) = g2 = g1$) we directly store (A, x) on the revocation list.

At the key update phase the user receives the revocation list from the CA. In order to update his private key he has to take all private keys into account which have not been considered in a previous update phase. Please note that the order of the revocations matters at update time, so the keys on the revocation list have to be processed according to their revocation time (from old to new).

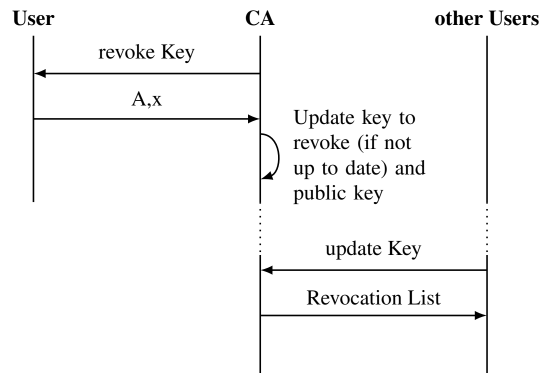


Figure 3: Key-revocation and Key-update Process

Signature length

A signature over a message consists of three elliptic curve points and six large integers. This leads to the signature length presented in table 4 ($|x|$ denotes the bit length of x).

In our setup (160Bit Curve) an elliptical curve point needs 320 Bits and a large integer needs 160 Bits memory. So we achieve a total signature length of 1920 Bits or 240 Bytes.

Setting up the Development Environment

J2SE In order to set up the development environment for J2SE all you need is to have the jpbc library [8] (the jpbc library is an open-source library) on the classpath. As this setup is very simple further details are omitted.

Gumstix If one wants to develop software for a Gumstix controller he has two opportunities, namely setting up a qemu which emulates a Gumstix environment or developing natively on the Gumstix controller and debugging over a serial terminal. In both cases a cross compiler toolchain has to be set up. Furthermore the needed libraries have to be compiled. In the following sections we describe how to set up the needed tools.

Name	Type	Size
T_1	Elliptical Curve Point	$\lvert G_1 \rvert$
T_2	Elliptical Curve Point	$\lvert G_1 \rvert$
T_3	Elliptical Curve Point	$\lvert G_1 \rvert$
c	Large Integer	$\lvert \mathbb{Z}_p \rvert$
s_α	Large Integer	$\lvert \mathbb{Z}_p \rvert$
s_β	Large Integer	$\lvert \mathbb{Z}_p \rvert$
s_x	Large Integer	$\lvert \mathbb{Z}_p \rvert$
s_{δ_1}	Large Integer	$\lvert \mathbb{Z}_p \rvert$
s_{δ_2}	Large Integer	$\lvert \mathbb{Z}_p \rvert$
		$3 \cdot \lvert G_1 \rvert + 6 \cdot \lvert \mathbb{Z}_p \rvert$

Table 4: Signature components and length

Qemu We assume that a qemu, capable of emulating arm systems, is running on your system. In order to get a gumstix image running in qemu we firstly need to obtain a u-boot image [1], a root file system [2] and a kernel image[3]. If all files are on your harddisk you can create an image for qemu as shown in the following listing.

```
dd of=flash.img bs=128k count=256 if=/dev/zero
dd of=flash.img bs=128k conv=notrunc if=\$1
dd of=flash.img bs=128k conv=notrunc seek=2 if=\$2
dd of=flash.img bs=128k conv=notrunc seek=248 if=\$3
```

In order to run this image in qemu the command provided in the following listing can be used. However this script only works if the path to qemu-system-arm is on your path and the command is executed from the directory where your `flash.img` file is stored.

```
qemu-system-arm -M verdex -pflash flash.img
 -nographic -serial pty
```

Once you have started qemu using this command a terminal is is provided over `/dev/pts/*`.

[1] http://cumulus.gumstix.org/feeds/u-boot/u-boot-verdex-400-r1604.bin, http://cumulus.gumstix.org/feeds/u-boot/u-boot-verdex-400-r1604.bin

[2] http://cumulus.gumstix.org/feeds/current/glibc/images/gumstix-custom-verdex/Angstrom-gumstix-basic-image-glibc-ipk-2007.9-test-20080512-gumstix-custom-verdex.rootfs.jffs2, http://cumulus.gumstix.org/feeds/current/glibc/images/gumstix-custom-verdex/Angstrom-gumstix-basic-image-glibc-ipk-2007.9-test-20080512-gumstix-custom-verdex.rootfs.jffs2

[3] http://cumulus.gumstix.org/feeds/current/glibc/images/gumstix-custom-verdex/uImage-2.6.21-r1-gumstix-custom-verdex.binhttp://cumulus.gumstix.org/feeds/current/glibc/images/gumstix-custom-verdex/uImage-2.6.21-r1-gumstix-custom-verdex.bin

Establishing a connection using Kermit

If you are running a debian flavoured linux distribution you can get Kermit over your package management system. Alternatively you can grab it from hereOnce Kermit is installed you can start it over:

```
kermit -l /dev/DEVICE_TO_CONNECT_TO
```

After this the connection has to be configured (see here

```
set speed 115200
set reliable
fast
set carrier-watch off
set flow-control none
set prefixing all
set file type bin
set rec pack 4096
set send pack 4096
set window 5
connect
```

Sending files If you are connected to the Gumstix controller you can set it into receive mode by starting the program `rz`. After this you can go back to the kermit shell and send the file using the `send` command. If you are running a real device the files can be copied to the SD-Card and accessed over `/mnt/card/`.

Cross compiler toolchain

The provided image uses the glibc v2.5 as C library for userspace programs. As we want to omit static linkage of the libc we looked for a compiler which uses the same version of glibc. It turned out that such a compiler is part of the scratchbox project The required compiler is scratchbox-toolchain-cs2007q3-glibc2.5-arm7. Once you have installed it and the bin-folder on your path you are ready to build executeables for the gumstix image.

Building the needed libraries

In the following sections we provide the commands which are required to build the needed libraries. We assume that the bin folder of the previously installed compiler is on your path. Once all libraries are built they have to be copied to the Gumstix image. To get a program, which is built atop one of these libraries, running their path has to be provided over the environment variable `LD_LIBRARY_PATH`.

Building the GMP Library

```
./configure --host=arm-none-linux-gnueabi
--prefix=PATH_TO_YOUR_BUILDDIR
make
make install
```

Building the PBC Library

```
./configure --host=arm-none-linux-gnueabi
--prefix=PATH_TO_YOUR_BUILDDIR
make
make install
```

Building the pbc_sig Library

```
./configure --host=arm-none-linux-gnueabi
--prefix=PATH_TO_YOUR_BUILDDIR
make
make install
```

Philips LPC2138 with Keil Eval Board MCB2130

In this section we explain how to set up the needed tools and libraries for a Philips LPC2138. For our explanations we assume that there is no OS running on the target controller and the program is directly called by the startup script.

Keil μVision 4.0

The most comfortable way to build programs for the LPC2138 is to use Keil μVision 4.0. An evaluation version can be obtained from Unfortunately the evaluation version only links executables up to 40KB code size, but there exists a quite simple workaround shown in the following paragraph.

CodeSourcery Toolchain Due to the 40KB issue mentioned above we make use of the CodeSourcery toolcain. A Windows-Installer can be downloaded from here

As the GNU Assembler Syntax doesn't conform to the one used by the Assembler which is shipped with μVision 4.0 we have to provide our own startup script. Furthermore a own linker script has to be provided. These files are attached to the site of Application Note 199 at the Keil website. However we added some things (e.g. end symbol in linker script and sbrk in syscalls file for heap support) and therefore we recommend using the files located in our repository.

Once the toolchain is installed it has to be selected in the *Components, Environment and Books* Wizard in μVision 4.0. Also the path to the linker script has to be supplied in the *LD*-Options of your target. Finally the startup-Script has to be added to your project.

Syscalls, Serial

We also provide the (slightly refined) files `syscalls.c` and `serial.c` from Application Note 199 in our repository. These files are needed for stdout over the serial port 1 and for a few other functions such as heap support. If you want to make use of these files simply add them to your project.

The MIRACL Library

Due to the fact that we need to compute pairings over elliptical curves we had get a library which supports such things running on our controller. As there are only 512KB of flash memory and 32KB of ram available on the LPC2138 our decision fell on the MIRACL library, which was said to be suited for very constrained environments. The Miracl library is a C library with a few C++ high level interfaces. Owing to the thight memory constraints on the LPC2138 we had to go for a C only build. Unfortunately all the pairing functions are implemented in C++ so we had to port them to C (see `pairing1.c`, `pairing1.h`).

Moreover the MIRACL library is very flexible. There is a file (`mirdef.h`) where all options can be specified to suit your needs and to get the maximal performance. In the following listing the mirdef - File we used is shown.

```
#define MIRACL 32
#define MR_LITTLE_ENDIAN
#define mr_utype int
#define MR_IBITS 32
#define MR_LBITS 32
#define mr_dltype long long
#define mr_unsign32 unsigned int
#define mr_unsign64 unsigned long long
#define MAXBASE ((mr_small)1<<(MIRACL-1))
#define MR_ALWAYS_BINARY
#define MR_STRIPPED_DOWN
#define MR_SPECIAL
#define MR_GENERALIZED_MERSENNE
#define MR_NO_LAZY_REDUCTION
#define MR_NOASM
#define MR_COMBA 5
```

ASM optimizations It is also possible to compile the MIRACL library with assembler optimizations. If such optimizations are available for your target platform you can do so by simply compiling and running the program `mex` which is distributed with the MIRACL library. For our setup we used: `./mex 5 armgcc mrcomba`.

If you get the following error when you are trying to compile `mrcomba.c` try to disable ARM Procedure Call Standard in the compiler options for your target.

```
./mrcomba.c: In function 'comba_mult':
./mrcomba.c:118:3: error: can't find a register in class
 'GENERAL_REGS' while reloading 'asm'
./mrcomba.c:118:3: error: 'asm' operand has impossible
 constraints
```

Redirecting Stdout For debugging reasons we slightly changed the `mrio1.c` file to redirect the stdout to the serial port 1 of the controller. This file is also supplied in our git repository.

Adding the MIRACL Library to your project The easiest way to add the MIRACL library to your project is to add all files provided in the `miracl_port/src` directory of our repository to your project. If you need some custom functionality we recommend to download the whole library from here

Implementation Aspects

J2SE

In order to have a reference implementation for our Gumstix implementation we decided on a J2SE implementation using the jpbc library [8]. As we were interested in timings on smaller hosts we do not provide any further information about this proof-of-concept implementation in this report, even though the jPBC library would be well suited for the implementation of the host part.

Gumstix

Our Gumstix implementation is written in C and is based on the pbc_sig [19] library. This library is built upon the pbc [17] library which in turn relies on the gmp [1] library. The pbc_sig library already provides an implementation of the BBS signature scheme which supports join, sign, verify and open. So all we had to do was to add revocation support.

Philips LPC 2138

For the implementation on the Philips LPC2138 controller we make use of a C-only build of the Miracl library [24]. In order to be able to compute pairings on elliptical curves we had to port the highlevel C++ pairing functions to C (see `pairing1.c`, `pairing1.h`). We also introduced a file `bbs.h` containing all datatypes and function prototypes needed for our implementation of the BBS protocol.

Performance Evaluation

In this section we show the performance evaluation results of our implementation of the BBS Signature Scheme [5].

Gumstix

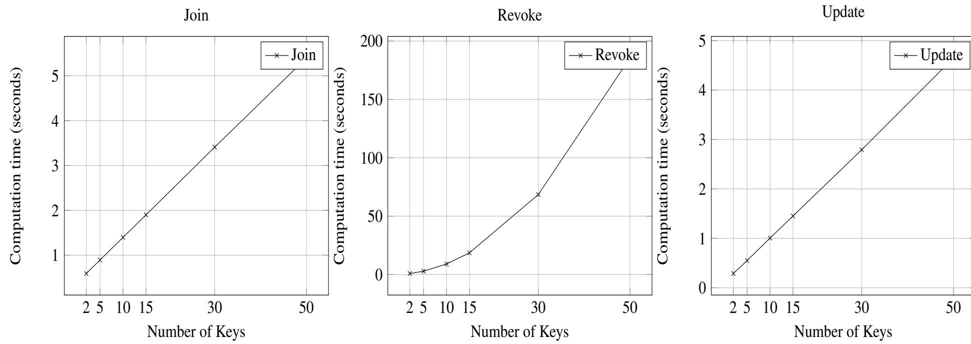
Each timing value presented in the following section is an average over 10 runs of the particular protocol step. The computation times of Join and Revoke are cumulative times for #*Keys* joins/key revocations. The other times correspond to one Sign/Verify/Open process.

We can clearly see that all computations except Revoke and Join are independent of the key amount. Therefore we omitt the plots for these operations. As expected Join increases linear with the key amount, so we do not further investigate it. Nevertheless we want to take a closer look at the revocation step. The presented revocation values correspond to the times needed for the revocation of $n-1$ keys where the revocation of the $n-k$th key implicts an update of $k-1$ keys, which basically takes $O(n!)$ time. But as in real world applications not all keys are revoked at a time update and revocation of one key can be computed in $O(n)$ time.

Table 5 shows the computation times of each protocol step for the different number of keys. Figure 4 gives a graphical overview over our results.

# Keys	2	5	10	15	30	50
Join	591ms	893ms	1395ms	1897ms	3410ms	5398ms
Revoke	872ms	2849ms	8986ms	18563ms	68456ms	185520ms
Update	288ms	549ms	1005ms	1451ms	2792ms	4648ms
Sign	494ms	493ms	496ms	495ms	499ms	496ms
Verify	457ms	458ms	460ms	462ms	463ms	459ms
Open	716ms	716ms	719ms	722ms	726ms	722ms

Table 5: Computation Times



(a) Issuing of group and secret keys (b) Revocation of $n-1$ keys (c) Update 1 key with

Figure 4: Computation times in relation to number of keys

Philips LPC 2138

In this section we show our results for the Philips LPC2138. Due to the very limited memory we can only provide timing values for 2 Keys. Another issue was that we had no timer implementation on the LPC2138, therefore only approximate timing values are presented.

The provided times (see table 6, figure 5) correspond to a run without pairing precomputation, one with pairing precomputation and one with pairing precomputation and enabled assembler optimizations. We can see that the precomputation of the pairings saves the most time.

	Join	Update	Sign	Verify	Open
w/o precomputation	9s	2s	11s	19s	1s
w precomputation	13s	7s	7s	15s	1s
w precomputation and ASM optimizations	13s	6s	6s	14s	1s

Table 6: Computation Times for 2 Keys

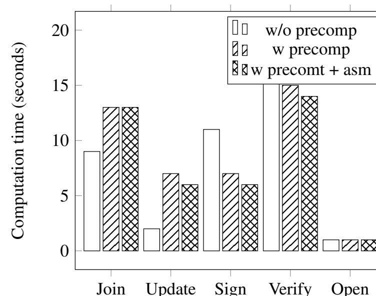


Figure 5: Computation Times for 2 Keys

Conclusion

In this report we present timing aspects and other protocol related findings obtained from our BBS implementations. We can achieve signature lengths of 240 Byte using a 160 Bit type A setup [18]. Further we can see that the computation times on a Gumstix XL6P are absolutely acceptable, although a Gumstix controller might be too large for real world applications. Moreover we conclude that the computation times on the LPC2138 are acceptable if we exploit the all optimizations (precomputation, asm optimizations) as *Update* and *Sign* are fast enough and they are the only operations which would, in real world applications, be carried out on the client side in a recurring manner.

3 Improvements of existing PETs

In this section, we analyze how PETs can be improved in the sense of optimizing the relevant algorithmic computations. Moreover, we propose a design for implementing this modified scheme on mobile phones that are equipped with security ICs, in this case Secure Elements (SE). Furthermore, we discuss a design that allows the application of the DAA scheme for contactless, anonymous authentication scenarios, with respect to secure storage, authorized access to the DAA credentials and sufficient performance. We show 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, we use off-the-shelf devices to demonstrate our efforts in order to support this statement. Our 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 modified DAA Sign Protocol

We modify the DAA sign algorithm in the sense that for computing the DAA signature, we can reduce the parameters involved. Instead of computing $R^r 1 * R^r 2 \bmod n$ we just compute R^r and increase the range of r respectively f. r and f have to be chosen as $r > f$ and $f \geq 160 bits$ and $r_l <$ than the largest value that is accepted by the crypto-coprocessor. This way, we can save one modular exponentiation and one modular reduction. Moreover, the private key f only requires 160 bits instead of 2*104 bits. This approach does not affect the security of the overall algorithm as f is sufficiently large.

Algorithm 1 shows the basic steps for computing a DAA signature. n is an RSA modulus with $n = p * q$ an $p = 2p' + 1, q = 2q' + 1$. S_1 is a random quadratic residue $\bmod n$ that generates the bases $R, S_2, Z \in S_1$. For further details on the parameters we refer to [20].

Rogue Tagging: In order to identify malicious TPMs, the TCG has introduced the so-called *rogue tagging* mechanism. Algorithm 2 shows the basic steps for computing the rogue tagging pseudonym of the signer.

If a TPM is compromised and its private-key f becomes publicly known, a verifier can identify the compromised TPM via the pseudonym ζ [34]. The verifier holds a list with all known keys f, and computes the pseudonym values of these keys with

Algorithm 1 DAA Signature Creation

Require: $R, S_0, S_1, Z, \nu_0, \nu_1, n, f, \Gamma, \rho$.
Ensure: $\sigma = (T, c, n_t, s_{\bar{\nu}}, s_e, s_f, s_{\nu_0}, s_{\nu_1})$

1: The *host* selects a random w and computes: $T = A * S^w$, note: $S = S_1 * S_2^{2^{l_s}}$ and $\nu = \nu_1 + \nu_2 * 2^{l_s}$ for $l_s = 1024$
2: *Host* and *TPM* compute the "signature of knowledge":
3: The *TPM* computes: $\tilde{T}_t = R^{r_f} S_1^{r_{\nu_1}} S_2^{r_{\nu_2}} \bmod n$ with random r_f and r_ν
4: The *host* computes: $\tilde{T} = \tilde{T}_t T^{r_e} S^{r_{\bar{\nu}}} \bmod n$
5: and: $c_h = H(n \| R \| S_1 \| S_2 \| Z \| T \| \tilde{T} \| \Gamma \| \rho \| n_\nu)$
6: The *TPM* selects a random n_t and computes computes $c = H(H(c_h \| n_t) \| m)$ and $s_f = r_f + c * f$, $s_{\nu_0} = r_{\nu_0} + c * \nu_0$, $s_{\nu_1} = r_{\nu_1} + c * \nu_1$,
7: The *host* computes $s_e = r_e + c * (e - 2^{l_e - 1})$ and $s_{\bar{\nu}} = s_\nu + r_{\bar{\nu}} - cwe$
8: Finally, the *host* assembles the signature $\sigma = (T, c, n_t, s_{\bar{\nu}}, s_e, s_f, s_{\nu_0}, s_{\nu_1})$

Algorithm 2 DAA Rogue Tagging

Require: *basename*, Γ, ρ.
Ensure: ζ.

1: The verifier selects a random *basename* and sends it to the signer
2: The *TPM* computes $\zeta = H(basename)^{(\Gamma-1)/\rho} \bmod \Gamma$

respect to Algorithm 2 and the basename he provided. If ζ is on the computed list, the verifier knows that the signer was a malicious TPM.

Implementation options

The slow performance of the DAA computations executed inside the SE show that it is inevitable to include a host that contributes to 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 device or the SE can be used to generate RSA key-pairs - which we address as ephemeral authentication keys (EAKs) from now on - which 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.
We investigate two approaches how anonymous signatures can be used for authentication: 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 Algorithm 1 is executed entirely inside the TPM. The pre-computation of the certified EAK key-pairs can be executed on the card without further involvement of the host. In the second approach, the signature computation (see Algorithm 1) is partially computed inside the TPM and partially on the host. Details of the implementation and performance results can be found in Section 3.
However, in both approaches, the rogue tagging value cannot be computed in advance as it depends on a base value created by the verifier. Although a DAA signature, according to the specification [33], contains the actual signature and the computed rogue tagging value, both computations are rather independent crypto-

graphic operations (only the computed hash c - see 1 contains the rogue parameters). Hence, they can be computed separately.

Anonymous Authentication Scenario

Figure Figure 7 shows the application of our 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, nu_0 and nu_1, R_0, S_0, S_1). By loading different credentials, the TPM (or in MTM in this case) may create DAA signatures on behalf of different groups it has joined before.

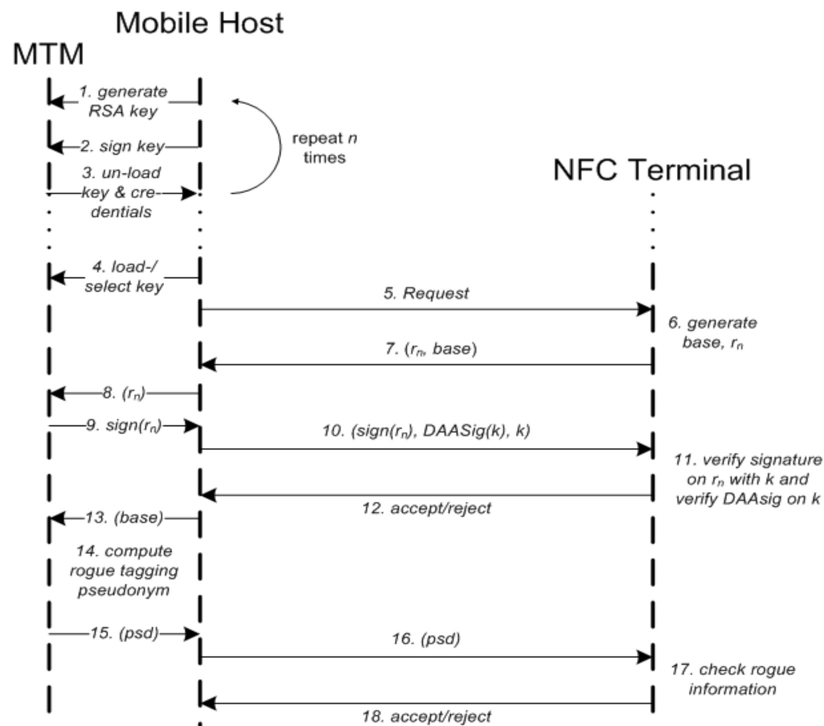


Figure 6: NFC Authentication Protocol Sequence

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 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 [20].

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.

Proof of concept prototype

For our experiments, we used a Nokia 6212 NFC mobile phone. This phone is equipped with a Giesecke & Devrient SmartCafe smart-card as SE. Our 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 17 shows our concept. The host application uses a TPM command library to issue commands which are sent to the SE via application protocol data units (APDUs).

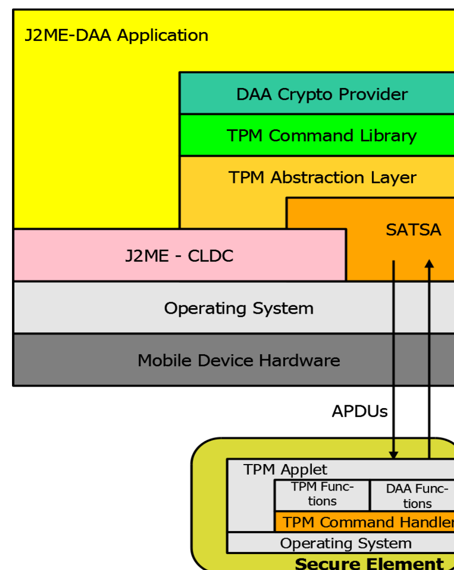


Figure 7: Architecture Overview

The host part is implemented as a Java2MicroEdition (J2ME) application that allows the installation of mobile applications on the phone. Moreover, we take advantage of the Security and Trust Service API (SATSA) [30] respectively of JSR 257 the *Contactless Communication API* [21] which allows our 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, we require several TPM commands and structures as defined in [33] as well as support for different algorithms. The JavaCard 2.2.1 environment does not provide support for implementing cryptographic protocols. We follow the ideas from [25] and [4] concerning algorithm implementations on JavaCard. 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.
Our 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, our 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 our approach, we use 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.

The Pre-computation Step

The TCG specifies two commands *TPM_DAA_Join* and *TPM_DAA_Sign* which are executed repeatedly in different stages [33]. For simplicity reasons, we reduce these stages to a single stage.

Table 7 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 7 shows a slight performance advantage when computing the entire DAA signature in the TPM[4]. 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.

A detailed performance analysis of the single cryptographic steps (i.e. random number generation, hash operations, modular exponentiation etc.) of the JavaCard

[4]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 7: Performance comparison of the DAA sign approaches

applet and the Java application on the host can not be given in this version of the paper due to its length restrictions. Note that all performance measurements are average values that were estimated by 100 executions of the single operations.

The NFC Authentication Step

For the actual authentication over the NFC channel, we can use the certified EAK-keys from the pre-computation step. As shown in Figure 6, the terminal sends a nonce to the mobile/TPM which basically applies an RSA signature according to PKCS#1.5 [16] 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 8: 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 requires 3,4 seconds.

Parameter lengths In our prototype implementation, we use the following parameter lengths:

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 9: Parameter lengths in number of bits

3.1 Revocation

Revocation is a very important mechanism when using PKI like structures. However, applying revocation on anonymous credentials is complex. In the following paragraph, we discuss different revocation mechanisms that can be used instead of the original proposals. We propose to use revocation mechanisms based on symmetric cryptography instead of using the costly pseudonym computation proposed

in the current schemes. Traditional revocation is based on the idea that f becomes public. However, how likely is that this happens? An attacker that gets hold of the private key wont publish the key. Moreover, there no mechanism even for the owner of the TPM to revoke his own key and to publish it. We assume that it is more likely that it becomes publicly known which platform and with it, which TPM has been compromised or misbehaves. Hence, 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.

The proposed algorithm is shown in the following listing:

1. the verifier generated as random number r_V and sends it to the prover.
2. the prover generates a random number r_P and computes $PS = HMAC(f, U \| r_V \| r_P)$ and returns PS, r_P to the verifier.
3. the verifier checks if PS in $HMAC(f, U \| r_V \| r_P)$ (The verifier holds a table with all revoked (U_i, f_i)

Figure 8: Hash-based revocation check

Maintaining backward secrecy The protocol may be modified in order to provides backward-secrecy. Instead of publishing the actual signing keys and credentials f_i, U_i, only the hashes of these paramters are published on *RL*.

In order to prevent the disclosure of signatures and their signers which have been created before the publication of *RL*, the revocation info (U, f) may be published as a list of

It is crucial not to post the actual revoked U, f pairs, because otherwise an adversary may use U, f to create a signature σ and try to get access via terminal which has no updated *RL* yet.

4 Hardware Primitives for Cryptography on mobile handsets

4.1 The need for cryptographic instruction extension

Todays SoC implement dedicated hardware cryptographic accelerators for DMA driven SHA and AES operations. Despite their limited power consumption they only support a very limited number of use cases and are quite difficult to share between the TEE and the RichOs applications. As matter of fact, todays SoC generally implement several instances of the same hardware cryptographic engine to solve these sharing issues. Moreover, each SoC vendor have their own cryptographic engine leading to very high fragmentation and create difficulties for the RichOS SW teams to handle these different implementations and use theses accelerations into standard SW cryptographic framework such as openSSL.
The ARM cryptographic instruction extension is aimed to reduce the fragmentation and to ease the SW development and usage of cryptography. It is also aimed to provide better performance than dedicated hardware accelerators by decreasing the time needed for context switching applications that uses cryptographic operations and by providing a better and faster support for cryptographic protocols such as IPSec that involve small packets (64Bytes).
As a matter of fact, small packets cannot be handled efficiently by current SoCs DMA driven hardware cryptographic acceleration due to the time needed for reconfiguring the different engines between packets and the lack of coherency and memory caching that introduce high latencies, especially when operating on DDR memories.
The following table describe the packets size distribution in a typical internet flow.

Packet size (bytes)	Nr. Packets	Bytes	Distribution
64	67	4288	57,00%
570	1	570	7,00%
594	2	1188	16,00%
1518	1	1518	20,00%

Table 10: Wikipedia Internet MIX packets size distribution

4.2 ARM Cryptographic extension performance target

In the following example, an ARM CortexA CPU running at 1.2 GHz is integrated in a SoC that have a DDR access latency of 100ns for reading a 128bits data. The ARM CortexA CPU has also an access latency of 10ns to read a 512bits cache line of data from its L2 cache.

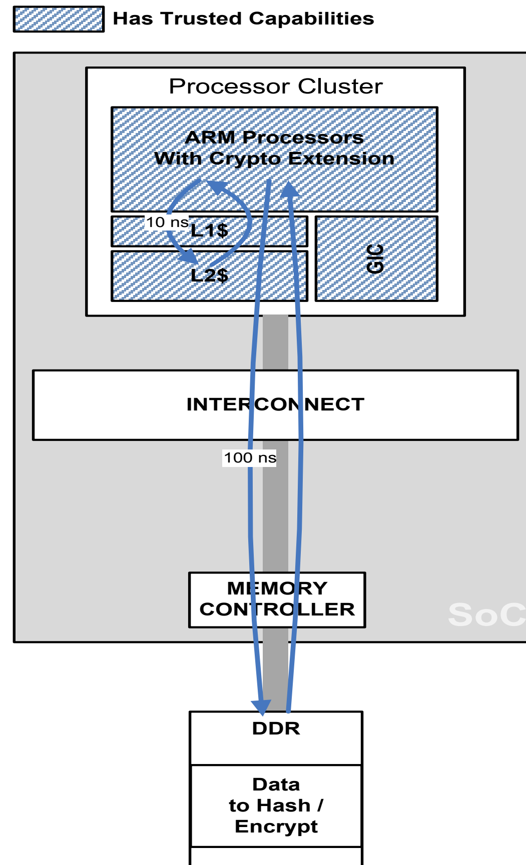


Figure 9: Architecture Overview

AES128 performance targets

The first encryption scenario consists on reading 512bits of data from the DDR and performing AES128 CBC encryption on it. The standalone AES hardware accelerator performance required at SoC level by the OEM vendors is 100Mbytes/s for 512bits processing. In order for the ARM Cryptographic extension to reach the objective of defragmentation and standardisation, the following performance target must be reached.

The following calculus shows the CLK/Byte performance the ARM cryptographic extension need to reach to fulfill OEM requirements:

512bits Encryption(AES128) from DDR on a CortexA with 64Bytes Cache Line Size							
CPU Frequency (Mhz)	ns	**DDR Access Latency (ns)**	AES128 Crypto ISA (CLK/Byte)	Cost for 64Bytes (CLK)	Latency between DDR 128bit element (DDR CLK=4*MPU CLK)	Total cost: AES128 on 512bits + 2xDDR Access + Burst 4 Elements	**Mbytes/s**
1200	0.83	**100**	7	448	4	704	**104.0371982**

Figure 10: ARM Performance targets for 64Bytes AES128 Encryption from DDR

The second scenario is the IPSec acceleration for which SoC hardware accelerator are relatively bad at due to high number of 64Bytes packets (57%) contained in the IPSec flow. Assuming that the target of 7CLK/Byte is easy to achieve for a reasonable hardware cost, the performance benefit on IPSec types of protocols is expected to be important since they highly benefit from L2 caching on the contrary of dedicated hardware accelerators.

512bits Encryption(AES128) from L2$ on a CortexA with 64Bytes Cache Line Size							
CPU Frequency (Mhz)	ns	**L2 Latency (ns)**	AES128 Crypto ISA (CLK/Byte)	Cost for 64Bytes (CLK)	Latency between each 64Byte $line (L2$ CLK=MPU CLK)	Total cost: AES128 on 512bits + 2xL2$ Access + Burst 4 Elements	**Mbytes/s**
1200	0.83	**10**	7	448	0	472	**155.1741261**

Figure 11: AES128 Performance expectation for IPSEC on smallest packets

SHA256 performance targets

The first hashing scenario consists on reading 512bits of data from the DDR and performing a SH256 hash on it. The standalone SHA256 hardware accelerator performance required at SoC level by the OEM vendors is 150Mbytes/s for 512bits processing. In order for the ARM Cryptographic extension to reach the objective of defragmentation and standardisation, the following performance target must be reached.

The following calculus shows the CLK/Byte performance the ARM cryptographic extension need to reach to fulfil OEM requirements:

512bits Hashing (SHA256) from DDR on a CortexA with 64Bytes Cache Line Size							
CPU Frequency (Mhz)	ns	**DDR Access Latency (ns)**	SHA256 Crypto ISA (CLK/Byte)	Cost for 64Bytes (CLK)	Latency between DDR 128bit element (DDR CLK=4*MPU CLK)	Total cost: SHA256 on 512bits + 1xDDR Access + Burst 4 Elements	**Mbytes/s**
1200	0.83	**100**	5	320	4	456	**160.6188322**

Figure 12: ARM Performance targets for 64Bytes SHA256 Hashing from DDR

The second scenario is the IPSEC acceleration fzor which SoC hardware accelerator are relatively bad at due to high number of 64Bytes packets (57%) contained in the IPSec flow. Assuming that the target of 5CLK/Byte is easy to achieve for a reasonable hardware cost, the performance benefit on IPSec types of protocols is expected to be important since they highly benefit from L2 caching on the contrary of dedicated hardware accelerators.

512bits Hashing (SHA256) from L2$ on a CortexA with 64Bytes Cache Line Size							
CPU Frequency (Mhz)	ns	**L2 Latency (ns)**	SHA256 Crypto ISA (CLK/Byte)	Cost for 64Bytes (CLK)	Latency between each 64Byte $line (L2$ CLK=MPU CLK)	Total cost: SHA256 on 512bits + 1xL2$ Access + Burst 4 Elements	**Mbytes/s**
1200	0.83	**10**	5	320	0	332	**220.6089985**

Figure 13: SHA256 Performance expectation for IPSEC on smallest packets

4.3 ARM cryptographic instruction set extension overview

For the reasons explained previously, ARM has added a cryptographic instruction extension to the ARM architecture for supporting NSA suite B 128Bits security level cryptographic operations. The current set of instructions that have been added support at least the following algorithms:

* AES128 CBC, CTR, GCM

  SHA256

AES symmetric encryption instructions usage overview

Four instructions are provided in the cryptographic extensions to implement the AES algorithm in CBC, CTR modes.

AES128 in CBC or CTR mode of operation:

* AES single round encryption AESE Vd.16B, Vn.16B

  AES mix columns. AESMC Vd.16B, Vn.16B
* AES single round decryption. AESD Vd.16B, Vn.16B
* AES inverse mix columns. AESIMC Vd.16B, Vn.16B

Two of these instructions allow implementing the AES encryption algorithm with the help of already existing instruction as explained hereafter:

SubBytes Multiplicative Inverse GF28 Affine transformation GF2

ShiftRows Byte-wise rotation of the state treated as rows.

MixColumns Each column treated as four-term polynomial GF28 Multiplied with a fixed polynomial, modulo x4+1

AddRoundKey XOR state with 128-bit of key

AESE Vd.16B, Vn.16B Q = Vd XOR Vn; Q = ShiftRows(Q) Vd = SubBytes(Q)

AESMC Vd.16B, Vn.16B Vd = MixColumns(Vn)

Two more instructions are provided in the cryptographic extension to implement the GHASH calculation of the AES GCM.

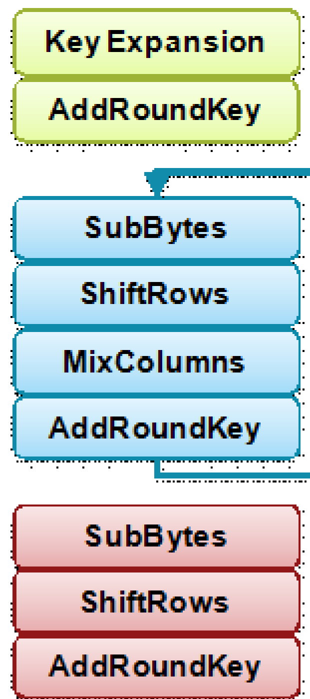


Figure 14: AES symmetric encryption instruction usage overview

AES128 GCM mode support: Polynomial multiply long (vector, first half): 64x64 to 128-bit. PMULL Vd.1Q, Vn.1D, Vm.1D

Polynomial multiply long (vector, second half): 64x64 to 128-bit. PMULL2 Vd.1Q, Vn.2D, Vm.2D

SHA256 secure hash instruction usage overview

SHA256 is an algorithm that iterate over 64-bytes of data and produces a 256-bit hash digest. This algorithm requires 64 rounds and 48 schedules operations for each 64 bytes of input data.

The ARM cryptographic extension implements a number of instructions to accelerate the SHA256 algorithm in such a way that several rounds and schedules operations can be performed at each execution of them. Therefore the principle of operation consists in looping a maximum of 48 times the round and schedule operation followed by a maximum of 16 more round operations to complete the hashing. The wk[] used in the first iteration is the result of w[]+k[] and is not processed by a schedule operation.

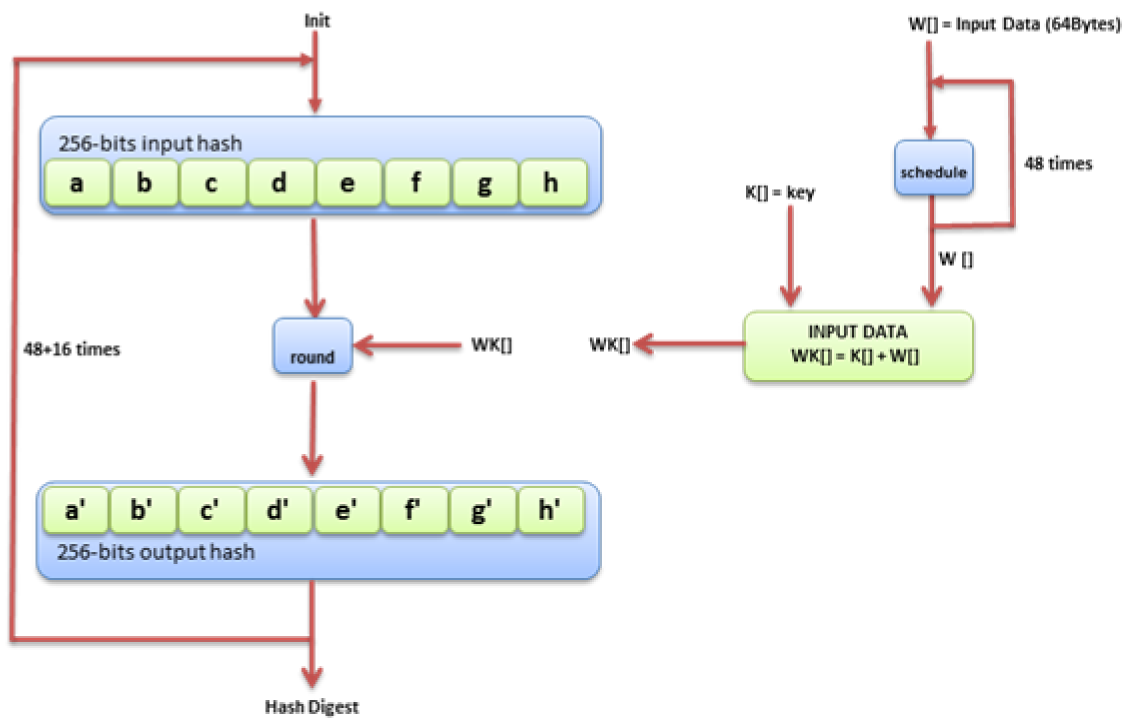


Figure 15: SHA256 round and schedule operations overview

5 Hardware Improvements for PETs on Security ICs

5.1 Introduction and Background

The SEPIA PET implementation is based on the DAA protocol. The results of Work Package 2 suggest that the DAA signing phase shall be performed for privacy reasons on the Secure Element (SE).

The signature computation has the following requirements:

* True Random Number Generator

  Efficient ECC point multiplication (ECC point multiplication is called 4 times)

* Efficient hash operation (optional)

Apart from the cryptographic requirements, there are additional requirements to a Security IC to be used as SE within the SEPIA reference platform (e.g. the supported interfaces).
The Infineon Security ICs which were available already at the beginning of the SEPIA project cannot meet all these requirements (e.g. to allow an effective implementation of DAA based on ECC). For example, the released Infineon Trusted Platform Module V1.2 (SLB 9635 TT 1.2) supports DAA as specified by the Trusted Computing Group (TCG). This DAA implementation is based on RSA (RSA-DAA). The RSA-DAA scheme is less efficient in both computation cost and communication cost than the DAA implementation based on ECC (ECC-DAA) which is researched in SEPIA.
Infineon is currently developing a new generation of Security ICs for Trusted Computing applications in embedded systems. Within SEPIA, Infineon develops the PET applications requirements for the SE, and how to optimize the new Security ICs especially for the usage within the SEPIA reference platform. The goal is to achieve a secure and effective implementation of the SEPIA PET application scenario.

5.2 Security IC Optimisation for the SEPIA Reference Platform

The following table compares selected SEPIA relevant functionalities of the current Security IC generation with the new Security IC generation. To compare the functionalities of concrete products, the Infineon TPM V1.2 (SLB 9635 TT 1.2) is selected as representative of the current generation of Security ICs. This chip is chosen, because it already implements privacy functionalities, which are based on the DAA protocol. The new generation of Infineon Security ICs is called SEPIA SE. This term will be used in the rest of the document. The SEPIA SE shall become the successor of SLB 9635 TT 1.2.

Security IC Functionality	**Released Security IC Generation Infineon TPM V1.2 (SLB 9635 TT 1.2)**	**New Security IC Generation SEPIA SE**	**Relevance for SEPIA**
Application Areas & Target Platforms	Trusted Computing on PC (notebooks, desktops and servers)	Trusted Computing on Embedded Systems (e.g. mobile phones) and PC	The SEPIA PET application requires a Security IC to meet advanced security and privacy requirements on mobile platforms.
Hardware Security Concept & Attack Resistance	1. Mainly sensor-based security 2. Some error-detection mechanisms (for selected memory areas).	Digital security concept: 1. Full error-detection 2. Total encryption 3. Signal protection 4. Sensor-based security is getting less prominent.	An improved hardware security concept and resistance against a wide range of attacks are required (in particular for application areas dealing with very valuable data)
Buses	The LPC (Low Pin Count) bus is used in common PC platforms to connect the TPM.	The SEPIA SE provides an I2C (Inter-Integrated Circuit) interface.Other busses are available too (e.g. SPI).The SEPIA SE internally makes use of an AMBA^{TM} (Advanced Microcontroller Bus Architecture) bus structure for optimal performance with low power consumption.	I2C is widely available in embedded systems like mobile phones, while LPC is not available. The internal AMBA^{TM} bus enables the flexible integration of the Security IC with other components (e.g. an ARM host processor).
Certification	Available: 1. CC EAL 4 moderate 2. TCG compliance certification	Planned: 1. CC EAL 4 moderate 2. TCG compliance certification Other Security ICs from the same generation are already certified according to CC EAL 5+ high.	Possible improvements to the certification subjects & process are developed in Work Package 3 of SEPIA (see [6]).
Cryptographic Functions concerning DAA	DAA based on RSA	DAA based on ECC (RSA still included for compatibility)	The SEPIA SE implements the new ECC-DAA for the PET application scenario. DAA using ECC significantly improves the energy-efficiency and performance of the next Security IC generation.

Table 11: Security IC functionality

Further details on the new Infineon Security IC generation will be provided in the Work Package 4 deliverables.

5.3 Security IC Optimisation for the SEPIA PET Application

This chapter focuses on the cryptographic improvements in the SEPIA SE for the SEPIA PET application. The SEPIA SE supports the requirements of the PET application with specific hardware and embedded software implementations.

Hardware

Figure 17 shows the SEPIA SE block diagram. The highlighted blocks show the SE components which are required and optimised for the SEPIA PET application.

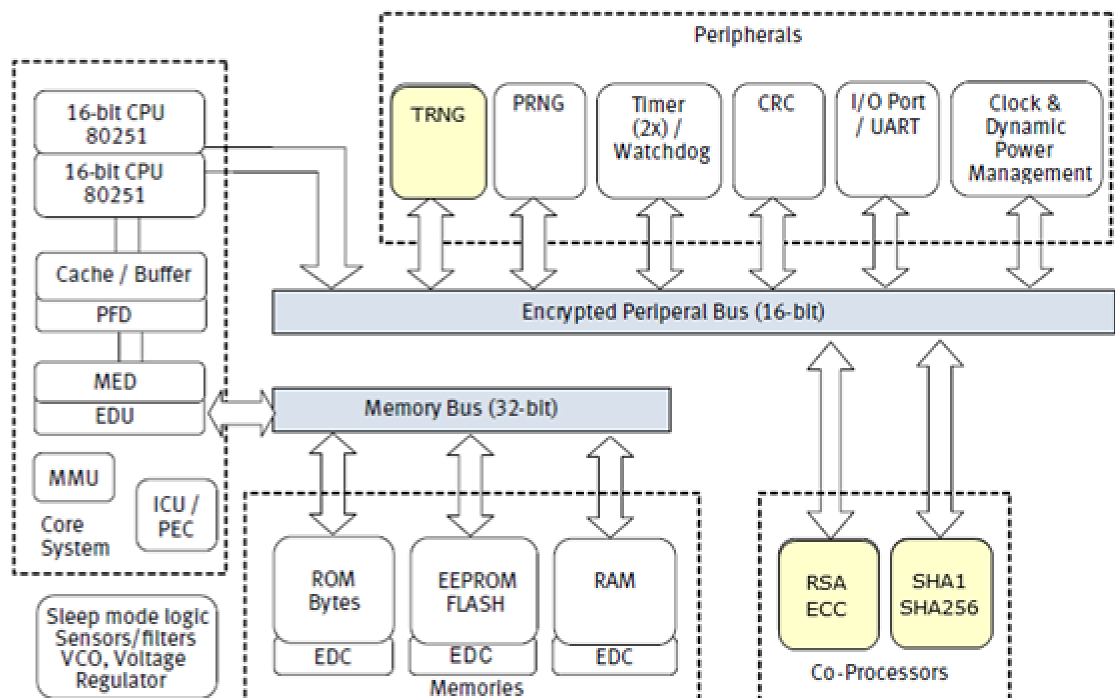


Figure 16: SEPIA SE Block Diagram

The cryptographic functionalities of the SEPIA SE are summarised in table 12:

Security IC Functionality	Released Security IC Generation Infineon TPM V1.2 (SLB 9635 TT 1.2)
True Random Number Generator	A True Random Number Generator (TRNG) is able to supply the CPU with true random numbers. Additionally, a Pseudo Random Number Generator (PRNG) is integrated for cases which require a fast generation of random numbers.
Efficient ECC Point Multiplication	The asymmetric Crypto Coprocessor supports ECC with key lengths up to 521 bit. Hardware based ECC support is the major enhancement in the new Infineon security IC generation, which was developed to fulfill the requirements of the SEPIA project.
Efficient Hash Operation	The hash accelerator supports the SHA1 and SHA256 hash algorithms in hardware.

Table 12: Required cryptographic functionality

Embedded Software

SEPIA SE Cryptographic Library

The SEPIA SE provides an embedded library with cryptographic functions. Figure 17 shows the blocks of the SEPIA SE Cryptographic Library.

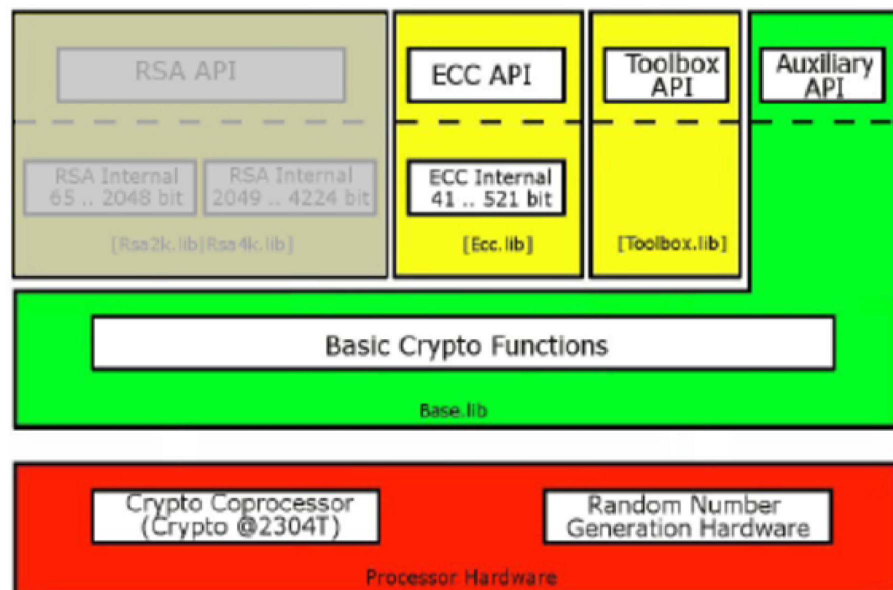


Figure 17: SEPIA SE Cryptographic Library

The asymmetric Crypto Coprocessor provides hardware-based asymmetric cryptography which can be accessed by applications via the following high-level cryptographic application interfaces:

* The ECC API provides cryptographic functions necessary for the ECC-DAA implementation of the SEPIA PET application.

  The RSA API can be used to implement RSA-DAA, which is still available to preserve the compatibility to legacy applications.

A high-level interface to perform long integer arithmetic and modular arithmetic functionality is provided by the **Toolbox API**. The toolbox library is functionally decoupled from the RSA and ECC libraries. However, it shares the same Base library. It contains the following functionalities:

* Arithmetic addition, subtraction, multiplication, and division

  Modular arithmetic addition, subtraction, multiplication, exponentiation, inversion and reduction

* Long integer comparison

The PET protocols and cryptographic algorithms were examined with respect to energy consumption and performance. These result showed that the access to the basic coprocessor functions is required to achieve optimal results.
Therefore the cryptographic functionalities required by PET protocols and algorithms were implemented on a low level of the Security IC instead of using a high-level API. To enable the realisation of these optimised implementations, access to these low-level functions was granted to partners of the project. The development of these low-level functions required an extensive and permanent support, as the development environment is very limited in Security ICs on this level. An auxiliary API was developed and contributed to the project to reduce the complexity and efforts for the development.

Outlook: SEPIA ECC-based PET Library
Currently the analysis of PET protocols and cryptographic algorithms is almost finished, and the development of the SEPIA PET application has been started. The next step is to develop a high-level API, which provides the PET-specific cryptographic functionalities of the SEPIA SE also to other PET applications. It is planned to realize this high-level API in a SEPIA PET library which is based on ECC. Figure 18 shows such a SEPIA ECC-based PET Library, which uses the ECC, the Toolbox and Auxiliary API to perform the required operations.

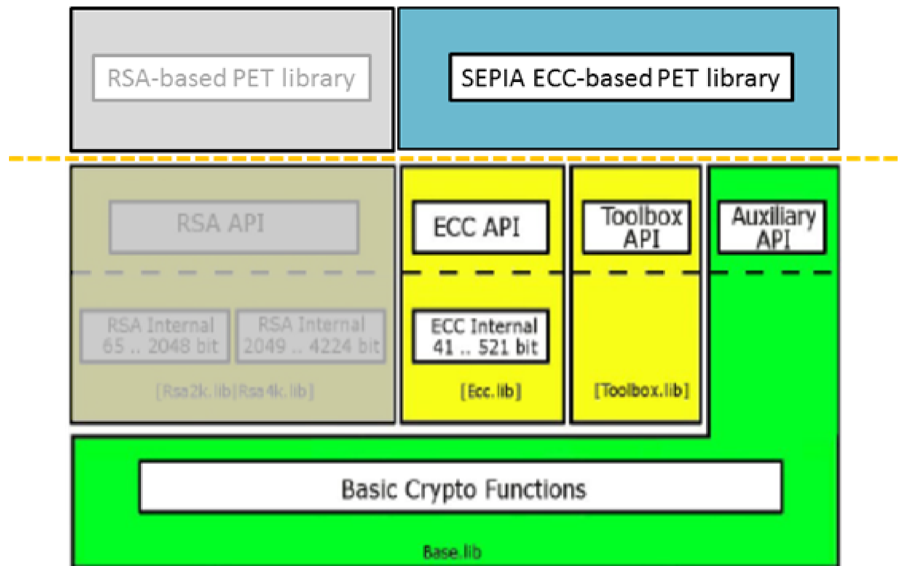


Figure 18: SEPIA ECC-based PET library

6 New developments of PETs for embedded systems

In this section, new anonymisation models and PETS based on modern cryptographic primitives are discussed.

6.1 A model for issuer less anonymisation

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, we present an anonymity preserving approach that allows trusted platforms to protect their anonymity without involvement of a trusted third party. We show how this new scheme can be used with existing Trusted Platform Modules version 1.2 and provide a detailed discussion of our proof-of-concept prototype implementation.

Background

One of the fundamental goals of Trusted Computing is to establish “trust-relationships” between computing platforms. These relationships are based on information about the integrity of the communication endpoints. This information can be provided by Trusted Platform Modules (TPMs) which form the core of every Trusted Platform and are shipped with millions of platforms (e.g. notebooks, PCs and servers). TPMs are capable of storing and reporting platform integrity information and performing cryptographic operations. In order to exchange the configuration settings and integrity information of a platform, the TCG introduced the concept of remote attestation.

However, 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.

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 what 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.

In order to address this problem, we turn our attention to *ring-signatures* which have been introduced by Rivest et al in the year 2001 [22]. This kind of signatures allows 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.

However, such signatures can become large, depending on the number of contributing public-keys. Efficient ring-signature schemes have been proposed in [11] and [9]. Unfortunately, these schemes cannot 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.

In this document, we provide a detailed discussion of how ring-signatures based

on the Schnorr signature algorithm [23] can be created using the TPM DAA commands. 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, we define a protocol that allows a TPM to obtain a credential from a TPM vendor which is further used in our approach.

Highlevel Description of Our Approach

In this section, we give a high level discussion of our approach and define the following assumptions and definitions:

* All TPMs are shipped with a unique RSA key-pair, the endorsement-key EK. Moreover, we assume 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.

  Both the signing platform H and the verifying platform V have to trust the TPM and the TPM vendor.
* An endorsement-key or *EK* denotes the endorsement key-pair (public and private part).
* An *endorsement-key-certificate* or *EK certificate* denotes a certificate that contains the public part of an endorsement key-pair.
* A *schnorr-key* or *SK* denotes a schnorr key-pair (public and private part).
* A *public-schnorr-key* or *public-SK* denotes the public part of a Schnorr-key-pair.
* *schnorr-key-certificate* or *SK-certificate* denotes a certificate that contains the public part of a Schnorr key-pair.

We take 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.

Our 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, we have 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 [22] and [2] which we use as a basis for our 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 19.

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 our 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 our approach is discussed in Section 6.2.

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 6.2.
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 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 (RFC3280). The subject field is left empty. For our experiments, we created SK-test-certificates with according entries. 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. Our 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 [22].

6.2 Schnorr Signature based Approach

In this section, we discuss our Schnorr signature based approach which is based on a publication from Abe et al, who proposed to construct ring-signatures based on Schnorr signatures [2] in order to reduce the size of the overall signature. In contrast to the approach from Rivest [22], 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 our approach with a few modifications of the sign and verify protocol. A major advantage of this approach is that we can use existing TPM 1.2 functionality to compute this kind of signatures. In order to do so, we can exploit the DAA *Sign* and *Join* protocol implementation of the TPMs v1.2.

Signature Generation. Let n be the number of public-SKs contributing to the ring-signature and H a hash function $H_i : \{0,1\}^* \Rightarrow \mathbb{Z}_n$. j is the index of the signer's public-key SK_j consisting of y_i, the modulus N_i and g_i with $N_i = p_i q_i$ and p_i, q_i 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 our approach, we want to involve the TPM of the signer 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, we can use the TPM_DAA_Sign and TPM_DAA_Join commands to compute Schnorr signatures for our purpose. Therefore, we extend the algorithm description with the stages that have to be gone through during the execution of the TPM commands:

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, we use a common modulus N and a fixed base g for all contributing platform's in our further discussions. M is 20 byte long nonce required for computing a DAA signature inside a TPM.

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 13 for the DAA Sign command steps).
3. Compute a random M_{T_i} and $c_{j+1} = H_{j+1}(H(H(g||n||y_0||..||y_{n-1}||T))||M_{T_i})||1 or 0||m \text{ or } AIK)$
4. For $i = j+1..n-1$ and $0..j-1$.

   Compute a random M_{T_i}, s_i.

   Compute $c_{i+1} = H_{i+1}(H(H(g||n||y_0||..||y_{n-1}||e_i))||M_{T_i})||1 or 0||m \text{ or AIK})$ 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 $\sigma = (c_0, s_0, .., s_{n-1}, M_{T_0}, ..., M_{T_{n-1}})$

Figure 19: Schnorr Ring-Signature creation

Computing the Schnorr Ring-Signature. In order to compute a Schnorr signature, we can exploit the TPM_DAA_Sign command. Therefore, we start the protocol and execute stages 0 to 11 as defined in [33], however, for our purpose, only stages 2 to 5 and 9 to 11 are of interest.

Table 13 shows the steps for running the DAA Sign protocol with a TPM. The TPM_DAA_Sign command is executed in 16 stages by sub-sequent execution of

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	$P_3 = P_2 * S_0^{r_{v_1}} \mod N$	-
5	$S_1 = 1$	n	$T = P_3 * S_1^{r_{v_2}} \mod N$	T
.	.	.	.	.
.	.	.	.	.
9	c_{in}	-	$c' = H(c_1 \| \| M_T)$	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 13: TPM_DAA_Sign Command Sequence

the command.

It is not required to finish the *Sign* protocol and we can terminate the DAA session 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, we had to modify the Schnorr signature generation and verification scheme: 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. We do not require this proof and set $c_{in} = H(g||N||y_0||..||y_{n-1}||e)$ (with $e = g^{r_0} y_j^{c_{j-1}} \mod N$) in our scheme. However, the resulting value M_{T_j} has to be recorded as we require it to verify the signature. As a result, the TPM computes $c_j = H((H(c_{in})||M_{T_j})||1or0||morAIK)$.

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 [33]. 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 6.2: The verification of the signature does

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

Figure 20: Schnorr Ring-Signature verification

not involve a TPM.

Parameter Setup. Before executing the Join protocol, we have to generate the DAA parameters i.e. issuer public-key, issuer long-term public-key [33] which we require during the execution of the protocol to load our signature settings into the TPM. In order to compute the platform's public and private Schnorr key, we first have to commit to a value f_0 by computing $y = g_0^f \mod N$. 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	$P_3 = P_2 * S_0^{s_{v_0}} \mod N$	-
7	$S_1 = 1$	n	$y = P_3 * S_1^{s_{v_1}} \mod N$	y
.	.	.	.	.
24	-	-	E=enc(DAA_param)	E

Table 14: TPM_DAA_Join Command Sequence

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

The DAA commands (as shown in Table 14) are executed in 25 stages by subsequently 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 we can obtain the public-key $y = g^{f_0} \mod N$. 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 7.

Security Parameter Sizes. We suggest the following 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.

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, we 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, we have 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 14)
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.

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. We used a Java implementation for our tests, hence, optimized C implementations (e.g. based on OpenSSL [31]) 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 7.

For the sake of completeness, we provide the performance values of our test TPMs, demonstrating the time required for a full DAA-Join command and the stages 0-11 of the DAA-sign command (see Table 15).

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 15: TPM_DAA_Sign Command Measured Timings

For the DAA *Sign* operation, we are only interested in the stages 0-11 (see Figure 19), hence we can abort the computation 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 [15]. Details of the evaluation environment can be found in Section 7.

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 [5].

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 $U = R_0^{f_0} R_1^{f_1} S_0^{v_0} S_1^{v_1} \mod n$ 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 13 we compute the signature (c, s) on the message m. 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 EK_S 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_S and the assumption that only valid EK_Ss may contribute to a signature is violated.

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

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.

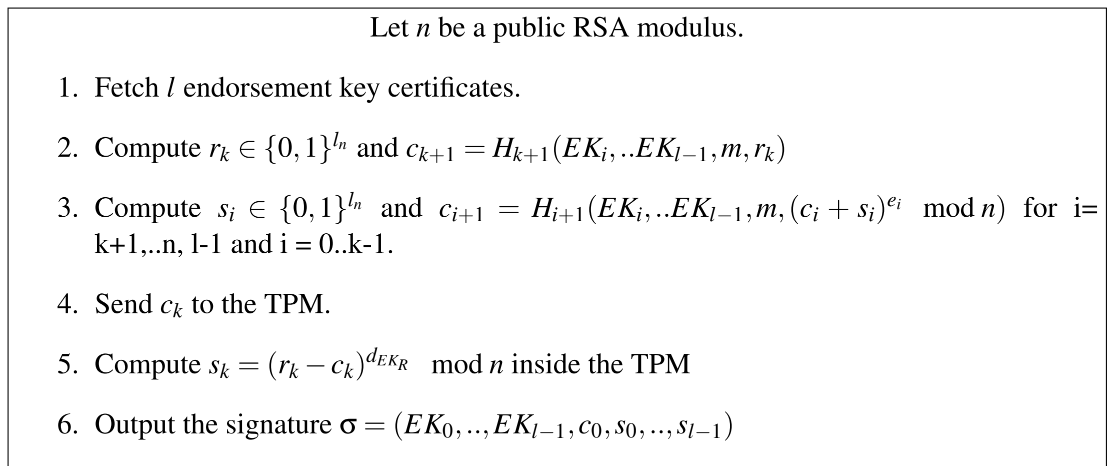
[5] http://www.trustedcomputinggroup.org/media_room/news/95

6.3 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 [6].

A drawback of this approach is, however, that we require to compute the reverse operation of a signature with the EK inside the TPM in order to form the ring. Existing TPMs do not support this feature the way we need it. The TPM can decrypt data that was previously encrypted with the public EK. This operation is based on the RSA-OAEP [16] encryption scheme which requires an OAEP padding of the encrypted data. However, we require a decryption operation 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:



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 $\sigma = (EK_0, .., EK_{l-1}, c_0, s_0, .., s_{l-1})$

Figure 21: RSA ring-signature creation

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

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

before encryption [16]. It is not possible to perform a *raw* decryption operation which we would require to close the ring.

In order to overcome this problem, we propose an additional TPM command which we added, for testing purposes, to the TPM emulator [28]. The TPM_Ring_Close command allows us to apply this decryption operation and to form the ring.

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:

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.

A single signature operation with this approach takes 8 ms. We can give no real-world performance values for the TPM operation as we used the TPM emulator. However, if we assume 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.

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

6.4 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 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 TPM's is vendor dependent (see Table 15) 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.

7 Implementation Notes

In order to obtain experimental results, we implemented a Java library 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 [3] 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 [13].
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).

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 [7] and the security parameters given at the end of Section 6.2 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

[7] This is the size of a typical ASN.1 [12] encoded EK-certificate from Infineon which we used as a template.

of this optimization is that 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.

Conclusion

In this paper, we have proposed an anonymization scheme for trusted platforms that does not rely on specialized trusted third parties. Our 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.
We have shown that our scheme 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.

8 References

[1] The gnu multiple precision arithmetic library. http://gmplib.org/.

[2] Masayuki Abe, Miyako Ohkubo, and Koutarou 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.

[3] Scott A. Vanstone 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.

[4] Patrik Bichsel, Jan Camenisch, Thomas Groß, and Victor 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.

[5] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, pages 41–55, 2004.

[6] Brightsight. Sepia wp3, d3.1, stakeholder requirements for platform security. Technical report, Nov 2011.

[7] F. Baumgartner C. Latze, U. Ultes-Nitsche. Extensible Authentication Protocol Method for Trusted Computing Groups (TCG) Trusted Platform Modules. Internet-Draft, July 2009.

[8] Angelo De Caro. Java pairing based cryptography library. http://libeccio.dia.unisa.it/projects/jpbc/.

[9] Nishanth Ch, Jens Groth, and Amit Sahai. Ring signatures of sub-linear size without random oracles. In *In ICALP07, LNCS*. Springer, 2007.

[10] Kurt Dietrich. Anonymous credentials for java enabled platforms. In Liqun Chen and Moti Yung, editors, *INTRUST 2009*, pages p. 101 – 116, 2009.

[11] Yevgeniy Dodis, Aggelos Kiayias, Antonio Nicolosi, and Victor Shoup. Anonymous identification in ad hoc groups. In *IN EUROCRYPT 2004, VOLUME 3027 OF LNCS*, pages 609–626. Springer-Verlag, 2004.

[12] Olivier Dubuisson and Philippe Fouquart. *ASN.1: communication between heterogeneous systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[13] IBM. *TrouSerS The opensource TCG Software Stack*, 2 November 2007.

[14] Intel. *Intel Desktop Board DQ965GF Technical Product Specification*. Specification available at: `downloadmirror.intel.com/15033/eng/DQ965GF_TechProdSpec.pdf`, September 2006.

[15] Intel. *Intel Desktop Board DQ45CB Technical Product Specification*. Specification available at: `http://downloadmirror.intel.com/16958/eng/DQ45CB_TechProdSpec.pdf`, September 2008.

[16] RSA Labs. *PKCS1 v2.1: RSA Cryptography Standard*, 2001.

[17] Ben Lynn. The pairing-based cryptography library. http://crypto.stanford.edu/pbc/.

[18] Ben Lynn. *On the implementation of pairing-based cryptosystems*. PhD thesis, 2007.

[19] Ben Lynn, Hovav Shacham, Joe Cooley, and David Jao. Pairing-based signature schemes. http://crypto.stanford.edu/pbc/sig/.

[20] Chris Mitchell. *Direct Anonymous Attestation in Context*. In *Trusted Computing (Professional Applications of Computing)*, pages p. 143–174, Piscataway, NJ, USA, 2005. IEEE Press.

[21] SUN Community process. *Java Specification Request (JSR-257): Contactless Communication API*. Specification available at: `http://jcp.org/en/jsr/detail?id=257`, October 2004.

[22] Ronald L. Rivest, Adi Shamir, and Yael 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.

[23] Claus 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.

[24] Michael Scott. Miracl library. http://www.shamus.ie/, 2011.

[25] Michael Sterckx, Benedikt Gierlichs, Bart Preneel, and Ingrid 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.

[26] 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.

[27] 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.

[28] Mario Strasser. *TPM Emulator*. Software package available at: `http://tpm-emulator.berlios.de/`.

[29] 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`.

[30] SUN Microsystems. *JSR 177: Securtiy and Trust Services API*. Specification available at: `http://java.sun.com/products/satsa/`.

[31] The OpenSSL Project. OpenSSL. Programa de computador, December 1998.

[32] 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.

[33] Trusted Computing Group - TPM Working Group. *TPM Main Part 3 Commands*. Specification available online at: `http://www.trustedcomputinggroup.org/files/static_page_files/ACD28F6C-1D09-3519-AD210DC2597F1E4C/mainP3Commandsrev103.pdf`, 26 October 2006. Specification version 1.2 Level 2 Revision 103.

[34] 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.

[35] David E. Williams and Juan R. Garcia. *Virtualization with Xen: including XenEnterprise, XenServer, and XenExpress*. Syngress, Burlington, MA, c2007. Includes index.