Performance issues of Selective Disclosure and Blinded Issuing Protocols on Java Card

Hendrik Tews* and Bart Jacobs

Digital Security Group, Radboud Universiteit Nijmegen,
The Netherlands
`http://www.cs.ru.nl/{~tews,~bart}`

Abstract. In this paper we report on the performance of the RSA variants of Brands protocols for zero-knowledge proof and restrictive blinded issuing [1]. The performance is relatively bad: For 4 attributes and an RSA key size of 1280 bits, blinded issuing takes about 10 seconds and the zero-knowledge proof takes about 9 seconds. For 2 attributes the zero-knowledge proof drops to 5 seconds. The poor performance comes from the fact that the cryptographic coprocessor on the Java card can only be employed in very limited ways. With appropriate support of the cryptographic coprocessor both protocols would run much faster.

Key words: Java Card · selective disclosure · blinded issuing · performance

1 Introduction

This paper has a (partly) negative message: it shows that certain desirable things cannot be done, ... currently. In particular, it shows, via various performance measurements, that the current generation of Java cards is unsuitable for advanced cryptographic protocols, such as privacy-friendly selective disclosure of attributes, via zero-knowledge proofs. The simple reason is that current cards are too slow. The more subtle reason is that the Java-Card API does not permit access to the (fast!) cryptographic primitive operations on the cryptographic coprocessor. The hope that a clear exposition of this problem will contribute to a solution in the near future is an important motivation for writing this paper.

The emergence of severe vulnerabilities in the Mifare Classic chip card [4, 11, 5], which is heavily used in public transport (like London's Oyster, or the Dutch OV-chipkaart), has led to renewed interest in smart cards for public transport. The current generation of cards is identity-based:

– cards have a fixed UID in anti-collision that allows tracing of individuals, also outside the context of public transport, since this UID can be picked up by any reader;

* Sponsored by the NLnet foundation through the OV-chipkaart project.

– cards have a fixed (application level) identity that is used in every transaction, enabling detailed travel logging and profiling of individuals (with a personalized card).

There is a desire, at least in certain communities, to move to more privacy-friendly mechanisms, based for instance on attributes instead of identities. After all, in most cases there is no compelling reason why you should tell who you are upon entering a bus; possession of a valid travel attribute should be sufficient. Advanced cryptographic protocols have been developed for such attribute-based access control, such as [1] based on zero-knowledge and blind signatures or [13] based on bilinear pairings on elliptic curves. In this paper we evaluate the approach of Stefan Brands [1] via a prototype implementation on Java Card. We focus on two of the crucial protocols, namely for selective disclosure of attributes and for blinded issuing of a signed attribute expression on currently publicly available Java cards. This is part of a project that is informally called "OV-chip 2.0". As Brands suggested, we combine the RSA variants of his proof of knowledge protocol with his protocol for blinded issuing. We equip the protocols with the necessary code for initialization and key generation and implement everything in a Java-Card applet and an appropriate host-driver application. The host driver runs on a normal PC and talks to a Java card through a CCID compliant smart-card reader. The host driver can install the applet, download the key material and personalize the applet, run the protocols, and, of course, measure their execution time.

We actually implemented two versions of the applet. The first one, the *coprocessor-enabled applet*, performs the computations as far as possible on the cryptographic coprocessor of the Java card. The second one, the *pure Java-Card applet*, computes everything on the virtual machine of the Java card. The host driver can talk to both applets.

The pure Java-Card applet is, of course, very very slow. It is only discussed here to provide an impression about the speedup of the cryptographic coprocessor. However, also the coprocessor-enabled applet is not as fast as we wished. For 4 attributes and an RSA key size of 1280 bits, blinded issuing takes about 10 seconds and the zero-knowledge proof takes about 9 seconds on the coprocessor-enabled applet. When using only 2 attributes the zero-knowledge proof takes about 5 seconds. The main performance limitation is the Java-Card API (together with the provided security of Java cards) that permits no adequate access to the cryptographic coprocessor. We analyze the problems that lead to this unexpectedly bad performance in more detail in Section 2 and Section 3.

To achieve better performance for Brands protocols one needs access to the native (assembly) methods for standard and modular multiplication, modular exponentiation and for division that fully exploit the cryptographic coprocessor on the card. For using elliptic curves with the discrete log (DL) variants of Brands protocol one would need access to native methods for point addition and scalar point multiplication. Even the current Java Card 3.0 draft does not specify any of these methods although any card with support for RSA and elliptic curves does contain such methods. With adequate access to the cryptographic coprocessor Brands protocols would probably run in about 1 second.

Our implementation is based on the Bignat library, a newly developed library for big natural numbers on Java Card. The implementation further exploits the Java-Card protocol layer for the communication between the applet and the host driver. The Java-Card protocol layer is a custom layer for remote method invocation on Java cards that supports methods with an arbitrary number of arguments and results of up to 32 KByte in size. The complete sources are available for download from `https://ovchip.cs.ru.nl/OV-chip_2.0` with one exception: Because of Brands patents on his protocols the few methods that implement the protocol for the two applets and the host driver are missing from the distribution.[1] The protocol is however fully described in [1] and in Appendix B of this paper so that it should be not too difficult to get the applets running for research purposes.

This paper is structured as follows. Section 2 gives insight into the Java-Card API and explains why currently any implementation of Brands protocols on Java cards will have to fight with performance problems. Section 3 presents our Bignat library for operations on big integers on Java Card. Section 4 describes the protocols that we implemented and presents our performance measurements. In Section 5 we shortly discuss elliptic curves and Section 6 concludes. Appendix A shortly introduces Montgomery multiplication, because it is mentioned very often in this paper. The Appendix B contains the technical description of the implemented protocols, similar to the descriptions in [1] but with adoptions for our implementation.

2 Performance Limitation in the Java-Card API

The performance critical part in the RSA variants of Brands protocols are expressions of the form $(g_1^{a_1} \cdots g_k^{a_k}) \;\texttt{mod}\; n$, which we call *modular multi-powers* in the sequel. Such a modular multi-power encodes the attributes $a_1, \ldots, a_n$ of the card as numbers and its blinding in a blinded attribute expression. The length of the bases, exponents and the modulus determine the security level. A modulus n and bases b_i of 1280 bits and exponents a_i of 160 bits provide reasonable security over the next few years.

Apart from the modular multi-powers one also needs multiplication, modular multiplication, addition, division and modulus on big natural numbers. Current Java cards are equipped with a cryptographic coprocessor and a suitable native (assembly) library that can perform these operations in a reasonably fast way. For instance, RSA public key encryption takes only 120 milliseconds for a RSA key and a cipher text of 1280 bits and a public exponent of 200 bits. This leads to speculated 0.3 milliseconds for one modular multiplication of 1280 bit numbers. However, Java-Card applets can only use the public Java-Card API [8, 9] and extending this API with new native (assembly) methods is not

[1] The current patent owner is Microsoft. Microsoft lawyers are still pondering our request from January 2009 to permit the distribution of the complete sources.

permitted. The current Java-Card API version 2.2.2 [9] gives very limited access to the cryptographic coprocessor in class `BigNumber` in the optional package `javacardx.framework.math`. This class contains multiplication and addition, but no modular multiplication, no division or even modular exponentiation. It appears that almost no cards are available that implement version 2.2.2 of the Java-Card API. Until now we only found two such cards: The Athena IDProtect[2] and a recent JCOP31 card from NXP. Both do not support the optional package `javacardx.framework.math`. The older API version 2.2.1 [8], which is implemented by most of the currently available cards, does neither contain the package `javacardx.framework.math` nor the `BigNumber` class.

Without direct access to the cryptographic coprocessor the only remaining possibility is to trick one of the high-level cryptographic methods into performing, for instance, a modular multiplication or a modular exponentiation. Current Java cards provide a number of such high-level methods that perform big-integer calculations internally, for instance for RSA (encryption, decryption and key generation), Diffie-Hellman key exchange and DSA. However, internally most of these high-level methods use random padding or randomly generated arguments, which cannot be controlled from the API level. These random ingredients are essential for the security of those methods, but they make it impossible to turn them into a big-integer operation.

We only found one exception: The `ALG_RSA_NOPAD` cipher algorithm contains no random padding and can be used to compute a modular power $g^a \;\mathsf{mod}\; n$. There are some restrictions on the arguments, but one can easily work around them. Our NXP JCOP cards, for instance, only support moduli between 64 and 244 bytes. The modulus must further have a first non-zero byte and a length (in bytes) that is divisible by 4. As a further peculiarity an exponent of 0 yields 0 as cipher text, that is, $x^0 = 0$ when using the RSA cipher. With an exponent of 1 the `RSA_NOPAD` cipher can be used to compute a modulus $g \;\mathsf{mod}\; n$. This is, however, not very useful, because g cannot be longer than n (in bytes) and for such numbers a simple schoolbook division achieves the same performance.

On currently available Java cards it is impossible to directly use the cryptographic coprocessor for multiplication, modular multiplication or division of big integers.

It took us a some time to remember that $ab = \frac{(a+b)^2-a^2-b^2}{2}$. For odd moduli this equation can actually be turned into a method for computing modular products. This method will be called *squaring multiplication* in the following. For one modular product squaring multiplication needs to do 3 modular squares, 2 subtractions, 1 right shift and 1 to 4 additions. The number of additions varies, because, for instance, $(a+b)^2 \;\mathsf{mod}\; n$ might be smaller than $a^2 \;\mathsf{mod}\; n$ and in this case $((a+b)^2 - a^2) \;\mathsf{mod}\; n = ((a+b)^2 \;\mathsf{mod}\; n) - (a^2 \;\mathsf{mod}\; n) + n$.

On Java cards squaring multiplication gives a big speedup, because the squares can be computed on the cryptographic coprocessor with the help of the RSA cipher. For instance for numbers of 1280 bits, one square costs only 14 milliseconds, while one addition, which must be done on the Java virtual

[2] See http://www.athena-scs.com.

machine of the card, costs 75 milliseconds. Montgomery multiplication, which must also run on the Java virtual machine of the card, requires 320 additions for numbers of 1280 bits and takes about 25 *seconds*. A squaring multiplication for such numbers costs only between 350 and 580 milliseconds.

The RSA cipher on Java Card computes only modular exponents. But if one chooses a modulus $n > (a+b)^2$ then one can use squaring multiplication also to compute a normal (non-modular) product ab.

We can conclude here that the Java-Card API does not facilitate the implementation of advanced cryptographic protocols, because the API does not give access to the fast big-integer operations that are available on the card. Without support from the cryptographic coprocessor one is forced to implement the missing operations in Java using bytes and shorts (as there are usually no 32 bit integers on a Java card). With the overhead of the Java Virtual Machine added on top of the limited execution speed of the main processor this will almost certainly yield an unacceptable performance. As things stand, the situation is not likely to improve much, because the current draft of the Java-Card specification for upcoming Java Card 3.0 [10] does not contain any additions to the `BigNumber` class that is already present in version 2.2.2. So even if some future cards implement the relevant optional package, one still has to implement division and addition in the Java Virtual Machine. With the trick of squaring multiplication, the cryptographic coprocessor can speed up multiplication and modular multiplication but a multiplication directly on the coprocessor would probably still be about 100 times faster than our squaring multiplication method.

One aim of this paper is to draw attention to the limitations of the Java Card API for advanced cryptographic protocols and to motivate the Java-Card community in general and the card producers in particular to allow access to basic cryptographic operations on the coprocessor via extensions of the Java-Card API. The paper illustrates the need for such extensions for the next generation of (privacy-friendly) smart card applications.

3 Bignat: A Big-Integer library for Java Card

The limitations of the Java-Card API force us to perform some computations in a big-integer library on Java Card. We decided to implement such a library from scratch, for the following reasons. Although different big-integer libraries have been developed in the past in different projects [2, 3], no such library is currently publicly available. As [2, 3] already point out, porting an existing big-integer library does not make much sense because of the limitations of Java Card. The absence of a garbage collector, for instance, enforces a completely different Java programming style, in which all allocations are performed at applet initialization time and temporary objects appear in the interface of those methods that need them. We further believe that a library interface tailored towards the application can improve the performance. For Brands protocols, for instance, the bases g_i in a blinded attribute expression $(g_1^{a_1} \cdots g_k^{a_k}) \bmod n$ are constant, which makes special optimizations possible.

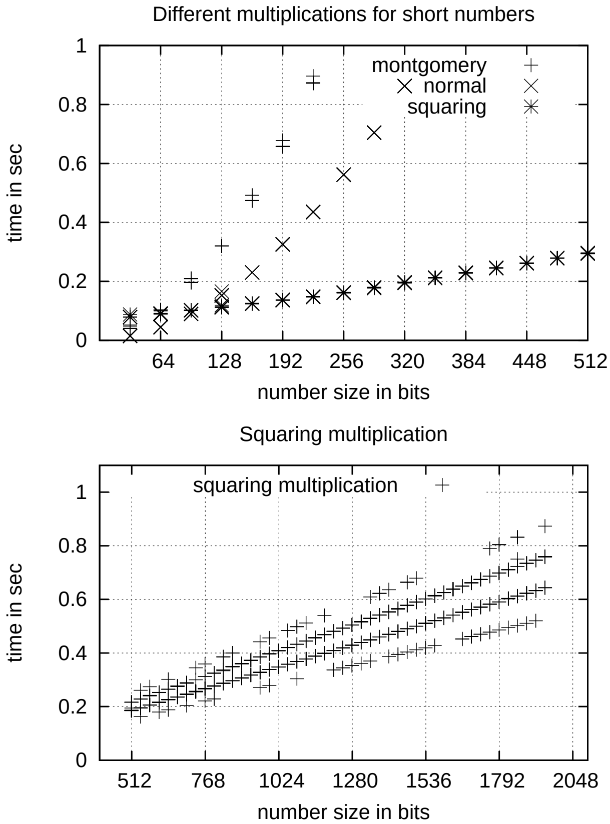

Fig. 1: Performance of multiplication. The top chart compares Montgomery, school-book, and squaring multiplication for short numbers. Squaring multiplication is fastest from about 92 bits. The bottom chart displays the performance of squaring multiplication for large numbers. One can clearly recognize the different number of additions that were necessary for the randomly chosen parameters. All measurements were done over the contact interface of the card.

Our library implements natural numbers of arbitrary but fixed size that must be specified at object creation time. The numbers are mutable; for many operations the result is stored in the object on which the operation is invoked. If this object is not big enough to hold the result, an exception is thrown. The library implements addition, subtraction, multiplication and division with their school-book algorithms. The Bignat library additionally implements Montgomery multiplication (see Appendix A) and squaring multiplication, which are both modular multiplications. Squaring multiplication employs the cryptographic coprocessor of the card via the `RSA_NOPAD` cipher. Figure 1 shows the performance of these different multiplication methods. Montgomery multiplication has a quadratic complexity, its computation time rises from 4.1 seconds for 512 bit numbers over 25 seconds for 1280 bit to 64 seconds for 2048 bit numbers. As the bottom chart in Figure 1 shows, squaring multiplication is much faster. However, from the RSA encryption performance we estimate that a 1280 bit multiplication performed directly on the cryptographic coprocessor would only take 0.3 *milliseconds.*

The Bignat library contains a wrapper method for accessing the cryptographic coprocessor via the RSA cipher for computing modular powers. The

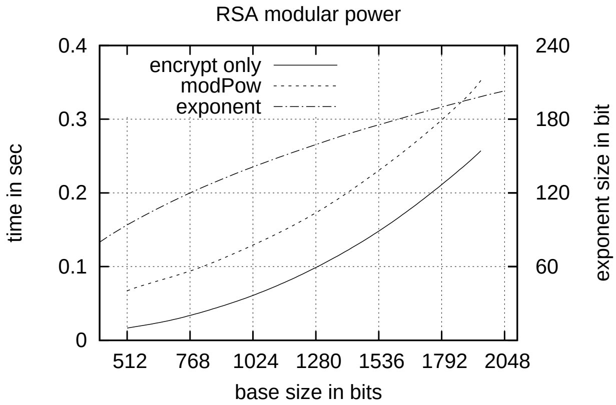

Fig. 2: Performance of computing exponents on the cryptographic coprocessor (contact interface only). Cipher and key initialization has a significant overhead over the pure encryption time. The exponent length depends on the base length and is displayed on the right y-axis.

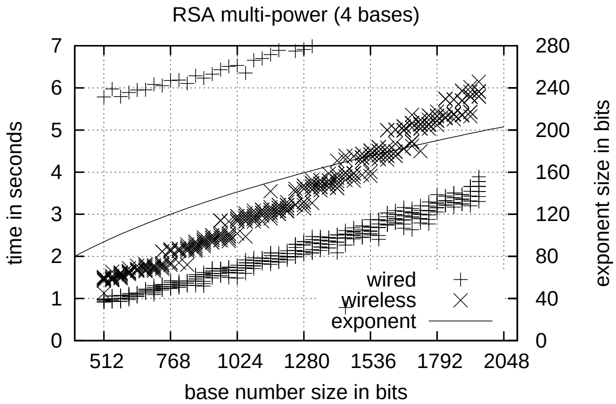

Fig. 3: Performance of the RSA method to compute modular multi powers. The first computation for any base size takes much longer and is partly outside the chart. For unknown reasons the computation takes longer over the wireless interface. The exponent length is displayed on the right y-axis as before.

wrapper works around known problems, for instance, it correctly computes $x^0 = 1$. Figure 2 shows the performance of this method for computing modular powers. For the measurements the size of the exponent was chosen such that it provides similar security for Brands protocols as an RSA modulus of the same size as the bases. The security level of the RSA modulus is thereby estimated following Lenstra [6]. The exponents we use grow from 94 bits for bases of 512 bits to 198 bits for bases of 1952 bits. In Figure 2 the third line displays the exponent length against the right y-axis.

For modular multi-powers $(g_1^{a_1} \cdots g_k^{a_k}) \ \mathsf{mod}\ n$ the Bignat library contains two specialized methods: the *RSA multi-power method* that uses the cryptographic coprocessor as much as possible and the *simultaneous squaring multi-power method* that computes the result entirely without the cryptographic coprocessor.

The RSA multi-power method computes the single modular exponents $g_i^{a_i} \ \mathsf{mod}\ n$ with the RSA cipher of the card and multiplies the results with squaring multiplication. Figure 3 displays the performance of this method for

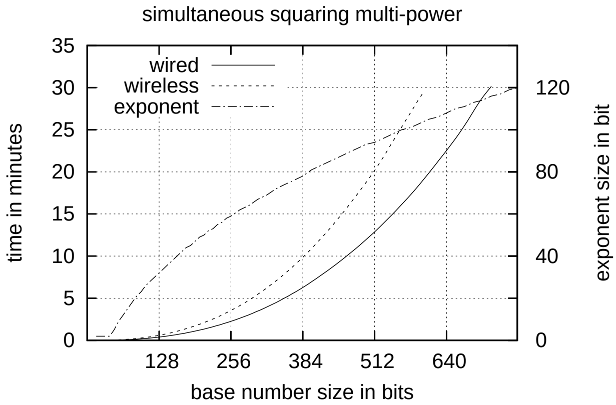

Fig. 4: Performance of the simultaneous squaring multi-power method. Note that the left y-axis displays *minutes*. The measurement has been manually stopped after the running time exceeded 30 minutes. The exponent length is displayed on the right y-axis as before.

computing a multi power with 4 bases (i.e., $k = 4$). The subtractions and additions inside squaring multiplication are responsible for a significant part of the computation time. Counting subtractions as additions, the computation of one multi-power consists of 9–18 additions, which costs between 0.6 and 1.3 seconds for 1280 bit numbers.

The simultaneous squaring multi-power method uses the simultaneous squaring method on the basis of Montgomery multiplication. It takes advantage of the fact that for Brands protocols the bases g_i are constant and uses a precomputed table of all possible products of the bases g_i. Therefore it only needs about $2|a|$ Montgomery multiplications, where $|a|$ denotes the maximal size of the exponents a_i in bits. The simultaneous squaring method requires that all bases and also the precomputed table of factors are provided in Montgomery representation. Figure 4 shows the performance of the simultaneous squaring multi-power method. It is clear that on Java Card the simultaneous squaring multi-power method has mostly anecdotic value. We only discuss it here for two reasons. Firstly, it provides an impression of the performance benefits of the cryptographic coprocessor on Java Card. Secondly, an implementation based on the simultaneous squaring multi-power method can easily ported to a platform without cryptographic coprocessor support, such as a smart phone.

4 Implemented protocols and their Performance

In this section we describe in somewhat more detail the protocols that we implemented and show their performance on current Java cards. For reasons of space the precise technical description of the protocols has been moved to Appendix B.

We actually implemented two applets, the *coprocessor-enabled applet* and the *pure Java-Card applet*. The same host driver is used to control both applets. The coprocessor-enabled applet uses internally the RSA multi-power method while the pure Java-Card applet uses the simultaneous squaring multi-power method. The pure Java-Card applet is only shown here for the comparison.

Both applets are functionally equivalent. They hold k attributes $a_1, \ldots, a_k$ that could encode the card type (e.g., whether it is a month card or a reduction card), the expiration date, possibly a balance, and so on. One of the attributes is the private key of the applet, which will never be disclosed to anybody. From the attributes the applet computes its blinded attribute expression $A = b^v\, g_1^{a_1} \cdots g_k^{a_k} \ \mathsf{mod}\ n$, where the bases g_i, the RSA modulus n and the public RSA exponent v are public system parameters. The b is a blinding factor that is private to the applet and that ensures that the attribute expression A does not function as a pseudonym. To ensure that the attributes are original the whole attribute expression A is signed. The signature is constructed in such a way that the signing authority does not see the resulting signature and therefore cannot use the signature to recognize the applet later.

In our implementation one can configure the number of attributes k and the size of the RSA modulus n and the size of the public RSA exponent v at initialization time. A modulus of 1280 bits and an exponent of 160 bits are sufficient to ensure security over the next few years. Together the host driver and each of the applets implement the following protocols (for a complete technical description of the protocols see Appendix B):

Key Setup and Initialization The host driver generate the keys, the bases $g_1, \ldots, g_k$ and chooses the first attributes $a_1, \ldots, a_k$ of the applet. The key material, the bases and the attributes are installed in the applet and the applet computes its first attribute expression A. As last part of the initialization the resign protocol is run to let the applet change its blinding b and to equip it with a valid signature.

Resign Protocol The applet shows its blinded attribute expression A and the signature and the host driver checks the validity of the signature (this check is of course left out if resigning runs as part of the initialization). The host driver can then change selected attributes (for instance to change the expiration date) and the applet chooses a new blinding b. Finally the applet obtains a new signature for the changed blinded attribute expression.

Gate Protocol The applet shows its blinded attribute expression A and the signature, which is checked by the host as in the resign protocol. The applet then proves with a zero-knowledge proof that it knows suitable attributes $a_1, \ldots, a_k$ that give rise to A. Thereby the host learns nothing about the attributes.

A feature currently missing is the partial disclosure of some attributes. For instance, at the gate the card would disclose its card type and claim that the expiration date lays in the future.

Figure 5 shows the transaction times for the complete system, using either the coprocessor enabled or the pure Java-Card applet. For 4 attributes and a RSA key size of 1280 bits and a public RSA exponent of 160 bits the resign protocol takes between 10 and 11 seconds and the gate protocol between 8 and 9 seconds (the lines in Figure 5 show the average of a number of measurements). For 2 attributes resigning takes between 8 and 9 seconds and the gate protocol between 5.2 and 5.8 seconds. The applet is therefore probably too slow for

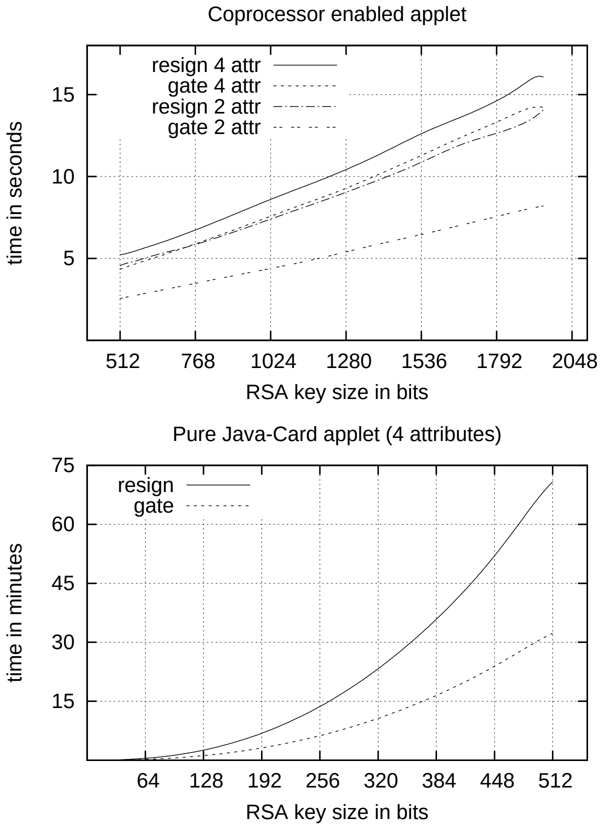

Fig. 5: Performance of the two applets. Note that y-axis of the top chart is in *seconds*, while for the bottom chart it is in *minutes*. Timings are complete transaction times over the wired interface, that is, including the computation of the host driver, the communication time, and, of course, the computation of the applet. The coprocessor enabled applet only supports key sizes between 64 and 244 bytes, because the RSA cipher on our cards only supports these key sizes. The measurement for the pure Java-Card applet has been stopped after the key size 512.

public transport and most other applications. However, the performance of the coprocessor enabled applet shows that with proper support from a cryptographic coprocessor Brands protocols could already be used today on Java cards. With an appropriate API for the coprocessor we estimate that transaction times of about 1 second are possible for currently available Java cards.

5 Variants based on Elliptic curves

Brands protocols for selective disclosure and blinded issuing do also exist in a discrete log (DL) variant. This variant can be implemented on the basis of elliptic curves [12]. The main advantage of elliptic curves is that they permit much smaller key sizes—keys of 150–200 bit would be sufficient. Therefore the numbers that one has to manipulate for the DL variant are much smaller: 150–200 bits instead of 1200–2000 bits as for the RSA variants. The disadvantage is that the base operation on elliptic curves —addition of two points— is much more involved.

Although many Java cards implement cryptographic protocols based on elliptic curves, there is no support for adding points of an elliptic curve in the Java-Card API. There are two high-level elliptic curve related methods in the Java-Card API: ECDSA, the digital signature algorithm over elliptic curves and ECDH, Diffie-Hellman key agreement over elliptic curves. ECDSA specifies some random padding, so it cannot be used to perform addition or scalar multiplication of points. It should be possible to trick the Diffie-Hellman algorithm into performing a scalar multiplication of an elliptic-curve point. However, a point of an elliptic curve has two coordinates and the Diffie-Hellman key agreement on Java Card only returns the x-coordinate. This could suffice for those protocols that just do one scalar multiplication at the end, because then the missing y-coordinate can be reconstructed on the host. For Brands protocols, however, one would have to reconstruct the missing y-coordinate on the card. As this involves a square root we are very sceptical about the performance benefits of exploiting the Diffie-Hellman key agreement.

We have not done any experiments yet, but we expect that Brands DL variants would actually be slower than our coprocessor enabled applet. We expect that the disadvantage of the missing coprocessor support outweighs the advantage of shorter keys.

6 Conclusion

In this paper we evaluated the performance of Brands selective disclosure and blinded issuing protocols on currently publicly available Java cards. The performance is not sufficient for most applications. A zero-knowledge proof for 4 attributes takes about 9 seconds, while blinded issuing takes about 10 seconds for an RSA key size of 1280 bits. For two attributes the zero-knowledge proof takes about 5 seconds for the same RSA key size. Limitations in the Java-Card API for accessing the cryptographic coprocessor are solely responsible for the bad performance. While we found a way to compute modular powers $g^a \mod n$ on the coprocessor by abusing RSA public key encryption, there is no direct way to execute a modular big-integer multiplication on the coprocessor. Montgomery multiplication executed on the Java Card Virtual machine takes 25 seconds for 1280 bit numbers. The familiar equation $(a + b)^2 = a^2 + 2ab + b^2$ can be used to dramatically speed up the computation of a modular product because the squares can be computed on the cryptographic coprocessor. With this trick one modular multiplication takes between 0.3 and 0.6 seconds for numbers of 1280 bits. In contrast, we estimate that a modular multiplication directly on the cryptographic coprocessor would only take about 0.3 *milliseconds* for numbers of this size.

We believe that, with appropriate support in the API, running times in the order of 1 second are possible.

To facilitate the development and use of new cryptographic protocols the Java-Card API should as soon as possible be enriched with at least two optional classes. One for the basic big-integer operations that are missing from `java-`

`cards.framework.math.BigNumber`: modular multiplication, modular addition, division, modulus, modular powers, and modular inverse. The second class should contain addition and scalar multiplication of points on elliptic curves. Note that all these operations are already implemented on most cards, so it is only a question of exporting them to the Java-Card API.

Acknowledgements We are grateful to Wojciech Mostowski for his help and his insights on Java-Card programming.

References

1. S. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy.* MIT Press, 2000. Freely available via www.credentica.com.
2. T. Dowling and A. Duffy. Java card key generation for identity based systems. Technical Report NUIM-CS-TR-2005-01, Department of Computer Science, National University of Ireland, Maynooth, February 2005. Available at http://www.cs.nuim.ie/research/reports/2005.
3. T. Elo and P. Nikander. Decentralized authorization with ECDSA on a Java smart card. In *Proceedings of the fourth working conference on smart card research and advanced applications on Smart card research and advanced applications*, pages 345–364, Norwell, MA, USA, 2001. Kluwer Academic Publishers.
4. F. Garcia, G. de Koning Gans, R. Muijrers, P. van Rossum, R. Verdult, R. Wichers Schreur, and B. Jacobs. Dismantling MIFARE Classic. In S. Jajodia and J. Lopez, editors, *Computer Security – ESORICS 2008*, number 5283 in Lect. Notes Comp. Sci., pages 97–114. Springer, Berlin, 2008.
5. Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly pickpocketing a Mifare Classic card. In *IEEE Symposium on Security and Privacy (S&P '09)*, pages 3–15. IEEE, 2009.
6. A. Lenstra. Key lengths. In H. Bidgoli, editor, *Handbook of Information Security*, volume II Information Warfare, Social, Legal, and International Issues and Security Foundations, pages 617–635. Wiley, 2006.
7. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 2001.
8. SUN Microsystems. Java Card v2.2.1 API. Available for free download on http://java.sun.com/javacard/specs.html, 2003.
9. SUN Microsystems. Java Card v2.2.2 API. Available for free download on http://java.sun.com/javacard/specs.html, 2005.
10. SUN Microsystems. Java Card Specifications Version 3.0. Available for free download on http://java.sun.com/javacard/downloads/, 2008.
11. K. Nohl, D. Evans, Starbug, and H. Plötz. Reverse-engineering a cryptographic RFID tag. In *17th USENIX Security Symposium*, pages 185–194, San Jose, CA, USA, 2004.
12. N.P. Smart. Elliptic curve based protocols. In I.F. Blake, G. Seroussi, and N.P. Smart, editors, *Advances in Elliptic Curve Cryptography*, number 317 in LMS, pages 3–19. Cambridge Univ. Press, 2005.
13. E. Verheul. Self-blindable credential certificates from the Weil pairing. In C. Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001*, number 2248 in Lect. Notes Comp. Sci., pages 533–550. Springer, Berlin, 2001.

Appendix A Montgomery Multiplication

This appendix briefly describes Montgomery multiplication as used in our implementation, see [7, Algorithm 14.36] for a more general description. For clarity we use $- \cdot -$ to denote standard multiplication.

Let n be an odd modulus and $l = \left\lceil \frac{|n|}{8} \right\rceil$ be the number of bytes it occupies. The *Montgomery factor R with respect to n* is then defined as $R = 2^{8l} \bmod n$. Its modular inverse with respect to n is denoted with R^{-1}, so $(R \cdot R^{-1}) \bmod n = 1$. The *Montgomery representation* of a number x is $(x \cdot R) \bmod n$, where n is clear from the context. *Montgomery multiplication*, denoted with $- \times -$, is defined as follows: $x \times y = (x \cdot y \cdot R^{-1}) \bmod n$. If the arguments are in Montgomery representation then so is the result: $(x \cdot R) \times (y \cdot R) = (x \cdot y) \cdot R \bmod n$. To compute the modular product of a fixed number of factors it is not necessary to convert all factors into Montgomery representation. Instead one adds an additional correction factor R^k, where k is the number of factors. For instance $(a_1 \cdot a_2 \cdot a_3) \bmod n = a_1 \times a_2 \times a_3 \times R^3$. To convert a number from Montgomery representation back to normal one multiplies with R^{-1} or exploits $x = (x \cdot R) \times 1$.

Montgomery multiplication can be computed with a modified schoolbook multiplication algorithm. To compute $x \times y$ one decomposes y into l byte-digits $y_l y_{l-1} \cdots y_1 y_0$ and performs precisely l multiplication rounds. In multiplication round i one adds $x \cdot y_i$ to the accumulator and shifts the accumulator one byte to the right. Before shifting one makes the last digit of the accumulator equal to 0 by adding a suitable multiple of n. For an odd modulus n such a suitable multiple does always exist. Which multiple of n to use can be deduced from the last byte of the accumulator. The final accumulator might be bigger than n, so one has to take the modulus with respect to n at the end. The accumulator must be capable of holding l bytes plus 9 bits. In our implementation all numbers that come in contact with Montgomery multiplication are simply allocated with $l+2$ bytes.

Appendix B Description of the Implemented Protocols

This appendix describes the protocols from [1] that we implemented. The technical description is for the coprocessor enabled applet. The changes for the pure Java-Card applet are summarized at the end of each subsection.

B.1 Initialization and Personalization

Parameter setup Before starting the following points must be configured.

- The number k of attributes each applet possesses.
- The size of the RSA modulus n in bits, denoted with $|n|$ in the sequal.
- Optionally the size of the public RSA exponent v, denoted with $|v|$ in the following. If not configured, $|v|$ is derived from $|n|$ using Lenstras estimations on the security level of RSA keys [6].

The following system parameters are computed once. If not further determined the items are randomly chosen to satisfy the relevant conditions.

- The RSA modulus n of size $|n|$, where $n = p\,q$ with p and q prime.
- The public RSA exponent v of size $|v|$, such that v is prime and coprime to $\varphi(n) = (p-1)(q-1)$, where φ is Euler's totient function.
- The modular inverse of v with respect to $\varphi(n)$, denoted with v^{-1} in the following.
- The private system key $x \in \mathbb{Z}_n^*$ (i.e., $\gcd(x, n) = 1$) and the public system key $h = x^v \ \mathsf{mod}\ n$.
- k bases $g_1, \ldots, g_k \in \mathbb{Z}_n^*$
- For the pure Java-Card applet, the Montgomery factor R with respect to n (see Appendix A).

For each applet that gets initialized one generates k random attribute values $a_1, \ldots, a_k \in \mathbb{Z}_v$.

Applet initialization After the coprocessor enabled applet or the pure Java-Card applet has been installed on a Java card the following protocol initializes the applet. In the protocol description A denotes the applet and H the host driver.

$H \longrightarrow A :\quad |v|, |n|, k$

the applet allocates all data structures

$H \longrightarrow A :\quad n, h, g_1, \ldots g_k, b, \Pi_g, a_1, \ldots, a_k, v, R$

where $b = 1$ is the initial blinding of the card,

Π_g, the precomputed products of the bases g_i, and R

are only used on the pure Java-Card applet.

The card computes its blinded attribute expression $A = b^v \prod g_i^{a_i}$

subsequently the resign protocol is run, whereby the attribute updates

are 0 and the signature check is left out on the host

For the pure Java-Card applet there are the following changes. In the second step the values of $h, g_1, \ldots, g_k$ and b are transformed into their Montgomery representation on the host before sending. The precomputed products Π_g is an array of $2^k - 1$ elements containing the Montgomery representation of all possible products of the bases g_i, except for the empty product 1. On the coprocessor enabled applet Π_g is an array with one arbitrary element, because the Java-Card protocol layer does not support empty arrays. The Montgomery factor R, which equals the Montgomery representation of 1, is needed on the pure Java-Card applet to initialize the accumulator for the simultaneous squaring multi-power method.

B.2 Resign protocol

The resign protocol is taken from [1, Section 4.2.2.]. When the resign protocol runs as part of the applet initialization the signature (S_c, S_r) is not yet initialized and therefore not checked in the first step.

$A \longrightarrow H:$ $\mathit{applet\_id}, A, S_c, S_r$, where

$$\mathit{applet\_id} = \begin{cases} 3 & \text{for the pure Java-Card applet} \\ 4 & \text{for the coprocessor enabled applet} \end{cases}$$

The host checks the signature $S_c \stackrel{?}{=} H\big(A,\ S_r^v\,(hA)^{-S_c}\big)$
and aborts the protocol if the equation does not hold.

$H \longrightarrow A:$ $\alpha, \hat{u}_1, \ldots, \hat{u}_k$,
where $\alpha \in \mathbb{Z}_n^*$ is the host commitment, and
$\hat{u}_i$ are the encoded attribute updates for
arbitrary attribute updates $u_1, \ldots, u_k$ such that

$$-v < u_i < v \text{ and } \hat{u}_i = \begin{cases} v + u_i & \text{for } u_i < 0 \\ u_i & \text{otherwise} \end{cases}$$

the applet computes its new attributes $a'_i = (a_i + \hat{u}_i) \bmod v$
and the updated attribute expression $A' = b^v \prod g_i^{a'_i}$

$A \longrightarrow H:$ $c = (S'_c + \beta_3) \bmod v$, where

$$S'_c = H\left(A'',\ \alpha\,\beta_2^v\,(hA')^{\beta_3}\right)$$
$$A'' = \beta_1^v A'$$

and $\beta_1, \beta_2 \in \mathbb{Z}_n^*$, $\beta_3 \in \mathbb{Z}_v$ are random
the applet additionally computes

$$q = (S'_c + \beta_3) \div v \qquad \text{(where } \div \text{ denotes integer division)}$$
$$b' = \beta_1 b \bmod n$$

$H \longrightarrow A:$ $r = \big(\alpha(hA'_h)^c\big)^{(v^{-1})}$, where $A'_h = A \prod g_i^{u_i}$

$A \longrightarrow H:$ acc, where $acc = \begin{cases} \mathit{true} & \text{if } r^v = \alpha(hA')^c \\ \mathit{false} & \text{if } r^v \neq \alpha(hA')^c \end{cases}$

if $acc = \mathit{true}$ the applet computes $S'_r = r\beta_2\beta_1^{S'_c}(hA')^q$
and atomically switches to use a'_i, b', A'', S'_c and S'_r
instead of a_i, b, A, S_c and S_r

In the preceding protocol H is a one-way hash function and $\div$ denotes integer division with the property $b\,(a \div b) + (a \bmod b) = a$ for arbitrary $a, b \in \mathbb{N}$. Our implementation uses 160 bit SHA-1 for H.

The host does not have access to the attribute values and must therefore compute its updated attribute expression A'_h in a different way. Both A' and A'_h

must be equal, otherwise the protocol fails. Note that the attribute updates u_i might be negative, so it might be necessary to compute modular inverses (with respect to n) in the computation of A'_h.

The protocol will also fail if one of the attribute updates yields an under or an overflow, that is if $a_i + u_i < 0$ or if $a_i + u_i \geq v$. Therefore, the host can only update those attributes where it knows something about the value. In our implementation this problem is solved with an additional status protocol, in which the applet sends all its data, including attributes and blinding, to the host. In a real application such a status protocol must, of course, not exist.

For the pure Java-Card applet the protocol is identical, except that A, S_r, α and r are transmitted in their Montgomery representation and the arguments of the hash H are also in Montgomery representation (always with respect to modulus n).

B.3 Gate Protocol

The gate protocol is taken from [1, Section 2.4.4.].

$$
\begin{array}{rl}
A \longrightarrow H: & \mathit{applet\_id}, A, S_c, S_r, w, \text{ where the } \mathit{applet\_id} \text{ is as before, and} \\
& \quad w = \beta^v \prod g_i^{\alpha_i} \text{ is the applet's witness} \\
& \quad \text{with } \beta \in \mathbb{Z}_n^*, \alpha_1, \ldots, \alpha_k \in \mathbb{Z}_v \text{ randomly chosen by the applet} \\
& \text{the host checks the signature } S_c \stackrel{?}{=} H\big(A,\ S_r^v\,(hA)^{-S_c}\big) \\
& \quad \text{and aborts the protocol if the equation does not hold} \\
H \longrightarrow A: & \gamma \in \mathbb{Z}_v, \text{ the random challenge} \\
& \text{the applet checks that indeed } \gamma < v \\
A \longrightarrow H: & r_1, \ldots r_k, s, \text{ where} \\
& \quad \begin{array}{rcl} r_i & = & (\gamma a_i + \alpha_i) \ \mathsf{mod}\ v \\ q_i & = & (\gamma a_i + \alpha_i) \div v \\ s & = & \beta b^\gamma \prod g_i^{q_i} \end{array} \\
& \text{the host accepts the proof if } s^v \prod g_i^{r_i} \ = \ A^\gamma w
\end{array}
$$

For the pure Java-Card applet the protocol is identical, except that A, S_r, w and s are transmitted in Montgomery representation and the arguments of H are also in Montgomery representation.