EFFICIENT IMPLEMENTATION OF ANONYMOUS CREDENTIALS ON JAVA CARD SMART CARDS

Michaël Sterckx, Benedikt Gierlichs, Bart Preneel, Ingrid Verbauwhede

K.U. Leuven, ESAT/SCD-COSIC and IBBT
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
firstname.lastname@esat.kuleuven.be

ABSTRACT

The Direct Anonymous Attestation scheme allows to map procedures with an imperative requirement for anonymity, such as voting, to the electronic world while offering provable security. However, the scheme is complex and requires demanding computations to be performed on a tamper-proof device. Such devices, e.g., secure smart cards, are typically resource constrained. We present the first implementation of the (simplified) Direct Anonymous Attestation protocols suitable for contemporary Java Card smart cards. We point out performance bottlenecks and provide efficient solutions which allow our implementation to terminate within acceptable time.

Index Terms—SEC-PRIV, SYS-SOFT, SEC-INTE

1. INTRODUCTION

The security of modern information systems is becoming more and more dependent on cryptography. An ever increasing number of online transactions is being executed every day, involving potentially sensitive data. People are often not even aware of the amounts of information that are available about them freely in the online world. However, it is often not desirable that personal information is spread all over the world wide web for everyone to see.

The problem is twofold. To start with, it can be undesirable to share personal data with all kinds of organisations. But even if we trust a particular organisation not to abuse our data, we might not be able to trust it to secure this data properly.

A feature that is very desirable in the context of online transactions is anonymity. A problem is that many transactions still require authentication, which led to anonymous authentication protocols. A good example of these requirements is an e-voting system: a user must be authenticated prior to voting but of course wants his vote to be anonymous.

The purpose of an anonymous credential scheme is to allow users to authenticate and/or prove certain attributes to a verifier while preserving their anonymity. This means that the verifier must not be able to determine the identity of a user based on a single or multiple authentication transactions. Further, different authentications of the same user must be unlinkable to each other, e.g. to avoid user profiling.

A credential establishes certain attributes of some party, e.g. the identity, in a communication scheme and is typically issued to a user by a trusted third party. A user can use a credential to prove these attributes to a verifier, e.g. to authenticate. An anonymous authentication mechanism also has to avoid the misuse of credentials. For example, it must be impossible for users to authenticate with credentials that were issued to someone else.

The Direct Anonymous Attestation (DAA) scheme [1] is an anonymous authentication scheme that allows to remotely authenticate a hardware module. The scheme uses a distributed computation mechanism where computations on the user side are split between a trusted platform module (TPM) and a host.

We implemented the simplified DAA scheme [2, 3] and opted for a Java Card smart card as TPM. These devices run a simplified Java virtual machine. Advantages of implementing the scheme using the Java Card standard [4, 5] include code portability and multi-application support. The implementation can for example be installed on a Java Card-enabled TPM as they are widely deployed in laptops. However, due to the inherently inefficient execution of Java Card code an implementation of mathematically complex cryptographic protocols like the DAA scheme in Java code only would be far too slow for practical use. In this work we therefore identify performance bottlenecks and present workarounds that allow to obtain a reasonably fast implementation by clever usage of the functionality offered by Java Card platforms. We show how the cryptographic co-processor and library available on Java Card smart cards can help in speeding up the implementation of operations not directly offered by a Java Card application programming interface (API). An implementation of the (full) DAA scheme can directly benefit from our implementation tricks which should yield a similar performance improvement.

Section 2 of this paper summarises the purpose and high-level working principles of the (simplified) DAA scheme. In Section 3 we present all conversions necessary for the efficient implementation of the DAA protocols on Java Card platforms. Section 4 provides timing and memory details and Section 5 presents our conclusions.

2. THE (SIMPLIFIED) DAA SCHEME

The DAA protocols as presented in [1] are an anonymous authentication scheme providing a means for the remote authentication of a hardware module. This hardware module is a TPM that is integrated into a host. The TPM and host together form the “user” entity which authenticates towards a verifier. The TPM is some kind of tamper-resistant device, like a secure microcontroller or (in our case) a secure smart card.

The reasons for splitting up the user entity into host and TPM are closely linked to the properties of a typical TPM:

- The tamper resistance of the TPM helps to protect the secret keys of the user in a more reliable way, thereby providing better anonymity assurance. In contrast to a general purpose platform, which can have many vulnerabilities and complicated security issues, extracting secret keys from a
2.3 Simplified DAA Protocols

Whereas Sucasas [2] considers the host side. An extensive description of the simplifications made to the TPM.

In the following explanation we will mainly treat the TPM side of the protocols, because it is more extendable and easier to build a certificate (anonymous credential).

The scheme is the first standardised privacy enhancing protocol to de-identify any identity information, and his anonymity stays unrevocable. While doing this, the schemes and features involving pseudonyms are removed. This means that the issuer looses its ability to apply a credential granting policy, or to detect rogue TPMs. Further, some of the parameter sizes proposed in the original paper are reduced. Most of the simplifications were made to allow for a more extendable program and an easier implementation [2]. For more details on the original and the simplified scheme we refer to [1] and [2, 3], respectively.

3. IMPLEMENTATION

The vital requirement for a practical implementation of the (simplified) DAA scheme is an acceptable execution time. This is opposed to a Java Card smart card’s limited resources, in particular computational performance. The DAA protocols require complex computations on very large numbers. Java Card code, however, executes on a virtual machine which introduces a huge overhead. Further, there is no built-in support for large number arithmetic and efficient low-level programming is not possible. Our challenge is thus to outsource as many operations as possible to the efficiently implemented Java Card API.

For our implementation we need several complex (cryptographic) operations: secure random number generation and cryptographic hashing (SHA-1) are directly available through the cryptographic API. Large number addition and subtraction are not very complex and we implement them using standard paper-and-pencil algorithms. Extension to modular operations is then straightforward by comparing and adding, respectively subtracting the modulus once. What remains as the main challenge is modular multi-exponentiation (with 1024-bit prime or RSA moduli and exponents of length up to 1752 bits).

3.1. Modular multi-exponentiation

The modular exponentiation of base \( b \) to the power \( e \) modulo \( n \) is represented as \( b^e \mod n \). The modular multi-exponentiation of bases \( b_1 \) and \( b_2 \) to the powers \( e_1 \) and \( e_2 \) modulo \( n \) is represented as \( b_1^{e_1} \cdot b_2^{e_2} \mod n \).

When computing a multi-exponentiation, it is in general inefficient to compute the powers \( b_1^{e_1} \mod n \) separately and to multiply the results afterwards. Instead, efficient algorithms for multi-exponentiation typically combine both steps. For example, some algorithms first combine the bases \( b_i \) during a precomputation stage and then process all exponents simultaneously. For more information on this topic we refer to [7].

Although these algorithms can compute a multi-exponentiation efficiently, their implementation in Java would still be far too slow. Balasch already showed in [2] that even a highly optimised assembly implementation of a multi-exponentiation algorithm (due to Dimitrov, Jullien and Miller) using hybrid multiplication and Barrett reduction with folding does not execute in reasonable time on an 8-bit microcontroller (the multi-exponentiation takes more than two minutes). Thus we are better of not even considering to implement such algorithms in Java.
The Java Card API does not provide any support for (multi-)exponentiation, but it does include support for RSA [8] operations which are most likely accelerated by a cryptographic co-processor. In this section we explain how we can use the RSA functionality to efficiently compute single modular exponentiations, and Sect. 3.2 describes how to efficiently combine these single modular exponentiations to a multi-exponentiation result by multiplication.

An RSA encryption of the message $m$ to the ciphertext $c$ is executed by calculating $c = m^e \mod n$. It is clear that calculating a single modular exponentiation is therefore exactly the same as an encryption with RSA, provided it is possible to set all the values involved. The Java Card cryptographic API indeed allows to set the exponent (public-key) and the modulus while the RSA_NOPAD mode ensures that the base (message) remains unchanged.

### 3.1.1. Extension for Long Exponents

Some of the exponents in the DAA protocols have a size longer than the modulus. Since it is not possible to feed this arbitrary-length $e$ directly to the cryptographic API we split the exponent as follows:

$$ e = e_1 || e_2 || \ldots || e_L $$

where each sub-exponent is at most of size RSA_length (defined by the particular RSA function used). We can now write the exponentiation $b^e \mod n$ as $b^{e_1} || b^{e_2} || \ldots || b^{e_L} \mod n$. For each of the sub-exponents, a modular exponentiation is executed to produce the intermediate results $b^{e_i} \mod n$.

In the following we explain only the case where the exponent $e$ fits in two sub-exponents $e_1$ and $e_2$ to keep the equations simple. We thus write the exponentiation as:

$$ b^e \mod n = b^{e_1} \cdot b^{2^{\text{RSA_length}}} \cdot b^{e_2} \mod n. $$

The first exponentiation in Eq. 2 still has an exponent that is too large to feed to the cryptographic API directly. Splitting is however straightforward:

$$ b^{e_1} \mod n = b^{e_1} \cdot b^{2^{\text{RSA_length}}} \mod n. $$

All exponents involved now are at most of size RSA_length, such that all the exponentiations can be calculated using the RSA co-processor. As a result of all the transformations described above, we converted one modular exponentiation involving an exponent with size larger than RSA_length into several modular exponentiations and multiplications. For example, when it is sufficient to decompose $e$ into two sub-exponents, three modular exponentiations and one modular multiplication have to be calculated.

The remaining problem is thus to implement an efficient modular multiplication as this operation is not supported by the Java Card API. We need this operation to compute single modular exponentiations with long exponents according to Eq. 2 and to combine the results of single exponentiations in the multi-exponentiation. In the next section we discuss our implementations of (modular) large number multiplication.

### 3.2. Large number multiplication

A standard large number multiplication algorithm is possible based on the classical pencil-and-paper method for row multiplication. Multiplication is done word by word, creating a carry word each time.

#### 3.2.1. Modular multiplication

Since the product of two equally sized large numbers can be double this size, a straightforward extension of large number multiplication to modular multiplication, as we implemented it for addition and subtraction, is not efficient. Several standard solutions exist though:

Special algorithms for modular multiplication such as Montgomery multiplication [9] become advantageous only if many modular multiplications have to be computed. For example, Montgomery multiplication works by first converting the factors to the Montgomery domain, multiplying and reducing there in a more efficient way, and finally converting back from the Montgomery domain. These conversions create too much overhead when just a single modular multiplication is needed, so Montgomery multiplication is not a suitable solution in our case. Integer multiplication with Barrett reduction [9] poses the same problem and it additionally needs more memory to store the intermediate multiplication result.

The main problem with the interleaved row-multiplication and reduction method [10] are the intermediate reductions. The best solution for this is a Barrett-like underestimation for the number of times the modulus needs to be subtracted from the intermediate result as described in [11]. However, this solution turned out to be relatively slow.

In our particular scenario where fast modular exponentiation is available, a third option becomes very interesting. Using the binomial theorem we can convert modular multiplications to modular exponentiations, which can be computed efficiently on the co-processor. Eq. 4 shows two ways to do the conversion:

$$ (a + b)^2 - a^2 - b^2 = 2 \cdot a \cdot b \mod n $$

$$ (a + b)^2 - (a - b)^2 = 4 \cdot a \cdot b \mod n $$

These equations allow to obtain the result of a multiplication without actually computing this multiplication. However, they require other modular operations. Modular addition and subtraction have been briefly discussed at the beginning of Sect. 3. Modular exponentiation has been discussed in Sect. 3.1. Modular division by two or four can be easily implemented as a modular right shift. If the least significant bit (LSB) of the dividend is zero, modular division by two is merely a right shift. If the LSB is one, we have to compute the sum of the dividend and the modulus and then right shift the result. Note that this only works if the modulus is an odd number, which is always the case in the DAA scheme. Modular division by four is then merely repeated division by two.

Table 1 compares the number of operations needed for both options in Eq. 4. Depending on the performance of the cryptographic co-processor on a given card, it is possible to decide which one will be faster. Oddly enough, we have indeed observed that some cards can perform a modular squaring faster than a modular right shift.

<table>
<thead>
<tr>
<th>Operation</th>
<th>number of executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>modular addition/subtraction</td>
<td>3</td>
</tr>
<tr>
<td>modular squaring</td>
<td>3</td>
</tr>
<tr>
<td>modular right shift</td>
<td>1</td>
</tr>
</tbody>
</table>

This fast multiplication mechanism can also be used for normal (non-modular) multiplication, by applying the row-wise multiplication algorithm with a larger word size. The basic multiplications in
This operation with single exponentiations and multiplication of the modulus, the modular multiplications have the same result as regular multiplications.

### 4. RESULTS

To determine the practical applicability of our implementation, we performed tests on two different Java Card smart cards. “Card 1” runs version 2.1.1 of the Java Card standard [4], “card 2” runs version 2.2.1 [5].

#### 4.1. Timing Details

Table 2 presents a detailed overview of the on-card execution times of the DAA protocols. Note that execution times on the host are not included as we assume the TPM to be the bottleneck. We performed tests on both cards and took average timings of 15 executions. The overall performance difference between the two cards shows that execution times heavily depend on both the implementation of the virtual machine and the cryptographic library/co-processor, and less on the version of the Java Card standard used.

<table>
<thead>
<tr>
<th>Part of the protocol</th>
<th>Bit sizes</th>
<th>Card 1</th>
<th>Card 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>signing protocol</td>
<td></td>
<td>12.5 sec</td>
<td>4.2 sec</td>
</tr>
<tr>
<td>communication</td>
<td></td>
<td>7.4 sec</td>
<td>2.4 sec</td>
</tr>
<tr>
<td>random number generation</td>
<td>2232 bits</td>
<td>6%</td>
<td>15%</td>
</tr>
<tr>
<td>modular multi-exponentiation</td>
<td>1024 bit modulus and bases, 1752 bit and 400 bit exponents</td>
<td>50%</td>
<td>2%</td>
</tr>
<tr>
<td>→ single RSA exponentiations</td>
<td>1024 bit modulus, 1024 bit, 728 bit, 729 bit, 400 bit exp.</td>
<td>35%</td>
<td>50%</td>
</tr>
<tr>
<td>→ modular multiplications</td>
<td>1024 bit intermediate results and modulus</td>
<td>65%</td>
<td>50%</td>
</tr>
<tr>
<td>→ squaring with RSA</td>
<td>1024 bit modulus</td>
<td>19%</td>
<td>8%</td>
</tr>
<tr>
<td>→ modular add, subtract, shift</td>
<td>1024 bit numbers</td>
<td>81%</td>
<td>92%</td>
</tr>
<tr>
<td>hash (SHA-1)</td>
<td>240 bits to 160 bits, and 320 bits to 160 bits</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>(non-mod) multiplication with RSA</td>
<td>long operands: (160 bits) · (1512 bits)</td>
<td>28%</td>
<td>25%</td>
</tr>
<tr>
<td>→ squaring with RSA</td>
<td>1024 bit modulus</td>
<td>25%</td>
<td>7%</td>
</tr>
<tr>
<td>→ modular add, subtract, shift</td>
<td>1024 bit numbers</td>
<td>75%</td>
<td>93%</td>
</tr>
<tr>
<td>(non-mod) regular multiplication</td>
<td>short operands: (160 bits) · (160 bits)</td>
<td>8%</td>
<td>4%</td>
</tr>
<tr>
<td>addition</td>
<td>(1672 bits) + (1752 bits) and (320 bits) + (400 bits)</td>
<td>7%</td>
<td>5%</td>
</tr>
</tbody>
</table>

The observations regarding the signing protocol are similar. About 50% of the overall execution time is due to modular multi-exponentiation on both cards. As before, a large fraction of this time is eventually caused by modular addition, subtraction, and division on both cards. This bottleneck is due to the pure Java implementation of these operations. Modular squaring on the co-processor causes the remaining 19% resp. 8% of the execution time.

Whether a (non-modular) multiplication is faster with or without support of the co-processor depends on the size of the operands and the speed of the particular card. To multiply two 160-bit numbers both options take comparable time on card 1 while the option without the co-processor is more than twice as fast as the other on card 2. In contrast, when multiplying a 160-bit number with a 1512-bit number using the co-processor yields a speed-up of about a factor of three on card 1 and an improvement of about 50% on card 2. For each multiplication, our implementation of the signing protocol uses the faster option depending on the size of the operands. The timings in Table 2 reflect these choices.

In summary we can observe that modular multiplication is the most time consuming operation. Recall that we implemented this operation by converting the multiplication to squarings which can be computed on the co-processor, see Eq. 4. For comparison, we also implemented the interleaved row multiplication and reduction algorithm of [11] in pure Java. Running on the faster card 2 and operating on 1024-bit operands, it turned out that this modular multiplication
is about 31 times slower than our solution. The poor performance is of course not due to the algorithm but caused by the inefficient execution of Java code.

4.2. Memory Requirements

Table 3 shows the memory requirements of our implementation. Note that the code size includes only the size of the applet’s own code, but not APIs it might use which are already present on the cards. Also, the RAM requirement does not include the RAM usage of the built-in RSA methods, since memory for these is reserved in a different way by the API itself.

Since most Java Card smart cards have at least 32kB of EEPROM and 1kB of RAM available, neither code size nor volatile memory demands of our implementation pose a problem.

Table 3. Memory requirements of the implementation.

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEPROM (code size)</td>
<td>3648 bytes</td>
</tr>
<tr>
<td>EEPROM (variables)</td>
<td>2227 bytes</td>
</tr>
<tr>
<td>RAM</td>
<td>605 bytes</td>
</tr>
</tbody>
</table>

5. CONCLUSION

We presented the first implementation of the (simplified) DAA protocols suitable for contemporary Java Card smart cards. We pointed out performance bottlenecks and provided efficient solutions which allow our implementation to terminate within acceptable time (4.2 sec for the signing protocol on card 2). An implementation of the full DAA scheme can directly benefit from our implementation tricks which should yield a similar performance improvement.

On both cards, most of the execution time is due to modular multiplication and can eventually be assigned to modular addition, subtraction, and division. Our future work thus focuses on speeding up these operations, for which we see two options. On the one hand we will try to implement these operations more efficiently, possibly by outsourcing even more computation to the co-processor. On the other hand an interesting alternative may be to use cards implementing version 2.2.2 [12] or 3.0.1 [13] of the Java Card standard. Starting from this version the standard includes support for large numbers and their (non-modular) addition, subtraction, and multiplication.

Acknowledgements

This work was partially supported by the IAP Programme P6/26 BCRYPT of the Belgian State, by FWO project G.0300.07, by the European Commission under contract number ICT-2007-216676 ECRYPT NoE phase II and by K.U. Leuven-BOF (OT/06/40). The information in this document reflects only the author’s views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user therefore uses the information at its sole risk and liability.

6. REFERENCES