Developing Efficient Blinded Attribute Certificates on Smart Cards via Pairings

Lejla Batina, Jaap-Henk Hoepman, Bart Jacobs, Wojciech Mostowski, Pim Vullers

▶ To cite this version:

Lejla Batina, Jaap-Henk Hoepman, Bart Jacobs, Wojciech Mostowski, Pim Vullers. Developing Efficient Blinded Attribute Certificates on Smart Cards via Pairings. 9th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications (CARDIS), Apr 2010, Passau, Germany. pp.209-222, 10.1007/978-3-642-12510-2_15 . hal-01056105

HAL Id: hal-01056105
https://inria.hal.science/hal-01056105

Submitted on 14 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Developing Efficient Blinded Attribute Certificates on Smart Cards via Pairings

Lejla Batina[1], Jaap-Henk Hoepman[1,2], Bart Jacobs[1], Wojciech Mostowski[1⋆], and Pim Vullers[1⋆⋆]

[1] Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands.
{lejla, jhh, bart, woj, p.vullers}@cs.ru.nl
[2] TNO Information and Communication Technology, The Netherlands.
jaap-henk.hoepman@tno.nl

Abstract This paper describes an elementary protocol to prove possession of anonymous credentials together with its implementation on smart cards. The protocol uses self-blindable attribute certificates represented as points on an elliptic curve (which are stored on the card). These certificates are verified on the reader-side via a bilinear pairing. Java Card smart cards offer only very limited access to the cryptographic coprocessor. It thus requires some ingenuity to get the protocol running with reasonable speed. We realise protocol runs with on-card computation times in the order of 1.5 seconds. It should be possible to further reduce this time with extended access to the cryptographic coprocessor.

Key words: anonymous credentials, elliptic curve cryptography, smart card, bilinear pairing, attributes, blinding, protocols, Java Card

1 Introduction

With the growing use of smart cards in e-ticketing in public transport, huge centralised databases are compiled with detailed travel information of individual citizens. This raises considerable privacy and security concerns [14]. It leads to a renewed interest in anonymous credential systems, offering attribute-based authorisation. With such system a smart card may be personalised, but does not show its identity on entry into a public transport system. Instead it shows an attribute — such as "first class train pass valid in December 2009" — together with a signature on the public key of the card that is linked to this attribute. The corresponding private key is assumed to be stored in protected hardware in the card, inaccessible from the outside. We assume the attribute to be fairly general, and not identifying individual cards/people. The signature, however, is specific, and may be used for tracing. Therefore we are interested in self-blindable signatures as proposed by Verheul [25].

⋆ Sponsored by the NLnet foundation through the OV-chipkaart project.
⋆⋆ Sponsored by Trans Link Systems/Open Ticketing.

Other credential systems exist, see for instance [5,6,8]. They typically require non-trivial computational resources, such that implementing them on smart cards is a serious challenge, see for example [12,21,23]. Of these Danes [12] implements idemix [8] zero knowledge proofs on smart cards, with running times in the order of tens of seconds; Sterckx et al. [21] implement direct anonymous attestation [6] with running times under 3 seconds; Tews and Jacobs [23] describe an implementation of Brands' selective disclosure protocols [5], with running times around 5 seconds. All quoted times refer to the execution time on a smart card.

This paper distinguishes itself through its use of elliptic curve cryptography (ECC). It does so for two (main) reasons.

- ECC supports small key and certificate lengths — compared to RSA — and thus reduces the time and bandwidth needed for transfer (of public keys between card and terminal) and for blinding (via modular multiplication of big natural numbers).
- ECC supports a form of signature via bilinear pairings [20] which is stable under blinding [25].

Actual use of ECC on smart cards is relatively new. The major deployment is probably in the latest generation of European e-passports, using Extended Access Control to protect finger prints [7]. Java Card generation 2.2 smart cards offer support for ECC, through the cryptographic coprocessor, basically via only two primitives, namely Diffie-Hellman key agreement (ECDH) and Digital Signature Algorithm (ECDSA). This is barely enough to implement our protocol on a card efficiently. Crucial in our implementation are the following two points.

- We abuse ECDH to perform point multiplication (repeated addition, or integer scalar multiplication) on the card. Diffie-Hellman does such a multiplication implicitly, however, it only returns the x-coordinate of the resulting (multiplied) point. In principle, we have to reconstruct the corresponding y-coordinate (on the reader-side) by taking a square root, and checking which version (positive or negative) is the right one. However, we show how this reconstruction can be partially avoided via some tricks (see Section 4.2).
- Modular multiplication of two (big) natural numbers is not supported, that is, the Java Card API does not provide access to this operation on the card's cryptographic coprocessor. Therefore we use our own implementation in Java Card, which leads to a significant slow down. It can be mitigated by reducing the number of bits in the blinding factor, for instance from 192 to 96 (or even to 48).

The main contributions of this work are as follows.

- A new protocol for anonymous credentials is described that can be used for various applications which require smart cards, in particular e-ticketing.
- An implementation of this protocol, via several optimisations, on an ordinary Java Card, using bilinear pairings on elliptic curves on the terminal-side.
- A running time on the card in the order of 1.5 seconds.

Our results show that anonymous credentials on smart cards are becoming feasible. Also, they show that increasing access to the cryptographic coprocessor may increase the number of applications of advanced smart cards. Hopefully this perspective stimulates card manufacturers.

This paper is organised as follows. In Section 2 an overview of elliptic curve cryptography and pairings is given. These techniques are used for the protocol given in Section 3. Finally, Section 4 describes the implementation details and gives an indication of times needed for protocol runs, with a number of different parameters.

2 Elliptic Curves and Pairings

In this section we introduce some notation and we give an overview of the mathematical background of elliptic curves and pairings. The finite field containing q elements, where q is a prime power, is denoted by $\mathbb{F}_q$. An elliptic curve E over $\mathbb{F}_q$ is the set of all solutions, that is, all the points $P = (x, y)$ satisfying the following equation

$$E : y^2 = x^3 + ax + b, \qquad (1)$$

where $a, b \in \mathbb{F}_q$ and $4a^3 + 27b^2 \neq 0$, together with a special point ∞ called the point at infinity. The set of points $P \in {\mathbb{F}_q}^2$ on the curve is sometimes written as $E(\mathbb{F}_q)$.

Here, a and b are called the curve parameters. The field $\mathbb{F}_q$ is of the form $\mathbb{F}_{p^n}$ for some prime number p and $n \in \mathbb{N}$ ($p \neq 2, 3$). The solutions of equation (1) form an abelian group with point addition as the group operation and the point at infinity as the zero-element. The condition $4a^3 + 27b^2 \neq 0$ is required for E to be non-singular, as required for cryptographic applications. For cryptography we need a finite cyclic group in which the group operation is efficiently computable, but the discrete logarithm problem is very difficult to solve. Elliptic curve groups meet these criteria when the underlying field is finite and p is at least 160 bits.

We write $n \cdot P$, with scalar $n \in \mathbb{Z}$, for repeated group operation, that is, the point addition. Let E be an elliptic curve over $\mathbb{F}_q$ and let $P \in E$ be a point of order k. Let $Q \in \langle P \rangle$ be a point generated by P, that is, $Q = \alpha \cdot P$ for α where $0 \leq \alpha < k$. The problem of finding the logarithm α for given P and Q is called the elliptic curve discrete logarithm problem (ECDLP).

As mentioned above, we are using bilinear pairings on elliptic curves for our protocol. Therefore, a so-called pairing friendly elliptic curve is required, that is, a curve with a small embedding degree and large prime-order subgroup. In 2005, Barreto and Naehrig (BN) discovered a new method for constructing pairing friendly elliptic curves of prime order over a prime field [1]. More precisely, BN curves are defined over $\mathbb{F}_p$ where $p = p(u) = 36u^4 + 36u^3 + 24u^2 + 6u + 1$ for $u \in \mathbb{Z}$ such that p is prime. The order of a BN curve is a prime n where $n = n(u) = 36u^4 + 36u^3 + 18u^2 + 6u + 1$. Hence, a BN curve is constructed by generating integers u until both $p(u)$ and $n(u)$ are prime numbers. The embedding degree of BN curves is 12 and we detail the parameters for our case in Section 4.3.

2.1 DH and DSA for Elliptic Curves

The Diffie-Hellman key agreement protocol (DH) and the Digital Signature Algorithm (DSA) are easily adapted to the ECC case as in [2] and [15] respectively. We recall the protocols, which in this case are called ECDH and ECDSA.

EC Diffie-Hellman Key Agreement (ECDH) Alice (A) and Bob (B) wish to agree on a secret key over an insecure channel. They first agree on the set of domain parameters $(\mathbb{F}_q, E, n, h, G)$. Here, E is an elliptic curve over $\mathbb{F}_q$, G is a generating (publicly known) point in the elliptic curve group of order n and the integer h is called the cofactor. For the cofactor we have: $\#E(\mathbb{F}_q) = h\,n$. Due to the security of the ECDLP one usually selects a curve for which $h \leq 4$. Any random point of sufficiently high order on an elliptic curve E can be used as a key.

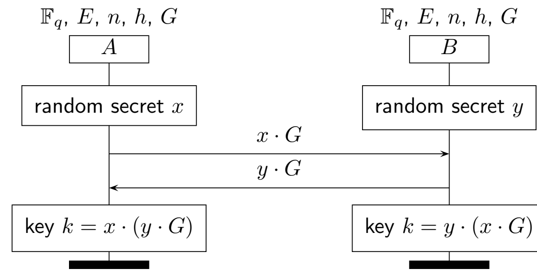

Figure 1. EC Diffie-Hellman key agreement protocol

Key Agreement Each time a shared key is required, the following steps, as depicted in Figure 1, have to be performed.

1. A chooses a random secret x, where $1 \leq x \leq n-1$, as her private key and sends B the corresponding public key $x \cdot G$
2. B chooses a random secret y, where $1 \leq y \leq n-1$, as his private key and sends A the corresponding public key $y \cdot G$.
3. B receives $x \cdot G$ and computes the shared key as $k = y \cdot (x \cdot G) = (xy) \cdot G$.
4. A receives $y \cdot G$ and computes the shared key as $k = x \cdot (y \cdot G) = (xy) \cdot G$.

So, they both end up with the same point as the common key: $k = xy \cdot G$. An adversary Eve may have knowledge of G, $x \cdot G$, and $y \cdot G$ but not of x or y. She wants to determine number $xy \cdot G$. This task is called the "(computational) Diffie-Hellman problem for elliptic curves".

EC Digital Signature Algorithm (ECDSA) The ECDSA is specified by an elliptic curve E defined over $\mathbb{F}_q$ and a publicly known point $G \in E$ of prime order n. As above, a private key of Alice is a scalar z and the corresponding public key is $Q = z \cdot G \in E$. The ECDSA requires a hash function in addition and consists of two parts as explained below.

Signature Generation In order to sign a message m, A should perform the following steps:

1. Select a random integer k, where $1 \leq k \leq n-1$.
2. Compute $(x_1,\ y_1) = k \cdot G$.
3. Compute $r = x_1 \mod n$. If $r = 0$, then go back to the step 1.
4. Compute $s = k^{-1}(h(m) + z\,r) \mod n$, where h is a hash function. If $s = 0$, then go to step 1.
5. A's signature for the message m is the pair $(r,\ s)$.

Signature Verification In order to verify A's signature on m, B performs the following steps:

1. Obtain an authenticated copy of A's public key Q.
2. Verify that r and s are integers in the interval $[1, n-1]$.
3. Compute $w = s^{-1} \mod n$ and $h(m)$.
4. Compute $u_1 = h(m)\,w \mod n$ and $u_2 = r\,w \mod n$.
5. Compute $(x_0,\ y_0) = u_1 \cdot G + u_2 \cdot Q$ and $v = x_0 \mod n$.
6. Accept the signature if and only if $v = r$.

2.2 Pairings

A bilinear pairing is a map $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ where $\mathbb{G}_1$ and $\mathbb{G}_2$ are typically additive groups and $\mathbb{G}_T$ is a multiplicative group and the map is bilinear, that is, linear in both components. Many pairings are used in cryptography such as the Tate pairing, ate pairing and the most recent R-ate pairing [24]. For all these pairings one often uses specific cyclic subgroups of $E(\mathbb{F}_{p^k})$ as $\mathbb{G}_1$ and $\mathbb{G}_2$ and $\mathbb{F}_{p^k}^*$ as $\mathbb{G}_T$.

The bilinearity property can be written as follows:

$$\begin{aligned} e(P + P',\ Q) &= e(P,\ Q) \cdot e(P',\ Q) \\ &\text{and} \\ e(P,\ Q + Q') &= e(P,\ Q) \cdot e(P,\ Q') \end{aligned}$$

As a result, $e(n \cdot P,\ m \cdot Q) = e(P,\ Q)^{n\,m}$. Pairings are used for many (new) cryptographic protocols [2], such as short signatures [4], three-party one-round key agreement [16], identity based encryption [3] and anonymous credentials [9].

Here we use pairings of the form $e\colon E(\mathbb{F}_p) \times E(\mathbb{F}_{p^k}) \to \mathbb{F}_{p^k}$, obtained by taking $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{F}_{p^k}$ and using the obvious inclusion $\mathbb{F}_p \to \mathbb{F}_{p^k}$ in the first argument. The number k is known in this context as the embedding degree. As previously mentioned, one uses $k = 12$ for BN-curves.

Pairing-based Signatures We briefly recall the pairing-based signature approach of Verheul [25], but here in the context of e-ticketing. Assume there is a system-wide public key $s \cdot Q$, for $Q \in E(\mathbb{F}_{p^k})$, with private key s under control of a scheme provider. Let ticket/card c have private key k_c, with corresponding public key $P_c = k_c \cdot P$, for some generator $P \in E(\mathbb{F}_p)$. An interesting form of signature, by the scheme provider, on such a key is simply multiplication with s. Thus, $s \cdot P_c = s \cdot (k_c \cdot P) = (s\,k_c) \cdot P = R$ can be used as signature on P_c. A pairing e can be used to check a claim that a point $R \in E(\mathbb{F}_p)$ really is a signature, namely by checking:

$$e(P_c,\ s \cdot Q) \stackrel{?}{=} e(R,\ Q).$$

Indeed, if $R = s \cdot P_c$ then both sides are equal to $e(P_c,\ Q)^s$, and thus equal. In this scenario the card terminal, and not the card, verifies the signature. Notice that the first argument of the pairing $e(-,-)$ involves a point in $E(\mathbb{F}_p)$ coming from the card. Hence the card does not have to do any of the (more complicated) work in $\mathbb{F}_{p^k}$.

A powerful aspect about these signatures is that they are invariant under blinding. Thus, if a card chooses an arbitrary number b as blinding factor, the resulting pair $(b \cdot P_c,\ b \cdot (s \cdot P_c))$ is again a signature, for the private-public key pair $(b\,k_c,\ b \cdot P_c = (b\,k_c) \cdot P)$. Each time the card is used it can thus present a different (signed) public key, such that the different uses cannot be linked.

Of course, when a card presents a reader with such a pair P_c, R the reader should not only check that R is a proper signature on P_c, that is, R is $s \cdot P_c$, but also that the card knows the private key corresponding to the public key P_c. This can be done via standard challenge-response exchange, for example using ECDSA.

2.3 Elliptic Curves on Smart Cards

The on-card part of our protocol is running on a Java Card smart card [11]. Java Card technology gives us an open environment where we can easily implement our protocols using Java and load them onto a development smart card. A Java Card may be equipped with a cryptographic coprocessor to support a selection of cryptographic algorithms defined by the official Java Card API [22], and possibly some proprietary extensions provided by the card manufacturer [18]. Since version 2.2 the Java Card API offers support for EC based algorithms. When it comes to the implementation of our protocol on a Java Card "EC support" alone is not enough as there are some issues that we need to be aware of, as follows.

Java Card is an embedded and closed device. Internally its cryptographic coprocessor implements all the routines required for a given cryptographic protocol (for example ECDH), but externally we can only utilise what is exported through the API. For example, we may be able to perform the ECDH key agreement protocol on the card (and hence scalar point multiplication), but we do not have access to the point addition or point doubling primitives. The only

other EC based algorithms that the card can support are EC key generation and ECDSA signature generation and verification. Extension of the algorithms that the card provides is practically impossible. The cryptographic coprocessor is not accessible through Java code, and implementing EC operations in Java (byte)code hinders performance significantly [23]. As long as we stick to the built-in cryptographic operations the performance is acceptable. For example, a single point multiplication for 192 bit keys/points (using the ECDH operation) takes about 0.10 seconds, ECDSA signature generation about 0.12 seconds, and EC key pair generation about 0.60 seconds. A detailed method to measure Java Card performance and some performance results can be found in [19]. Furthermore, the card may or may not support the compressed point format (and hence point reconstruction) as input for EC based algorithms.

Due to this limited environment and performance requirements we need to *balance* our protocol such that the operations required on the card are minimised and match the cards capabilities. The computationally expensive operations, like pairing, should be performed by the terminal. Ideally, the card should only be asked to perform operations supported natively by the Java Card cryptographic API.

Finally, as all cryptographic support is optional in Java Card, we need a development card that actually does support the EC based algorithms. We used the NXP JCOP31 2.4.1 Java Card based on the SmartMX platform. It supports ECDH and ECDSA with key sizes up to 320 bits, but it does not support the compressed point format.

3 Protocol for Attribute-proving

This section describes an elementary protocol of how a card c demonstrates in a secure and privacy friendly manner that it possesses some attribute(s) a. This attribute is just a number, with certain meaning that we abstract away.

3.1 The Protocol

System Setup The scheme provider has a public fixed point $Q \in E(\mathbb{F}_{p^k})$ and a finite set of attributes. For each attribute a a secret key s_a, which is a number below the order of Q, and public key $Q_a = s_a \cdot Q$ are generated. The associated pairs (a, Q_a) of attributes and public keys are publicly known, and stored in all terminals together with the fixed point Q. This use of different signing keys for different attributes goes back to work of Chaum on e-cash [10], where for instance an e-coin of 50 cents had a different ("50 cent") signature than an e-coin with a value of 100 cents.

A card c generates a key pair $k_c, P_c = k_c \cdot P$ where $P \in E(\mathbb{F}_p)$ is a fixed system wide generator. The private key k_c of the card is assumed to be stored in a protected manner such that it cannot leave the card. Upon personalisation it receives an attribute together with a certificate $C_a = s_a \cdot P_c$ linking its public

key P_c to the attribute a. The attribute a corresponds to a product that the owner of the card has bought.

Before the protocol, as described below, will be run, some form of authentication between the card and terminal should be performed. This is required to protect the card from proving the possession of an attribute to a rogue reader. After this initial authentication phase the card can be sure that it communicates with a genuine reader and can proceed with the execution of the protocol. This preceding step will be ignored here.

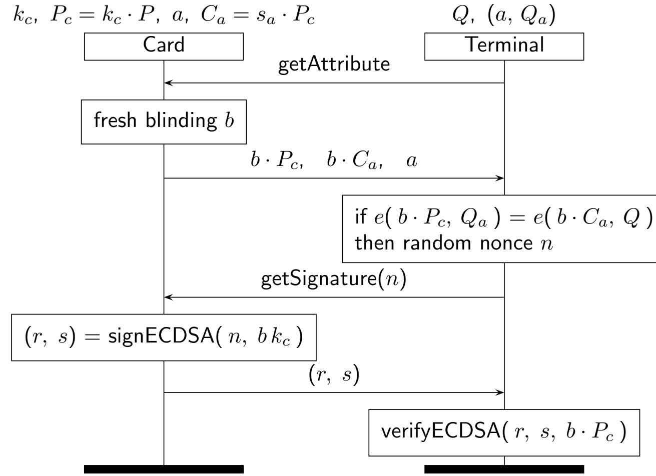

Figure 2. Protocol for proving self-blindable attributes

Protocol Description The protocol for proving self-blindable attributes, as depicted in Figure 2, is initiated by a terminal which requests an attribute from the card. The card generates a fresh blinding factor b to blind its public key and the certificate. It responds by sending the blinded values ($b \cdot P_c$ and $b \cdot C_a$) together with the attribute stored on the card.

The terminal can now perform a pairing signature verification, as discussed in Section 2.2, using the card's response, the attribute's public key Q_a and the fixed point Q. Note that the terminal can select the correct public key Q_a to use by matching the attribute returned by the card. If the verification succeeds the terminal generates a random nonce n which is used to challenge the card, that is, to request a signature over n. The card responds with an ECDSA signature $(r,\ s)$ of this nonce, which is created using its blinded private key $b\,k_c$.

Finally the terminal verifies the ECDSA signature using the blinded public key. This works since $b\,k_c$ together with $b \cdot P_c = (b\,k_c) \cdot P$ is a valid key pair. When the verification succeeds the card has proved possession of the requested attribute.

3.2 Some Issues

Privacy In this protocol the attribute itself is not hidden, and may thus be used to trace cards/people. This can be overcome to some extent by using only a limited number of fairly general attributes. For instance, as possible validity date one can choose to use only the first day of each month, and not every possible day of the year. Alternatively, one may provide a card with several attributes, stating for instance: "year card valid in January 2009", "year card valid in February 2009", and so on. In this way each card can present the appropriate attribute, after receiving the current date from the terminal.

Efficiency A drawback of the current protocol is that it only proves a single attribute to the terminal. Consider the situation in which a card holds more than one attribute. The terminal could then perform multiple requests to the card in which an index indicates which attribute is requested. This is, however, not a very efficient solution since the required amount of time for proving grows linearly with the amount of attributes to be proved.

One approach could be to combine all attributes into a single point on the curve, requiring only a single protocol run to proof all attributes. However, such method could lead to new privacy issues since all attributes are disclosed at once. Therefore selective disclosure of these attributes, as proposed by Brands [5], would be preferred. The efficient adaption of these techniques to the elliptic curve setting is a subject for further research.

Revocation Within various settings, for example e-ticketing for public transport, it is highly desirable to be able to revoke cards, for instance after abuse or after early termination of a contract. However, revocation is non trivial as terminals only get to verify blinded keys and blinded certificates. This blinding makes straightforward black listing impossible. Kiyomoto and Tanaka [17] proposed a solution to this problem that is unfortunately broken. The essence of their idea is that the private key consists of two parts, the second part encoding some personal identifiable information. This second subkey is put on the blacklist, and the protocol checks, in a blinded fashion, whether the subkey occurs on the blacklist. This approach does not work because the user, after getting revoked, can choose new subkeys that, when combined, still match with the original certificate, but where the second subkey is different and no longer occurs on the blacklist.

The solution to this problem can be obtained as follows (to be detailed in a forthcoming paper). The crucial observation is that even though $K_i = k_i \cdot P$ is *called* a public key, there is no reason to actually *make* it public: the proof that you have a certificate for a certain attribute always blinds both the public key and the certificate. So let us assume we keep the public key and certificate *secret*, and only publish it when we want to revoke it. This way, users whose public key is unknown cannot be traced because their key cannot be guessed, breaking out of the assumed paradox.

4 Implementation and Performance Indicators

To understand practical limitations and estimate performance of our protocols, we implemented the protocol from Figure 2 in Java (terminal-side application) and Java Card (card-side application). Our implementation involves the following components:

Bouncy Castle Library with Extension for Pairings The Bouncy Castle (BC) library[3] is a collection of cryptographic APIs for Java and C# programming languages. Bouncy Castle provides full support for ECC and an interface to the common Java Cryptography Extension API. However, it does not implement pairings or elliptic curves over fields other than $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$. Thus we have added our own implementations of $\mathbb{F}_{p^2}$ and $\mathbb{F}_{p^{12}}$, and the Tate, ate, and R-ate pairings. To minimise maintenance overhead we strived to keep our extensions purely *on top* of the Bouncy Castle library, that is, we did not change anything in the original library. In future we plan to contact the BC development team to incorporate our extensions into the official BC tree.

Smart Card IO Library Since version 6.0 the standard Java Development Kit includes support for communication with smart cards by providing the `javax.smartcardio` package. We used it to talk to a Java Card smart card on which our client applet was installed.

A Java Card with the Client Applet The protocol on the card-side is implemented as a Java Card [11] applet and loaded onto a development Java Card.

4.1 Java Card Applet

In Section 2.3 we described the practical limitations of Java Cards that we have to take into account while programming the card. The actual operations that the card needs to perform are scalar multiplication of points and modulo multiplication of natural numbers. In the end the applet performs the required steps of the protocol in the following way.

An on-card random number generator is used to generate the blinding value. This value is stored in an EC private key structure on the card, which we call the *blind* (not blinded) key. Then two ECDH key agreement operations (effectively two scalar point multiplications) are performed with this blind key to calculate the blinded public key, and the blinded certificate, both of which are EC points. As mentioned in the introduction, the ECDH implementation only returns the x-coordinate of these multiplications, forcing the terminal to reconstruct the y-coordinate (see below).

The second part of the protocol requires the card to sign a nonce with the ECDSA algorithm using a blinded private key $b \cdot k_C$. The modular multiplication of these two large natural numbers has to be performed using hand implemented

[3] `http://www.bouncycastle.org`

Java code. That is, for this last step we cannot utilise any of the Java Card API routines, like the key agreement above, to make the coprocessor do the multiplication with high performance. This single modular multiplication is the main bottleneck on the card-side implementation (see Section 4.5).

4.2 Terminal Application

The terminal application needs to cope with the shortcomings of the Java Card applet. This comes down to the fact that the terminal has to reconstruct the points received from the card, $b \cdot P_c$ and $b \cdot C_a$, before they can be processed any further.

If we know the x-coordinate of a point on the curve, the square of the corresponding y-coordinate is known, namely as $y^2 = x^3 + ax + b$. By taking the square root of $x^3 + ax + b$ we find either y or $-y$. This forms the basis of "point compression", for compact representation of points. This is important for the implementation, because Diffie-Hellman on a Java smart card only produces the x-coordinate of a multiplication, as mentioned in Section 4.1.

This reconstruction is a simple guess work, trying different signs for the two y-coordinates. For the ECDSA signature verification this is not a real issue since this verification is reasonably fast, although this is of course not optimal. For the pairing signature verification simple guessing is not desirable. Therefore we exploit the bilinearity of the pairing to avoid computing more than two pairings, as would be the case without point reconstruction.

First we calculate $e_1 = e(b \cdot P_c, Q_a)$ and $e_2 = e(b \cdot C_a, Q)$ where we take any sign for the y-coordinate of $b \cdot C_a$. If $e_1 = e_2$, which happens if we have two right, or two wrong, signs in the first parameters of the pairing, the verification succeeds. In the remaining case, which means we took one right and one wrong sign, we check whether $e_1 \, e_2 = 1$ holds. If it holds, the verification also succeeds. This is true because of the following. If $e_1 \neq e_2$, the error is caused by the wrong sign resulting in one pairing being the inverse of the other, that is, $e_2 = e_1^{-1}$. Here we can use that $e_1 \, e_2 = e_1(e_1^{-1}) = 1$ to avoid an extra pairing calculation for the negated point of $b \cdot C_a$.

4.3 System Parameters

For our test we selected three BN curves for keys of length 128, 160 and 192 bits. The domain parameters p and n are generated by the BN indices $u = 1678770247$, $u = 448873116367$ and $u = 105553250485267$ respectively (see Section 2). The curve E is defined as $y^2 = x^3 + 3$, that is, take $a = 0$ and $b = 3$ in the general form $y^2 = x^3 + ax + b$, with the default generator $G = (1, 2)$.

These key lengths have been chosen to indicate performance for various levels of security, that is, protection against fraud. A key length of 128 bits provides borderline security, whereas 160 and 192 bits provide, respectively, a minimal and a standard level of security [13].

The length of the blinding factor generated by the card is either equal to the chosen key length or a half or a quarter of this length. Reducing this length

of the blinding factor is a way to partially compensate for the slowness of the modular multiplication (in Java Card) of big numbers. This way a trade off can be made between privacy and performance.

4.4 Test Results

The results of our tests are summarised in Table 1. These values are the average of ten test runs for each configuration, that is, for each combination of key and blinding length.

Table 1. Test results for various key and blinding lengths

key (bits)	blinding (bits)	getAttribute (ms)	getSignature (ms)	card total (ms)	verification (ms)	protocol (ms)
192	192	545	2202	2748	143	2891
	96	543	1340	1884	136	2020
	48	544	907	1451	130	1582
160	160	442	1417	1860	126	1987
	80	443	912	1355	133	1489
	40	442	670	1113	127	1240
128	128	364	1235	1599	91	1691
	64	363	780	1143	93	1237
	32	362	565	927	86	1014

The table shows the duration (in milliseconds) of the getAttribute and getSignature requests to the card. The total amount of time spent by the card, which is the sum of the durations of the requests, is shown in the 'card total' column. For the terminal we measured the duration of the signature verifications, summarised in the 'verification' column. Finally the total duration of the protocol execution is shown in the 'protocol' column.

4.5 Analysis

We first look at the part of the protocol executed on the smart card. From the results given in Table 1 it can be seen that the duration of the getAttribute request depends only on the length of the key. This is in strong contrast with the getSignature request which also depends heavily on the blinding size.

This contrast can be explained by the available support from the cryptographic coprocessor. For the blinding in the getAttribute request the applet uses the ECDH primitive provided by the coprocessor to perform the required two point multiplications. The blinding in the getSignature request, which requires

a modular multiplication, has to be calculated *without* the help of the coprocessor. In theory it is possible to abuse the RSA cipher (and hence use the coprocessor) to do large part of the modulo multiplication by using the fact that $4ab = (a+b)^2 - (a-b)^2$, as in [21,23]. The squares in this equation can be performed by doing an RSA encryption/exponentiation with a suitable RSA public key, that is, one with the exponent 2 and the required modulus. The numbers $a+b$ and $a-b$ are then just messages to be encrypted using the RSA cipher, which is provided by the Java Card API.

We tried this approach, but with no success. The main obstacle is that the RSA cipher on the card operates only within valid bit lengths for RSA keys, starting with 512 bit keys. Although the number to be multiplied (the message) can be any value, the number of non-zero bits in the modulus has to be at least 488 bits for 512 bit keys according to our tests. Since our modulus is only 192 bit long the card refused to perform RSA encryption with such short modulus value. However, we believe that a more flexible RSA implementation on the card would allow this optimisation.

The performance of the terminal application is good, taking less than a tenth of running time on the smart card. The drawback on this side is caused by the use of the ECDH primitive to calculate the blinded value. This results in the problem that the card only responds with the x-coordinates of the blinded values. Therefore the terminal has to reconstruct the actual point from these values as mentioned above. If the card could respond with the actual points, either in compressed or uncompressed format, instead of just the x-coordinate, the duration of the verification phase could be shortened.

A large benefit of our use of ECC is the small amount of data that needs to be exchanged between the terminal and the card. For key lengths of 192, 160 and 128 bits the total amount of bytes exchanged is 168, 152 and 136 respectively. This would allow an implementation to use a *single* APDU pair (command and response) for all communication. This is in strong contrast with RSA-based protocols [21,23] which already require multiple APDUs to transfer a single command.

5 Conclusions

In line with results of others [12,21,23] this paper demonstrates that implementations of anonymous credential systems with smart cards are becoming possible. One important bottleneck (and source of frustration) remains the limited access offered to the coprocessor on current Java Card smart cards. This paper uses elliptic curves (with pairings) and abuses Diffie-Hellman key agreement for (scalar) point multiplication, together with several other tricks, to bring on-card computation times down to around 1.5 seconds. We expect to be able to further reduce this time via additional optimisations.

References

1. Barreto, P., Naehrig, M.: Pairing-friendly elliptic curves of prime order. In: Selected Areas in Cryptography - SAC 2005. LNCS, vol. 3897, pp. 319–331. Springer (2006)
2. Blake, I., Seroussi, G., Smart, N.P.: Advances in Elliptic Curve Cryptography. No. 317 in LMS, Cambridge Univ. Press (2005)
3. Boneh, D., Franklin, M.: Identity-based encryption from the Weil pairing. In: Advances in Cryptology - CRYPTO 2001. LNCS, vol. 2139, pp. 213–229. Springer (2001)
4. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil pairing. Journal of Cryptology 17(4), 297–319 (2004)
5. Brands, S.: Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy. MIT (2000)
6. Brickell, E.F., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Pfitzmann, B., Liu, P. (eds.) Computer and Communications Security - CCS 2004. pp. 132–145. ACM (2004)
7. BSI: Advanced security mechanisms for machine readable travel documents – Extended Access Control (EAC). Tech. Rep. TR-03110, German Federal Office for Information Security (BSI) (2008)
8. Camenisch, J., van Herreweghen, E.: Design and implementation of the idemix anonymous credential system. In: Computer and Communications Security - CCS 2002. pp. 21–30. ACM (2002)
9. Camenisch, J., Lysyanskaya, A.: Signature schemes and anonymous credentials from bilinear maps. In: Advances in Cryptology - CRYPTO 2004. LNCS, vol. 3152, pp. 56–72 (2004)
10. Chaum, D.: Blind signatures for untraceable payments. In: Chaum, D., Rivest, R.L., Sherman, A.T. (eds.) Advances in Cryptology - CRYPTO 1982. pp. 199–203. Plenum Press (1983)
11. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Java Series, Addison-Wesley (2000)
12. Danes, L.: Smart card integration in the pseudonym system idemix. Master's thesis, University of Groningen, the Netherlands (2007)
13. ECRYPTII: Yearly report on algorithms and keysizes (2008-2009). Tech. Rep. D.SPA.7, European Network of Excellence in Cryptology II (ECRYPTII) (2009)
14. Jacobs, B.: Architecture is politics: Security and privacy issues in transport and beyond. In: Gutwirth, S., Poullet, Y., Hert, P. (eds.) Data Protection in a Profiled World - CPDP 2008. Springer (2010)
15. Johnson, D., Menezes, A.: The elliptic curve digital signature algorithm (ECDSA). Tech. Rep. CORR 99-34, Department of Combinatorics & Optimization, University of Waterloo, Canada (2000)
16. Joux, A.: A one round protocol for tripartite Diffie-Hellman. Journal of Cryptology 17(4), 263–276 (2004)
17. Kiyomoto, S., Tanaka, T.: Anonymous attribute authentication scheme using self-blindable certificates. In: Intelligence and Security Informatics - ISI 2008. pp. 215–217. IEEE (2008)
18. NXP: Smart solutions for smart services (z-card 2009). NXP Literature, Document 75016728 (2009)
19. Paradinas, P., Cordry, J., Bouzefrane, S.: Performance evaluation of Java Card bytecodes. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.J. (eds.) Information Security Theory and Practices - WISTP 2007. LNCS, vol. 4462, pp. 127–137. Springer (2007)

20. Smart, N.: Elliptic curve based protocols. In: Blake, I., Seroussi, G., Smart, N. (eds.) Advances in Elliptic Curve Cryptography. pp. 3–19. No. 317 in LMS, Cambridge Univ. Press (2005)
21. Sterckx, M., Gierlichs, B., Preneel, B., Verbauwhede, I.: Efficient implementation of anonymous credentials on Java Card smart cards. In: Information Forensics and Security – WIFS 2009. pp. 106–110. IEEE (2009)
22. Sun Microsystems, Inc.: Java Card 2.2.2 Application Programming Interface Specification (2006)
23. Tews, H., Jacobs, B.: Performance issues of selective disclosure and blinded issuing protocols on Java Card. In: Markowitch, O., Bilas, A., Hoepman, J.H., Mitchell, C., Quisquater, J.J. (eds.) Information Security Theory and Practice - WISTP 2009. LNCS, vol. 5746, pp. 95–111. Springer (2009)
24. Vercauteren, F.: Pairings on elliptic curves. In: Joye, M., Neven, G. (eds.) Identity-Based Cryptography. CIS, vol. 2, pp. 13–30. IOS Press (2009)
25. Verheul, E.: Self-blindable credential certificates from the Weil pairing. In: Boyd, C. (ed.) Advances in Cryptology - ASIACRYPT 2001. LNCS, vol. 2248, pp. 533–550. Springer (2001)