Anonymous Client Authentication for Transport Layer Security

Kurt Dietrich

Institute for Applied Information Processing and Communications
University of Technology Graz, Inffeldgasse 16a, 8010 Graz, Austria
{Kurt.Dietrich}@iaik.tugraz.at

Abstract. Nowadays, 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, standard protocols like Transport Layer Security are still missing adequate support for these technologies. In this paper, we analyze how Trusted Computing technologies and anonymous credential systems can be used in order to allow clients to establish anonymous authentication over secure channels. Furthermore, we analyze how these technologies can be integrated into common security frameworks like the Java Cryptography Architecture. We discuss the performance that can be achieved with this approach and analyse which performance can be expected from currently available Trusted Platform Modules.

Keywords: Trusted Computing, Transport Layer Security, TLS, Direct Anonymous Attestation, Java, DAA.

1   Introduction

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 Trusted Computing Group (TCG): the *PrivacyCA* (PCA) scheme and the Direct Anonymous Attestation (DAA) 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 compromission 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 computations, making it hard to use on resource constrained devices. Moreover, the scheme requires a tamper resistant storage device (i.e. the TPM) 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 2.2. The support for both schemes is an integral part of all TPMs since version 1.2. Furthermore, as TPMs are available on many desktop and notebook platforms, the question arises whether these schemes can be applied for applications and technologies other than trusted computing related ones which also have a high demand for privacy protection. One such technology is the Transport Layer Security protocol (TLS) [1] which is used in many systems to establish secure and authenticated connections.

1.1   Related Work

Several ideas for integrating TPMs or Trusted Computing technology in TLS have been published. Latze et. al. propose to use the TPM for identity distribution, authentication and session key distribution and have defined an Extensible

Authentication Protocol (EAP) extension in order to integrate the Trusted Computing and TPM related information [2]. Although the protocol supports anonymous authentication, it is based on the PCA scheme and not on DAA. Moreover, this document is currently work-in-progress and is tagged to be in an "experimental" status.

The approach from Cesena et. al [3] aims at providing anonymous authentication for trusted platforms and trusted applications in the sense of Trusted Computing. In their work, they define a set of extensions for TLS for transporting DAA related information and propose a design for integrating it into OpenSSL. They use the DAA scheme as defined in the Trusted Software Stack specification [4] which requires them to use a full blown Trusted Software Stack (TSS) [5]. They also provide an ECC based variant of the DAA scheme which is not supported by existing TPMs. Another interesting publication that is not directly related to this work, but may have interesting implications on further applications, is discussed in [6]. Bichsel et. al. propose an implementation of DAA on a JavaCard. Using this approach, it might be possible to use the anonymous authentication mechanism in combination with smart cards instead of TPMs. This approach could increase the flexibility and area of application of the anonymous TLS authentication as the anonymous credentials can be bound to a specific user instead to a specific TPM and platform.

1.2   Our Contributions

In this article, we focus on the DAA scheme for providing anonymity in secure channels. Our approach does not focus on promoting Trusted Computing and its applications, it rather makes use of Trusted Computing technologies (i.e. the TPM and DAA support) which are available on many PC platforms and aims at using these technologies for arbitrary applications. Instead of using a full-blown TSS which is typically used for trusted applications to use the TPM features, we focus on a minimal set of functions that are required for communicating with the TPM and employ its DAA features. This is also true for the changes required in the TLS protocol. This approach allows us to use it on embedded systems and mobile phones which are going to be equipped with TPMs in the near future [7]. Different approaches for employing TPM functionality on mobile and embedded devices is discussed in [8], [9], [10] and [11]. In order to demonstrate our achievements, we developed a library with focus on portability, size efficiency and simple usability. This minimal DAA library is based on the DAA scheme which we will address as BCC05 [12], 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 which is a performance optimized scheme that requires less parameters and less computations than the BCC04 scheme, however, both can use the DAA features from TPMs without any modifications of the TPM. A discussion about the differences between BCC04 and BCC05 and the security of the BCC05 scheme can be found in [12]. 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 rest of this article is organized as follows: we give background information on TLS client authentication and the DAA scheme. We discuss our implementation and its integration in the JCA framework and examine the achieved performance values. In one paragraph, some comments on compatibility of TPMs from different vendors are given. Finally, we conclude the article with a summary and suggestions for future improvements.

2   Background

2.1   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. The messages marked with
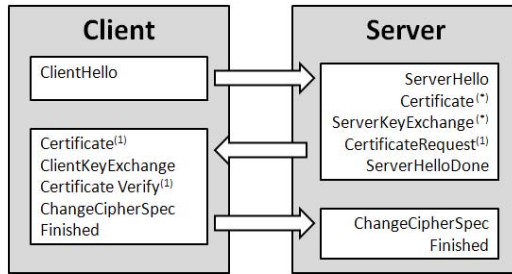

Fig. 1. The TLS Handshake Protocol

(1) are required for client authentication which works as follows: The server sends the client a list with certificate authorities (CAs), which it accepts. The client selects a CA and returns a $Certificate$ message that contains the client's certificate, which then certifies its authentication key. The $CertificateVerify$ message contains a signature on the hash of the handshake messages sent so far. By verifying the signature - the hash can be computed by the server as it knows all messages that have been exchanged with the client - and by verifying the client's certificate plus the corresponding certificate chain, the server can validate the client's authentication information [1].

2.2   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 f_0, f_1, ν' (the key f is separated in two parts for performance reasons) 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 4.2.

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)[1] 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 to outsource 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.

---

[1] The endorsement key pair is a RSA key pair that is unique to every TPM. It is not modifiable and can only be used for decryption operations inside the TPM.

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) [4]. 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.

3   The Authentication Scheme

In contrast to [3], we do not use TLS extensions to transport DAA specific information. In order to keep the implementation for our demonstrator small and simple, we do not implement the 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 2, 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 in Section 1.

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. At the moment, TPMs only support the RSA signature scheme but this will change in future versions. Current efforts are aiming at equipping TPMs with elliptic curve cryptography. With these new algorithms, TPMs will be able to create ECC signatures, allowing this authentication procedure to support ECC.

4   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 [13]. 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. Details of our implementation of the cryptographic functions and the BCC05 scheme can be found in [14].

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 [4] specification and makes it accessible to developers that are not familiar with Trusted Computing.

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

4.1   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

Table 1. Performance of the Join Protocol with Intel TPMs

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

system with a 2.6.30-1 Kernel (64 bit). A verification of a DAA signature takes about half the time required for signature creation and does not require a TPM. All results presented in this Section represent the average measurement values of 100 executions of the *Join* and the *Sign* protocol. We have also performed tests

Table 2. Performance of DAA signature creation with Intel TPMs

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

with TPMs from ST Micro and Intel. The results 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 [15], the Intel TPM is located in the Intel mother-board chipset (i.e. the Intel 82801JDO Controller Hub (ICH10DO)) itself [16].

Table 3. Comparison of the DAA Performance of different TPMs

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

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 [13].

4.2   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
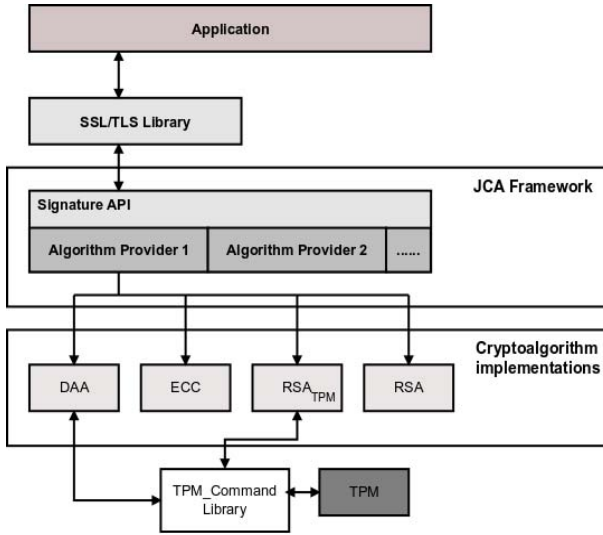
Fig. 2. Integration of the DAA library in the JCA Architecture

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) [17]. 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 [18] 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 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 [4] 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: $\tilde{T}_t = R_0^{r_{f_0}}R_1^{r_{f_1}}S_0^{r_{\nu_0}}S_1^{r_{\nu_1}} \mod n$ with r_{f_0}, r_{f_1} of size $l_f + l_\phi + l_H$ bits and $r_{\nu_{(0,1)}}$ with length $l_n + l_\phi + l_H$. $\tilde{N}_V = \zeta^{\tilde{r}_f} \mod \Gamma$
5. $\tilde{T}_t$ and $\tilde{N}_V$ are returned to the host which computes: $\tilde{T} = \tilde{T}_t T^{r_e}S_0^{r_{\bar{\nu}_0}}S_1^{r_{\bar{\nu}_1}} \mod n$ 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 $s_{\bar{\nu}_0} = s_{\nu_0} + r_{\bar{\nu}_0} - cw_0 e$, $s_{\bar{\nu}_1} = s_{\nu_1} + r_{\bar{\nu}_1} - cw_1 e$
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 $S_1 = S^{2^t}$ 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.

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 [14].

```
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

4.3   A Note on Specification Compliance

In the TPM specification [13], 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 [19]. 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.

5   Conclusion and Future Work

In this paper, we elaborate on different topics: we discuss how a anonymous credential systems based on TPMs can be integerated 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.

References

1. Dierks, T., Rescorla, E.: R.I: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard) (August 2008)
2. Latze, C., Ultes-Nitsche, U., Baumgartner, F.: Extensible Authentication Protocol Method for Trusted Computing Groups (TCG) Trusted Platform Modules. Internet-Draft (July 2009)
3. Cesena, E., Ramunno, G., Vernizzi, D.: D03c.3 ssl/tls daa-enhancement specification. Technical report, Politecnico Di Torino (May 2009)
4. Trusted-Computing-Group-TSS-Working-Group: TCG Software Stack (TSS) Specification Version 1.2 Level 1. Part1: Commands and Structures (January 2006), Specification available online at, `https://www.trustedcomputinggroup.org/specs/TSS/TSS_Version_1.2_Level_1_FINAL.pdf`
5. IBM: TrouSerS The opensource TCG Software Stack (November 2, 2007)
6. Bichsel, P., Camenisch, J., Groß, T., Shoup, V.: Anonymous credentials on a standard java card. In: CCS '09: Proceedings of the 16th ACM conference on Computer and communications security, pp. 600–610. ACM, New York (2009)
7. 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` (June 12, 2007)
8. Ekberg, J.E., Bugiel, S.: Trust in a small package: minimized mrtm software implementation for mobile secure environments. In: STC '09: Proceedings of the 2009 ACM workshop on Scalable trusted computing, pp. 9–18. ACM, New York (2009)
9. Winter, J.: Trusted computing building blocks for embedded linux-based arm trustzone platforms. In: STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing, pp. 21–30. ACM, New York (2008)
10. Dietrich, K.: An Integrated Architecture for Trusted Computing for Java Enabled Embedded Devices. In: STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing, pp. 2–6. ACM Press, New York (2007)
11. Dietrich, K., Winter, J.: Implementation aspects of mobile and embedded trusted computing. In: Chen, L., Mitchell, C.J., Martin, A. (eds.) Trust 2009. LNCS, vol. 5471, pp. 29–44. Springer, Heidelberg (2009)
12. Mitchell, C.: Direct Anonymous Attestation in Context. In: Trusted Computing (Professional Applications of Computing), Piscataway, NJ, USA, pp. 143–174. IEEE Press, Los Alamitos (2005)
13. Trusted Computing Group - TPM Working Group: TPM Main Part 3 Commands, Specification available online at, `https://www.trustedcomputinggroup.org/specs/TPM/mainP3Commandsrev103.zip` (July 9, 2007) Specification version 1.2 Level 2 Revision 103
14. Dietrich, K.: Anonymous credentials for java enabled platforms. In: Chen, L., Yung, M. (eds.) INTRUST 2009, pp. 101–116 (2009)
15. Intel: Intel Desktop Board DQ965GF Technical Product Specification, Specification available at, `downloadmirror.intel.com/15033/eng/DQ965GF_TechProdSpec.pdf` (September 2006)

16. Intel: Intel Desktop Board DQ45CB Technical Product Specification, Specification available at, `http://downloadmirror.intel.com/16958/eng/DQ45CB_TechProdSpec.pdf` (September 2008)
17. 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`
18. 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`
19. Williams, D.E., Garcia, J.R.: Virtualization with Xen: including XenEnterprise, XenServer, and XenExpress. Syngress, Burlington, MA (c2007) Includes index