Implementing Elliptic Curve Cryptography on PC and Smart Card*

István Zsolt BERTA – Zoltán Ádám MANN

Abstract

Elliptic Curve Cryptography (ECC) is a relatively new branch of public key cryptography. Its main advantage is that it can provide the same level of security as RSA with significantly shorter keys, which is beneficial for a smart card based implementation. It is also important as a possible alternative of RSA. This paper presents the authors' research concerning ECC and smart cards.

The authors introduce their ECC prototype implementation, that relies on Java Card technology and is capable of running on smart cards. Test results with various cards are attached. It is also analyzed, that in what extent can algorithms with the complexity of ECC be executed in smart card environment with limited resources.

1 Introduction

Mathematic research on elliptic curves has a long tradition. However, this field attracted the attention of cryptographers only in the 1980s, when it was discovered that elliptic curves can be used to define finite groups in which the discrete logarithm problem is hard [19, 17]. One and a half decades have passed and experience confirmed this assumption. But since no theoretic proof is known, it is an open question whether elliptic curve cryptography (ECC) is mature enough for industrial use. Therefore it is vital to gain as much practical experience as possible about the security of ECC. The authors, too, hope to contribute to this research.

One has to be quick about it though. Basically, there are two factors to worry about. The first problem is that actually only one public-key algorithm is widely used, namely RSA. No theoretic guarantee is known as to the security of RSA either, however, the whole PKI (Public Key Infrastructure) is based on RSA. If one an effective crack for RSA if found - which is not impossible -, the impact on today's cryptographic systems would be dramatic. This explains why other public-key methods would be welcome. ECC is a probable candidate.

The other factor is the enormous growth in computation capacity that is available for cracking. This is due to the wide spread of inexpensive and powerful workstations. Also, these computers have almost always network connectivity. As a consequence, the most spectacular cracks of the last years were not done by supercomputers but by thousands of cooperating workstations [6].

Another aspect in cryptography that has ever growing importance is the need for algorithms with low resource requirements so that they can run on smart cards. A typical smart card has a processor with clock frequency of about 3-5 MHz and about 4-32 KByte read-writable memory. This constrains the use of public-key algorithms in smart cards heavily.

Elliptic Curve Cryptography might solve this problem, since - according to our current understanding - it provides the same level of security with significantly smaller keys than RSA. This is caused by the fact that no sub-exponential time algorithm is known to solve the basis of ECC, the Elliptic Curve Discrete Logarithm Problem (ECDLP), while there are some sub-exponential,

*This paper has been published in *Periodica Polytechnica, Series Electrical Engineering*, volume 46, number 1-2, pp. 47-73, 2002.

although not polynomial algorithms to crack RSA. This also implies that the difference in key size between the two algorithms will constantly increase as the augmenting computational power will require higher levels of security. The relatively small keys of ECC open new possibilities for ECC in areas like smart card and mobile commerce applications. However, the complicated mathematical background of ECC results in more sophisticated algorithms so it is by no means obvious that the required computational power would be smaller.

Despite these possibilities, there are hardly any practical implementations of ECC for low-resource devices. The authors know of one ECC software implementation for smart cards [10]. However, this implementation can only deal with very limited key sizes (50 bit), which have no cryptographic significance.

This paper presents the authors' ECC prototype implementation on a smart card platform, which can also handle ECC problems of cryptographically meaningful sizes. First, an overview is given on the underlying mathematics and the theory of ECC. Next, the smart card environment is presented that was used by the authors to implement ECC. Test results are presented for three different smart cards and for PC as well. At the end of the paper, some possibilities for improvement are discussed.

2 Theoretical background

In this section a short overview is given about the basic results that build the fundament for elliptic curve cryptography. For the proofs of the referred theorems as well as for further information on elliptic curves see *e. g.* [13].

2.1 Elliptic curves

An elliptic curve is defined in the general case by an equation of the form:

$$y^2 + axy + by = x^3 + cx^2 + dx + e \tag{1}$$

The coefficients a, b, c, d and e as well as the values of the variables x and y are elements of a given field K. The solutions of (1) are the points of the elliptic curve E over the field K; their set is denoted by $E(K)$. First the case $K = \mathbb{R}$ (real elliptic curves) will be discussed. Although the complex and rational cases ($K = \mathbb{C}$ and $K = \mathbb{Q}$, respectively) also have theoretical significance, they will not be covered here, emphasis will be laid on the cryptographically more interesting case when K is a finite field.

2.1.1 Elliptic curves over $\mathbb{R}$

In this case the terms axy and by on the left side of equation (1) can be eliminated using linear substitution:

$$y^2 = f(x) \tag{2}$$

where $f(x)$ is a real polynom of order 3. It can even be assumed that

$$f(x) = x^3 + ax + b$$

It is usually also required that f have no multiple roots. This actually means that the function $F(x, y)$ defining the elliptic curve with $F(x, y) = 0$ be smooth. Such a curve can be seen in figure 1.

The cryptographic applicability of elliptic curves is a consequence of the group structure that can be defined on them. For this, one has to define an operation between the points of the curve. The operation will be called addition since this is the usual notation for Abelian groups. Now suppose two points P and Q are given on the elliptic curve. In order to compute $P + Q$, one has to do the following. First, draw a straight line through P and Q. This line intersects the curve in

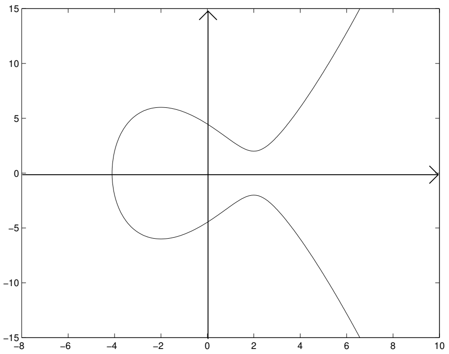

Figure 1: An elliptic curve

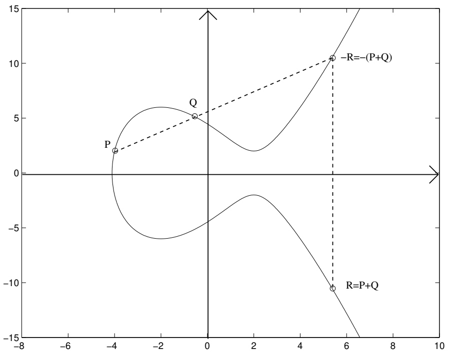

Figure 2: Adding two points of a curve

exactly one more point which will be denoted by $-R$. Then mirror $-R$ on the x axis; the resulting point is again on the curve, denote it by R. Let $P+Q=R$. Figure 2 shows an example.

Unfortunately, there are some problems with this definition. What happens if the line that connects P and Q is vertical and thus it has no third intersection with the curve? (This is only possible if the coordinates of the two points are $(x;y)$ and $(x;-y)$.) In order to correct the definition, the set $E(K)$ has to be extended with a new point, denoted by O, which can be visualized as the point at infinity in the direction of the y axis. As it turns out, this point will be the neutral element of the group.

In order to define the point O precisely, projective coordinates have to be introduced. The points of the projective plane are the equivalence classes of real (x,y,z) tuples, where $(x,y,z) \neq (0,0,0)$, the equivalence relation being $(x,y,z) \sim (\lambda x, \lambda y, \lambda z)$, $\forall \lambda \neq 0$. If $z \neq 0$, then the projective point $(\frac{x}{z}, \frac{y}{z}, 1)$ representing the class (x,y,z) corresponds to the 'normal' (affine) point $(\frac{x}{z}, \frac{y}{z})$, so the affine plane can be embedded into the projective plane. But there are other points in the projective plane, namely with $z=0$. These points can be regarded as the points at infinity of the affine plane; the projective point $(x,y,0)$ is the point at infinity in the direction of the vector (x,y), or equivalently, $(-x,-y)$ [9].

To obtain the equation of the elliptic curve on the projective plane, one has to substitute $\frac{x}{z}$ in place of x and $\frac{y}{z}$ in place of y. After multiplication with z^3, the following equation is obtained:

$$y^2 z = x^3 + axz^2 + bz^3$$

Substituting $z=0$, the equation yields indeed the point at infinity in the direction of the y axis: $(0,1,0)$. So the inclusion of the point O is natural, since the curve really contains such a point in the projective plane.

Getting back to the definition of addition, the question of what to do if the line connecting the two points P and Q is vertical, now can be answered. Namely, in this case O is the third intersection of the line and the curve. And, if O is mirrored on the x axis, we get again O. So in this case $P+Q=O$. As already mentioned, O is the neutral element of the group, so in this case P and Q are inverse to each other. This is why we used previously the notion $-R$ and R when describing the addition rule: those two points are indeed inverse to each other.

It has to be noted as well that a tangent counts as if it had a double intersection with the curve. So, for example, if $P=Q$, then 'the line connecting them' is the tangent to the curve. Similarly, if the line connecting P and Q is tangent to the curve in, say, P, then P is also the third intersection of the line and the curve.

$P+Q$ can also be defined in an algebraic way, with a formula. Let the two points be $P(x_1, y_1)$ and $Q(x_2, y_2)$. Then the coordinates of $P+Q$ are:

$$\begin{aligned} x_3 &= s^2 - x_1 - x_2 \\ y_3 &= s(x_1 - x_3) - y_1 \end{aligned} \tag{3}$$

where

$$s = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P = Q \end{cases}$$

is the slant of the line connecting the two points. It can be seen from this formula, that in the case where the x coordinates of P and Q are the same, one has to divide by zero. This corresponds to the fact that the result is the point at infinity O.

The existence of this formula is very important. First of all, we will not have to base our cryptographic system on geometric insight only, but rather on precise algebraic definitions. This is going to be vital in the case of finite fields where no such geometric view is known. Second, since it is possible to derive a formula for the sum of the two points, this proves that addition is well-defined. Third, since only addition and multiplication was used between field elements, this means that if the coordinates of P and Q and the parameter a of the curve are elements of the field K, then so are the coordinates of $P+Q$.

We have not proven that the resulting structure is an Abelian group. What we have seen is the closure, the neutral element and the inverse. The Abelian property is trivial. The only thing that should be proven is associativity. However, this is rather complicated, so we just refer to the literature [13].

2.1.2 Elliptic curves over finite fields

From the point of view of cryptographic applications, infinite fields are of little interest because of rounding and inaccuracy problems. That is why it is a good idea to use finite fields. In the following, the finite field is denoted by $GF(q)$, where the number of elements is $q = p^r$. Thus the characteristics of the field is p. Of special interest are the cases $r = 1$, *i. e.* $GF(p)$ and $p = 2$, *i. e.* $GF(2^r)$.

The method that was used to simplify equation (1) can only be used if the characteristics of the field is neither 2 nor 3. The case $p = 2$ is the most complicated: here two sub-cases exist, called supersingular and non-supersingular cases. The simplest form of (1) is as follows:

$p = 2$, supersingular case:

$$y^2 + ay = x^3 + bx + c$$

$p = 2$, non-supersingular case:

$$y^2 + xy = x^3 + ax^2 + b$$

$p = 3$:

$$y^2 = x^3 + ax^2 + bx + c$$

$p > 3$:

$$y^2 = x^3 + ax + b$$

Since the formulas defining the addition were also derived from equation (2), these are also invalid for the cases $p = 2$ and $p = 3$. So similar formulas have to be derived from the more general equation (1). These are more complicated so we just note one difference: since the curve is now rotated and translated, the inverse of the point (x, y) is not $(x, -y)$ but – using the notations of equation (1) – $(x, -ax - b - y)$.

In the case of finite fields the number of the points of the curve is also finite; it is usually denoted by N. The curve again has one (or more) point(s) at infinity. The number of non-infinite (that is, affine) points is of course at most q^2, but it is actually less. Since the equation of the curve is an equation of degree 2 in the variable y for every possible value of x, and x can have q different values, $N \leq 2q$. Actually N is only about q. Namely, by a theorem of Hasse (1934), the difference of N and $q + 1$ is less that $2\sqrt{q}$.

This makes it possible to find random points on an elliptic curve. Choose x randomly; there is a chance of about 0.5 that the equation is solvable for that x, yielding a point of the curve. However, no *deterministic* polynomial-time algorithm is known for finding points on a given elliptic curve. Actually, there are algorithms that are deterministic polynomial-time bound if the generalized Riemann-hypothesis is true.

There is indeed a deep connection between these results and the Riemann-hypothesis. The curve $E(GF(q))$ may also be considered over $GF(q^s)$. In this case, the number of points is denoted by N_s (so $N_1 = N$). From these numbers a formal power series can be constructed (zeta-function):

$$Z(E(GF(q)), T) = e^{\sum N_s T^s / s}$$

Due to another theorem of Hasse, this is a rational function:

$$Z(E(GF(q)), T) = \frac{1 - AT + qT^2}{(1 - T)(1 - qT)}$$

where only A depends on the actual curve. Moreover, $N = q + 1 - A$, so A can be calculated from N. As a consequence, every N_s is determined by N.

2.2 Cryptographic application of elliptic curves

2.2.1 ECDLP

The cryptographic application of the so-called Elliptic Curve Discrete Logarithm Problem (ECDLP) was suggested independently by V. Miller and N. Koblitz in 1985 [19, 17].

An elliptic curve E is given over the finite field $GF(q)$, as well as points P and Q on E. Let n be the order of P, *i. e.* the smallest positive integer such that $nP = O$. (nP means: the point P added to itself n times.) The task is to find an integer $0 < k < n$ such that $Q = kP$, provided that such a k exists.

This is really the discrete logarithm problem (DLP), only the notation is different. Namely, if the operation between the points of the curve were called multiplication instead of addition, then we would write $Q = P^k$ instead of $Q = kP$, so the task would be to find $k = \log_P Q$. The reason why additive notation is used is that this is a convention for Abelian groups.

Of course it is more convenient to use the modulo classes for calculations than points of an elliptic curve. So why would we bother with elliptic curves if we get the same discrete logarithm problem? The answer is that although the DLP may be defined of course for arbitrary groups, its hardness varies significantly. Elliptic curves seem to provide a group structure where the DLP is really hard: no algorithm is known that could solve the general ECDLP in sub-exponential time.

Nevertheless, there are some sub-exponential algorithms for specific special cases of the ECDLP. Therefore the elliptic curve E and the base point P have to be chosen carefully. For instance, Menezes, Okamoto and Vanstone gave in 1993 a relatively efficient algorithm for the ECDLP on supersingular curves. But most elliptic curves are not supersingular so it is still an open question whether there is a sub-exponential algorithm for the general case. Currently, no such algorithm is known so cryptographic systems based on ECC provide a high level of security with relatively small key sizes. However, as we will see, the complicated calculations make ECC somewhat less effective.

Note that elliptic curves are not the only mean to create groups where the DLP is hard. In fact, the elliptic curve is just the first member of a bigger family of groups, defined as a group structure on the Jacobi-surface of specific curves. The next one is called hyperelliptic. But it seems that these constructions do not add as much in security as in complexity.

2.2.2 The underlying finite field

From the implementation point of view, the choice of the underlying finite field is crucial. It is almost always $GF(p)$ or $GF(2^r)$ (for some prime p or some integer r, respectively) that is chosen. The former case is more straight-forward to handle: it is the usual arithmetic that can be used, the only difference is that numbers bigger than $p-1$ must be cut back in the interval $[0,\dots,p-1]$. The latter case requires more sophisticated algorithms, however, it fits better to the binary arithmetic supported by most computer systems, therefore it can provide better performance.

There are a few different ways to represent the elements of $GF(2^r)$. The most widely used is the polynom-based representation. For this, an irreducible polynom of degree r is chosen over $GF(2)$ first; let this be denoted by f. Now the elements of $GF(2^r)$ can be regarded as polynoms of degree at most $r-1$ over $GF(2)$. Addition is done as usual; the only difference is that because of being in characteristics 2, any number (or polynom) added to itself is zero. This also implies that subtraction is the same as addition. In order to multiply two elements of the field, one has to multiply them with polynom multiplication modulo f. That is, the product of the two polynoms is divided through f, and the remainder – which is again a polynom of degree at most $r-1$ – is the product of the two field elements. It can be proven that this construction gives indeed a field and of course it has 2^r elements. (Note that a polynom is often represented as a 0-1 vector of length r of its coefficients.)

Another approach is to regard $GF(2^r)$ as an extension field of $GF(2)$ and thus as a vector space over it. Again, the elements of the field are 0-1 vectors of length r but this time the vectors have different semantics. Incidentally, addition is done in the same way as with polynoms, *i. e.* component-wise. However, multiplication is different: it must be defined on the r base elements of the vector space. This requires $r \times r \times r$ coefficients, where coefficient (i, j, k) specifies the k. coordinate of the product of the i. and j. base vector.

Multiplication can be speeded up significantly if this 'cube' is sparse, *i. e.* it contains few 1-s. In the case of a so-called *optimal normal base* there are only about $2r^2$ 1-s (out of the possible r^3). There are some constructions for optimal or nearly optimal normal bases [21]; unfortunately they work for special values of r only and those special values would degrade security.

2.2.3 Some protocols based on ECDLP

Most traditional public-key protocols have an equivalent based on elliptic curves. As a last station of our theoretic excursion, let us see some of them.

As a first example, the Diffie-Hellman key exchange protocol will be examined. As usual, E is an elliptic curve over $GF(q)$ and P is a point on E. E and P are publicly known. Each user chooses a secret key k which is a positive integer, and compute their public key kP. Suppose that Alice and Bob would like to exchange secret messages using some symmetric cryptographic scheme and need a key that is known to both of them but to nobody else. If their secret keys are k_A and k_B respectively, then they can use the point $k_A k_B P$. Alice can compute this point by multiplying her secret key and Bob's public key. Similarly, Bob multiplies his secret key and Alice's public key and gets the same result. But nobody else can compute this point unless he either knows the secret key of Alice or Bob or he can solve the ECDLP to calculate them.

The ElGamal protocol also has an elliptic version. The same setup is assumed but now Bob wants to send a message to Alice. It is assumed that the message is encoded as a point M of the curve; the encoding and also the decoding of messages as points of the curves is publicly known. Now Bob chooses an arbitrary positive integer l and sends the points lP and $M + l(k_A P)$ to Alice. Without the knowledge of l and k_A nobody can learn the message M. On the other hand, if Alice multiplies the first point with k_A and subtracts it from the second point, she receives exactly M.

As a last example, the digital signature algorithm using elliptic curves, ECDSA, the elliptic curve equivalent of DSA, is presented. Again, E is an elliptic curve, P is a point on E, its order is n which is now assumed to be a prime. Alice's private key is the integer k_A, her public key is the point $k_A P$. Suppose now that Alice would like to sign a message M or actually its hash value h. In order to do this, she will have to do the following. First, she chooses an arbitrary integer l such that $1 \leq l \leq n-1$ and computes $lP = (x, y)$. Let $r = x \pmod{n}$. She also computes $s = l^{-1}(h + k_A r) \pmod{n}$. Then her signature is the pair of integers (r, s).

If Bob wants to check Alice's signature (r, s) on the message M, then he will have to do the following. First, he also computes the hash value h of the message. Then he computes the point $hs^{-1}P + rs^{-1}(k_A P) = (x', y')$. Let $r' = x' \pmod{n}$. If everything is alright, then $s^{-1} = l(h + k_A r)^{-1} \pmod{n}$, so $(x', y') = hs^{-1}P + rs^{-1}(k_A P) = l(h + k_A r)^{-1}(h + k_A r)P = lP = (x, y)$. That is, Bob will accept the signature if and only if $r = r'$.

Remark: if either r or s is 0, then the above algorithm does not work properly. Namely, if $r = 0$, then Alice's private key does not contribute to the signature; if $s = 0$, then Bob will not be able to compute s^{-1}. So if either of this happens, Alice should generate a new integer l and retry.

Of course it is vital for all of the above (and similar) schemes to be secure that the underlying field, the number of points on the curve and the order of the point P be large enough.

2.3 Cracks

The main motivator of ECC cracks is 'ECC challenge', sponsored by Certicom. This company has ECC-based products, and posed the ECC challenge to improve confidence in ECC technology. Certicom offered altogether \$500,000 as prizes for the challenge. Challenges are offered in different categories (problem lengths), and at least one $GF(p)$ and one $GF(2^m)$ problem represents each category.

'Excersises' were easy challenges with the length of 79-97 bits. All of these have been solved. The second category was the category of 'Level I Challenges' which contained ECC challenges of 108-131 bits. The highest category was 'Level II Challenges' with problems of 163-358 bits. Solving these with a single computer would last, according to the estimations of Certicom, 10^{12} – 10^{41} years.

Certicom offered these challenges in 1997. Since then, all of the exercises and one 108-bit-long 'Level I Challange' were solved. The effort required for this latter was 50 times higher than the one taken to break the 512-bit RSA. The above facts show that the approximations of Certicom were basicly correct, which means that a 150-bit ECDLP is practically impossible to solve.

3 Technical background

In the previous section the basics of ECC were presented. In this section a brief introduction to smart card and Java Card technology will be given, to introduce a new platform for ECC.

3.1 What is a smart card?

Smart cards are tiny devices with the shape and size of a credit card. However, these objects yield much more. They are called 'smart' because of the microchip, that is integrated onto them. Inside this chip these cards have a CPU, non-volatile memory, and I/O peripherals. In fact, they are standardized secure portable microcomputers.

3.2 Security

Figure 3 shows the block diagram of the inside architecture of smart cards. As can be seen, smart cards are very similar to computers of von Neumann architecture. The main difference is that in case of smart cards the internal buses cannot be driven by I/O ports directly. This is an important principle of security, that ensures that outside activities cannot affect the state of the card directly [22].

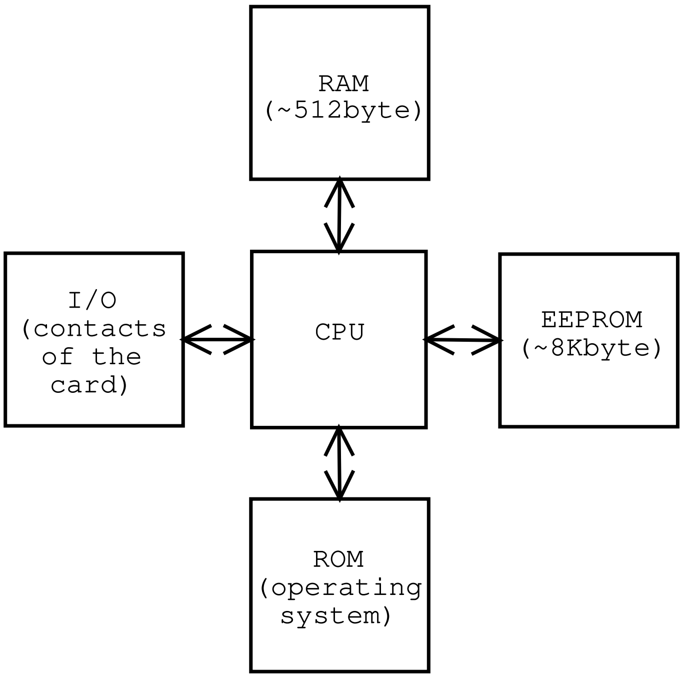

Figure 3: Physical architecture of a smart card

Smart cards are secure microcomputers designed with tamper-resistance as top priority. Information is stored in the special secure EEPROM and can only be accessed using the well-defined interface of the card with permission of the card OS and software.

Although they are inferior to personal computers in terms of speed, memory and I/O devices, they are made superior by their sophisticated security features. The interface to the outside is narrow and well defined (ISO-7816), and is a key issue of smart card security [14]. The responsibility of the on-card software is to monitor this narrow interface and grant or deny access to various data fields or pre- or post-process the output (e.g. using certain cryptographic keys). [22, 31] Since the access control logic is situated behind the gateway of the card in the ROM or EEPROM, this logic is also able to protect itself.

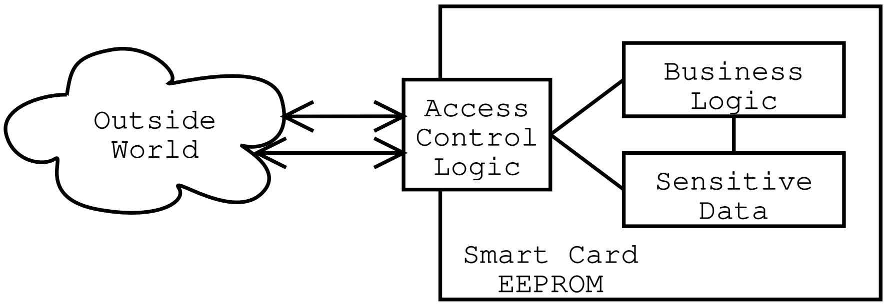

Figure 4: Logical architecture of a smart card application

Although powerful security mechanisms can be implemented on PCs as well, the sensitive data and the access control logic can be separated from each other in the case of a PC, resulting in a security hole. For instance, the hard drive can be removed and inserted into a different machine with different access control rules. In contrast, in the case of smart cards the secure data storage and the access control logic are implemented on the same microchip. The encapsulation of data storage, access control and business logic provides a secure, tamper-resistant architecture (see Figure 4).

The other main power of smart cards lies in their size. They can be easily carried in a wallet or put into the users' pocket. Since their interface is standardized, they can be connected to any ISO-7816-compatible terminal, and more and more smart cards start to appear in the life of everyday people.

The security and portability of smart cards open new possibilities for security-sensitive applications, especially in the area of key management. Cards with cryptographic potential can – besides storing users' keys – perform encryption/decryption, while the secret key does not leave the card. This can be easily ensured since the key could only be accessed through the narrow and configurable hardware firewall of the card. Some cards contain good quality random number generators, and the most advanced cards are even capable of generating 1024 bit RSA keys. Using on-board key generation it can be ensured that the key spends its entire lifecycle on the card. This means that it is generated, used and destroyed behind the secure hardware firewall, without any possibility to read it out.

Smart cards enable users to carry their secret key in their pocket and using it without exposing it to the insecure environment of the terminal. Many people consider the smart card the ultimate digital signature tool. The areas of access control and authentication also yield great perspective for smart cards.

However, the exact area where smart cards can be used is still subject to intensive research. While the narrow interface of the card to the outside world is a powerful security feature, in certain situations it may cause an enormous security hole. Although smart cards could be especially useful in case of untrusted terminals, their inability to communicate with the user on a secure channel may make them a 'handicapped security device'. (See e.g. [25, 1, 27]).

3.3 The three generations of smart cards

Smart cards vary in size of memory and capabilities. However, three generations can be differentiated.

1. The generation of *memory cards* is the earliest one. In this case a simple memory chip was placed on the card, where cells could be read or written directly. Functionally it was not a breakthrough compared to magnetic cards. Memory cards gave logical security equivalent to a floppy disk. Later on, these cards were equipped with more and more security features designed for certain applications.

2. The appearance of so-called *generic cards* opened new dimensions for smart cards. They were called 'generic', because they possessed a set of security tools that could be adjusted to the need of the each application by personalizing the card.
   These cards possessed a complex file system similar to that of a modern operating system of a PC. Data on these cards were stored in structures like files and folders. Users could be defined on these cards with various rights to different files. Users had to authenticate themselves by PIN code. The processors of more sophisticated generic cards supported cryptographic functions which enabled the utilization of challenge and response mechanisms in user authentication. A good example for this category is the Bull TB-100 card.
   At the beginning, generic cards were thought to be too expensive to be used widely. However, as their number increased, mass production made it possible to produce them cheaper. At present day, generic cards are often used even if their services are not needed. They have become cheap enough for general use.

3. Many *programmable cards* exist on the market today, but they are still too expensive. However, they are considered to be the cards of the future. Beside possessing a complex operating system in their ROM like their ancestors, they are capable of executing programs from their EEPROM. New programs can be loaded onto them, even after the issuance of the card. However, in this case special security methods should be used (see e.g. [29]).
   If their specification is open, the card manufacturer and the software manufacturer [25] do not need to have any connection. If multiple cards conform to the same specification, the same applications can run on them (see section 4.3). These cards have a functionality similar to a PC. All cards that the authors used for testing their ECC implementation (see table 1 in section 4.3) belong to this category.
   Today's programmable smart cards are relatively expensive. However, being even more generic than the so-called generic cards, mass production may lower their prices in a similar way it has happened in case of generic cards. It is likely, that in the future programmable cards will be used for purposes their services are not needed. [4] Good examples for this process are certain Oberthur AuthentIC cards (generic cards) that are implemented by an applet on an Oberthur GalactIC card (programmable card).
   In the forthcoming sections programmable smart cards and their possible applications shall be discussed.

It must be mentioned that the boundaries between the above three categories are not sharp. For example a memory card that is protected by a single PIN code could be situated between generation 1 and 2. The third generation appeared gradually too, as the access control logic of generic cards became more and more customizable. In generation 3, a business logic can be implemented with pure software.

Many cards are much more intelligent than generic cards but still do not reach the level of programmable cards. E.g. the Gemplus MPCOS-EMV card (which is commonly used in Hungary as student ID card) has strong cryptographic capabilities and can be programmed too. However, its specification is not open, and its language is not standardized, so parties independent from the manufacturers cannot write applications for it.

Some other cards are fully programmable (like Bull Odyssey I which conforms to the Java Card specification), but they are still inferior to certain generic cards because they lack crypto-support.

3.4 Java Card

Today three platforms exist for programmable smart cards on the market: Java Card, MULTOS and Windows for Smart Cards. The authors used Java Card [28] in their development since it is a widespread and fully open specification with platform-independence as top priority.

Java Card utilizes a subset of the Java programming language for card-side application development. It relies on the portability of the Java byte code, thus making applications transferable between Java Card compliant cards of different architecture.

Smart cards are security oriented platforms, and Java Card supports them with powerful security features. [8, 10, 28] As a base it relies on the security features offered by Java, with several additions. Apart from the powerful access control mechanisms, extensive type-checking is performed. Java Card supports transaction handling (to guarantee the atomicity of smart card transactions), that can be used to prevent card-removal attacks. In case multiple applications share the same card, the specification provides applet firewalling to isolate applets and to organize secure cooperation. Modern Java Cards make use of the Visa Open Platform [29] to ensure secure applet loading and deletion on the card.

Being a fully open and widespread specification, several reference implementations exist (see e.g. the cards in table 1 in section 4.3). Java Card managed to create a class-file level compatibility between different card hardwares. However, there are still certain compatibility problems (see section 4.3), especially where cryptographic support is involved.

The literature provides numerous examples for research or even formal analysis concerning the security of Java Cards. The Java Card specification was subject to both formal and semi-formal analysis. A large part of this analysis focused on proving the soundness of the Java Card Virtual Machine, especially its correctness concerning namespaces and type safety. However, certain works address problems like the possibilities of applet certification, or monitoring various communication channels between applets. (See e.g. [5, 20, 23].)

4 The implementation

Since ECC operates with keys significantly shorter than RSA, while providing the same security, it could be a considerable alternative for RSA on low-resource platforms, such as a smart card. This section is based on the authors' work in [3], describing a smart card based ECC implementation.

4.1 General remarks, aims of development

The authors' ECC implementation had a dual aim. On one hand they wanted to study ECC and perform certain measurements and calculations, on the other hand, they wanted to create a smart card based prototype ECC implementation.

Naturally, a software-based solution cannot have the performance required for commercial use. One of the authors' aims was to prove that a complex algorithm like ECC can be implemented on today's weak smart cards. The other aim was to create a solution that later could be used as a base for designing hardware acceleration. It was definitely not the goal of the authors to manifest an ECC implementation fast enough for commercial use, since pure software implementations cannot compete with hardware-acceleration. The top priority of the development was to implement efficient algorithms (with polynomial complexity) on low-resource smart cards.

The program is capable to run on PC and on Java Card. That means that the same source code and the same Java classes are running on both platforms. That means the authors' aim was not to optimize the program for any platform (not even on a Java VM), but to implement a portable solution capable of running on both platforms. Due to the limited amount of memory of the card, speed often had to be traded for memory.

4.2 Architecture of the program

4.2.1 Structure

The program was designed in a modular form taking advantage of object orientation in Java. This enabled the authors to experiment with various finite fields and various representations in the card's memory. It is easy to switch to another Galois field because of the structure of the ECC engine. (Figure 5).

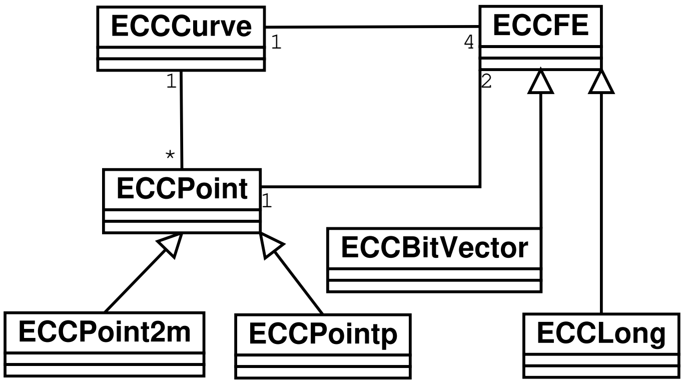

Figure 5: Simplified UML diagram of the ECC engine

The arithmetic of field elements is independent from the curve and can easily be replaced by another arithmetic. The point-arithmetic layer situated on the top of the field arithmetic layer implements operations between points of the curve (Figure 6). The ECC engine is the top-level layer which implements ECDLP-based protocols (see section 2.2.3), totally independent of the chosen field.

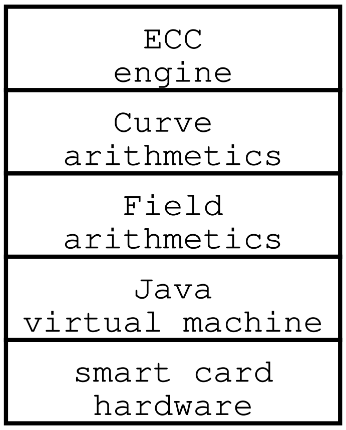

Figure 6: Layer structure

Applications using the engine can only access this upper layer, that grants access only to public data (the curve, the public point, the user's public key) and denies access to private data (e.g. the user's private key). Naturally, this layer offers services like initialization, encryption, decryption and signing described in section 2.2.3. The responsibility of this layer is to ensure that the engine works securely and consistently and with correct parameters only.

On the top, application specific layer could be placed by developers wishing to use the engine. The responsibility of this layer would be to implement application-specific functionality relying on the operations of the ECC engine:

- Define various principals who may access the ECC engine (e.g. user and administrator)
- Perform authentication of the above principals (e.g. the user is authenticated using a PIN code, while the administrator should be authenticated using a challenge and response method)
- Assign operations of the ECC engine to the above principals (e.g. only the administrator may modify system-wide parameters, while only the user may access his or her own public key)
- Manage application-specific data (e.g. personal information of the user)

4.2.2 Memory management

Memory appeared to be the main bottleneck in two ways. Physically, the amount of memory in the card was a strict limitation. Logically, the memory management of the Java Card platform is much less flexible than the memory management in Java.

The development was done using the Bull Odyssey I card with 7048 bytes of memory. This small amount of memory had to contain the data required for the calculations and the code of the ECC engine too. Consequently, in order to enable cryptographically meaningful problems, only simple algorithms could be implemented.

The memory management of the Java Card platform is rather weak in the software engineering sense, since there is no possibility to reuse memory. Memory allocated for objects cannot be freed, only if the entire applet is removed from the card. According to the specification, all memory reservations should be made in the constructor of the Java Card applet, to guarantee that the applet will not run out of memory. This is an important security principle. A critical attack would be against the card to bring it into an inconsistent state by filling its memory, and making it stop at an unpredictable point [28, 7].

Naturally, the amount of memory required by the applet is a function of the key size used. It could be given as:

$$OS + JCVM + Package + Engine + (K + Keysize) * n$$

Where:

- OS is the amount of memory required by the card operating system. This may vary from card to card. While the OS of the Odyssey I card is less then a kilobyte, in case of cards with crypto-support this is larger. The Oberthur Simphonic card is also a SIM card, with additional services, so in this case OS is over 20k.
- $JCVM$ is the memory required by the Java Card Virtual Machine.
- $Package$ is the amount of memory required by the code of the program in the chosen card's byte code language. Although the specification ([28]) strictly specifies the format of the Java Card byte code file, card manufacturers often modify it, so this is card-dependent too. It is approximately 5k.
- $Engine$ is the amount of memory required by the non-static variables of the applet. This is relatively small (<200bytes). Due to the above reasons this is card-dependent too.
- $Keysize$ is the size of a $GF(2^m)$ field element. The authors' applet is able to deal with 109-bit-long keys ($Keysize = 14$ Bytes) on Odyssey I.
- K is the amount of memory used by a field element object apart form the key itself. This means the memory required for object construction. This is relatively small, but may vary from card to card. (<10 bytes)
- n means the number of field element (FE) objects required by the applet. In this implementation, the $GF(2^m)$ arithmetic requires 5 FE objects. 9 more are needed for the Euclidean algorithm and inversion. For curve arithmetics 13 more were required. Altogether 27 FE objects are needed for the ECC calculations.

Unfortunately, several values are card-dependent and kept secret by card manufacturers. In many cases these can only be approximated. Performing the above calculations and approximating the unknown values gives the result that after uploading the applet with a 109-bit-long key, the Odyssey I card still has a significant amount of free memory (approximately 2 kilobytes). However, the authors were unable to upload the applet with larger keys to Odyssey I. Perhaps, some temporary memory while installing the applet. Unfortunately, the documentation does not mention this.

Naturally, in case of cards with more memory, larger keys could be used. However, similar undocumented features were experienced with other cards too. It seems that the Java Card technology is not yet prepared for programs of this complexity. Therefore, the authors tried to compare the performance of these smart cards using the same key size (109 bit).

4.2.3 Algorithmic solutions

In practice, ECC is used both above $GF(p)$ and $GF(2^m)$. The decision of the authors was to implement ECC on the card above $GF(2^m)$. The main reason for this was that computers can deal much easier with bit-vectors than numbers in $GF(p)$. Apart from making calculations faster and more simple, this decreased the size of the program too.

The following operations were implemented between $GF(2^m)$ elements (m denotes the length of the field elements):

- *addition*: In $GF(2^m)$ this means bitwise xoring of the two field elements. This is the trivial and optimal solution, which can be done in $O(m)$ steps.
- *obtaining the remainder* (in modular calculations): Our aim is to obtain the value of P modulo M. This can be done in $GF(2^m)$ using one single addition if $\deg(P) = \deg(M)$. If $\deg(P) < \deg(M)$, there is no need to calculate the remainder, but if $\deg(P) > \deg(M)$, calculating the remainder is a complex operation. The authors' decision was to exclude this latter case by not allowing the case $\deg(P) > \deg(M)$. If the size of P would grow above the size of M by one bit, the remainder is calculated immediately. The size of polinoms stored in the system are $m+1$ bit. If the MSB is zero, the poinom P is a valid element of the field. However, if the MSB would become 1, M has to be subtracted from P. In case of M the MSB has to be 1. This is not the most efficient method, since the aforementioned calculation has to be made every time the length of P would increase. However, this method consumes only a little amount of memory. So, obtaining the remainder is an addition with the cost of $O(m)$.
- *multiplication* was implemented like the script multiplication. In worst case m additions ($O(m)$) are performed. The total cost of the algorithm is $O(m^2)$. Naturally, this could be made faster with more memory available. (see section 5.2 and [24])
- *division (with residue)*: The script division was implemented here with shiftings and additions at the total cost of $O(m^2)$.
- *division (multiplication with the inverse)*: This means two operations; obtaining the inverse using the Euclidean algorithm [12], then a multiplication ($O(m^2)$). The Euclidean algorithm contains $O(m)$ divisions (with residue) (cost: $O(m^2)$). Thus, the total cost is $O(m^3)$.
- *negation*: In $GF(p)$, negation would be a much more difficult operation than in $GF(2^m)$, where the negate of a binary polynom is itself.

The above are operations between the elements of the field. The following operations are defined in the group of the points of the curve:

- Addition of P and Q points (where $P \neq Q$). The operation with the highest cost is division (multiplication by the inverse), so the total cost is $O(m^3)$.
- Duplication of point P (i.e. calculating $P + P$). $O(m^3)$ is the cost for similar reasons.
- Multiplication of point P by a number k (this is the cryptographically important operation): The naive algorithm (k additions) would be too slow. With a method similar to 'the method of repeated squaring' used *e.g.* in RSA [12], $O(\log(k))$ additions are enough.

4.3 Smart cards used for testing

The authors developed their ECC engine using the Bull Odyssey I smart card. The applet was written and optimized in order to be able to run using the limited resources of that card. Later on the engine was tested on two other smart cards too (see table 1). The applet was not rewritten to make use of the additional memory available.

Naturally, choosing a card with more memory would have given more freedom to implement faster but more complex algorithms. However, at the time of the development only the Odyssey I card was available. Moreover, Odyssey I has the best performance in those cases when the built-in crypto coprocessor cannot be used (see table 2).

Card	Odyssey	Cyberflex	Simphonic
Manufacturer	Bull	Schlumberger	Oberthur
Version	I	V3C	V3
Memory	8k	16k	64k
Crypto Support	No	Yes	Yes
Java Card	2.1	2.0	2.11

Table 1: Smart cards used for testing

4.4 Results

4.4.1 Java cards

As described in section 4.2.2, memory management was the key issue. The engine was optimized so that it could run on the Odyssey I card with the largest possible keys. Certicom's ECC2-109 challenge was chosen to be implemented on the card, since:

- It is practically strong enough since no one has broken an ECC problem with this difficulty by today.
- It is theoretically strong enough since its strength is similar to that of 1024 bit RSA.
- The challenge uses $GF(2^m)$ arithmetic.
- The key size is small enough to fit on the Odyssey card.

Certicom's ECC2-109 challenge was installed on the Bull Odyssey I card. According to the authors' measurements, an elementary operation (addition of two points) required approximately 10 minutes.

As described in section 4.2.3, approximately $\log k$ additions are needed to multiply a point with k. According to the theorem of Hasse that the number of points of a curve above $GF(q)$ can be approximated by q (see section 2.1.2), it can be supposed that $k \leq q$, since a higher k would have no cryptographic significance. That means that $\log k$ can be approximated by m. So in this case, approximately 109 additions are needed for a cryptographic operation (a multiplication of a point by a scalar). This would require many hours, which would be unacceptable in case of a commercial product.

However, with proper hardware acceleration it could be increased dramatically. The RSA algorithm with 1024-bit keys can be executed on a smart card with crypto-coprocessor (e.g. Schlumberger's Cyberflex) in less than 1 second. According to the literature, an RSA and an ECC encryption (with similar strength) takes similar amount of time [11]. Since ECC above $GF(2^m)$ can be parallelized easier than RSA (especially because of the low complexity of field element addition and obtaining the remainder [30]), a hardware acceleration would decrease the time-cost even below that of RSA. For more improvement possibities, see section 5.2.

The authors welcome the smart card manufacturers' uprising interest in ECC technology, such as the Atmel AT90SC6464C microcontroller or Oberthur's recent announcement of the ECC support for Authentic cards. Unfortunately they do not know of any existing card on the market with hardware ECC support.

Table 2 shows the test results with three different Java Cards. Although the program was not optimized for Odyssey, its results were significantly better than those of other cards. However, unlike its own PC-side emulator, Simphonic was not able to execute certain parts of the program. This latter card also showed certain features incompatible with the Java Card standard.

Field element operations with one operand were performed on the a parameter of the challenge, those with two operands were performed between the a and b parameters of the challenge. Point operations were performed between the P and Q points of the appropriate challenge. Table 3 shows the results for the group operations on Oberthur I for three different curves.

Card	Odyssey	Cyberflex	Simphonic
$A+B$	0:01.1	0:02.8	0:02.8
$A*B$	0:48.2	3.05.2	3:37.5
$A\,div\,B$	0:02.0	0:06.5	0:06.6
A^{-1}	8:11.2	29:15.0	27:19.2
$P+P$	9:32.1	33:53.2	N.A.
$P+Q$	9:57.3	36:45.1	N.A.

Table 2: Test results with three different cards. ($A, B \in GF(2^m)$, $P, Q \in E$, where E is the ECC2-109 elliptic curve. The meaning of $x:y$ is x minutes and y seconds.)

	ECC2-79	ECC2-89	ECC2-109
$P+P$	3:08.8	4:23.2	9:32.1
$P+Q$	3:23.6	4:58.1	9:57.3

Table 3: Test results with three different curves on Odyssey I.

The ECC system described in this paper is able to perform ECC encoding on a Java Card, but its speed is so low that it can only be applied in laboratory experiments. The authors hereby emphasize, that this applet is not a commercial product, but rather a prototype of a future hardware accelerated ECC. For some imporvement possibilities, see section 5.2.

4.4.2 PC

Although the program was developed for Java cards, the same source code and byte code is able to run on a PC too. It can be compiled using JDK, and encapsulated in any Java program. On a PC, the speed and memory limitations of smart cards are not to be considered, so that a field of practically any size can be applied.

The system was tested with the ECC2-109 problem on PC too, to be a valid comparison. The system performed the most complex cryptographic operation (multiplication of a point by a scalar) in 28 seconds (on a Pentium II 233MHz running Linux and JDK1.1.7). Since this is by far the most complex operation in all ECC-based protocols (see section 2.2.3) and is performed only once (or twice in the case of ElGamal encryption and ECDSA verification algorithms – however, in both cases the two calculations can be made fully parallel), this is approximately the amount of time required by encryption, decryption, signing and signature verification as well.

In a PC environment, the program is ready for practical use too. However, it was not specifically prepared for PC, so it is by far not optimal. Replacing the finite field with the *java.math.BigInteger* class (that is implemented in native code) would give the program tremendous performance boost in a PC environment. Unfortunately, the *java.math.BigInteger* class does not exist for smart cards.

5 Evaluation

5.1 Security

Although the test results of the above section clearly show the PC's superiority in speed, the smart card's main advantage does not lie in the numbers. In case of the PC the secret key is stored on the hard drive, loaded into the memory and used by the processor. An attacker has thousands of possibilities to get access to it. They do not even need direct access to the machine itself to tap the internal buses or steal the hard drive: if the computer is connected to a network, a virus or a Trojan horse may also get access to the key in the machine.

However, if the encryption engine is running on a smart card, the above methods do not work:

- The internal buses cannot be tapped, because there are not any. The whole smart card is one single microchip [22].
- The data storage device cannot be separated from the access control logic (see section 3.2).
- If the card is stolen, it is protected by a PIN code, which cannot be accessed due to the above reasons.
- After a certain amount of tries (typically 3) the PIN is blocked, and the card renders itself totally useless.
- Certain securely guarded keys are needed to load applications onto the card. This excludes malicious programs (e.g. viruses or Trojan horses) [29].
- If malicious applications are uploaded, the Java Card applet firewall still prevents them to access the secret key or the ECC application [8].
- The secret key never leaves the secure area behind the smart card's hardware firewall (see section 4.2.1).

Although the secret key can be used much faster on a PC, using it on the smart card has significant security advantages. If a message is signed with a secret key residing securely on a smart card, the receiver of the message can be absolutely sure that the card was present when the message was signed. If a message is encrypted with a public key, the sender can be sure that the message can only be opened when the smart card (with the secret key) is present. This way, smart cards transfer the meaning of security from the physical world to the electronic one, thus making electronic security more understandable to non-professionals. Naturally, the above is true for a smart card with any kind of public key cryptography.

In the case of ECC, key generation also needs careful investigation, since the keys are not just random numbers, but special numbers (which is also quite true for RSA). Key generation may also include the generation of system parameters, such as the curve E, and the base point P along with its order n. E has to be chosen in such a way that curves for which an efficient attack is known – e.g. anomalous curves – should be avoided. For the generation of the base point P we have already described a randomized algorithm in Section 2.1.2. However, there are more sophisticated algorithms for this, which guarantee that P will have a high order. The generation of k is simple, because it is just an integer in $[0, ..., n-1]$ (however, it should be statistically unique and unpredictable). The user's private key is then k, and the corresponding public key is either kP if the other parameters (E, P and n) are system-wide, or (E, P, n, kP).

In the first case, key generation is simple and can be realized on-board. This provides for maximum security and also avoids key injection requirements at manufacturing time. In the other case, off-board key generation seems to be the right choice because of the complexity of the process.[15]

5.2 Improvement possibilities

5.2.1 Algorithmic improvements

Two points were found to possess extremely critical influence on the performance of the algorithm. One of them was division in the underlying field (multiplication with the inverse), the other was the multiplication of a point with a scalar (see section 4.2.3). Unfortunately these two operations are encapsulated into each other in ECC.

Division of field elements has two components: obtaining the inverse and multiplication. The former was performed using the Euclidian algorithm, which is a highly efficient algorithm. However, it also contains multiplications, divisions (with remainder), and additions. If these operations could be accelerated, the whole process of obtaining the inverse would be faster. Addition is unlikely to be accelerated since $\Omega(m)$ steps are necessary. However, the speed of multiplication and division with remainder could be improved significantly. For example while shifting, large areas of 0-s could be shifted in one step, and not one by one. It would also be possible not to perform multiplication modularly, only converting the result to a valid field element. Multiplication would also be faster with the optimal normal base representation, however, it would decrease the speed of addition and it would also reduce security (see section 2.2.2). Another way of acceleration would be possible if the generator element of the multiplicative group of the base field would be known. Multiplication would boil down to simple addition.

Accelerating the multiplication of a point by an integer would be an even more difficult problem. Since no algorithm is known to calculate $k * P$ directly, multiplication can only be implemented as a sequence of additions. Taking this into consideration, the alorithm used by the authors is quite effective. A constant acceleration would be possible with some preprocessing, if the points of the form $2^i \cdot P$ were stored in a table. The complexity of the operation would still be $O(\log k)$, but the number of steps would decrease to its half. This table could be precalculated when the system is installed, since P is a public system-wide parameter. However, this would comsume an amount of memory which can only be taken into consideration on cards with more memory than Odyssey or on the PC.

The literature describes several other tricks to optimize ECC for performance [24]. Unfortunately, the authors could not apply them due to the lack of memory on the card.

5.2.2 Technical improvements

The authors focused on functionality: making the ECC algorithm work on a smart card platform. Thus, the implementation is not protected against various side channel attacks ([16]) such as the analysis of key-size dependent execution times or analysis of power consumption ([18]) or electromagnetic field ([2]). Naturally, these would require optimization for a specific smart card.

5.2.3 Further possibilities

As already noted, the implementation described in this paper is handicapped (concerning speed) in many ways:

- it is pure software;
- it is implemented in Java Card, which does not enable direct access to the smart card hardware;
- several known improvement methods had to be omitted because of the very little amount of memory available.

Therefore it is not fair to assess the applicability of ECC in a smart card setting based on the above numbers. Commercial availability would definitely require hardware acceleration, for instance in the form of an ECC-coprocessor.

Experience with Java programs has shown that their direct machine code implementation yields generally at least one order of magnitude acceleration. Moreover, the experience with the hardware acceleration of RSA has shown that a crypto coprocessor can give an additional factor of 20-100 performance boost [26]. In the case of ECC, this ratio can be even better since ECC requires less multiplication units and memory than RSA, which enables even stronger parallelization and pipelining. Hardware acceleration ratios for ECC are estimated to be about 2^{12} (see [30] for more detail). This would place the time required for ECC calculations below the 1 second line, even considering the restricted possibilities of smart cards.

5.3 Comparison with another Java Card ECC implementation

A similar development took place on the Helsinki University of Technology (HUT) in parallel with the authors' development, unknown by the authors at that time. The aim of that project was to create an ECDSA-based authentication system, using the Java smart card as a device for encryption and key generation [10].

There are a lot of similar and different features between the two engines. Both are pure software implementations of ECC using Java Card technology. The engineers of HUT also faced the dilemma of $GF(p)$ vs. $GF(2^m)$ (see section 4.2.3), but unlike the authors of this paper, they chose $GF(p)$, since they already possessed a well-tested Java ECC $GF(p)$ implementation that they wanted to port to the Java Card platform. They, too, claimed memory management and the lack of garbage collection to be the most severe problems.

The engineers of HUT developed their implementation for a Java Card with 16KB of storage. The maximum key size they could use to perform an inversion of a field element (that takes approximately as much time as addition between points of the curve) was 50 bit. Of course, such key sizes are cryptographically insignificant. In contrast, the authors of this paper created their implementation for a Java Card with only 8KB of storage and the maximum key size they could handle was 109 bit. An ECC with this key size has never been cracked, so it is indeed cryptographically significant.

6 Conclusion

Elliptic curve cryptography is a young part of public key cryptography. Although it is based on a more than fifteen-year-old theory, it is less widespread than RSA. However, ECC has several advantages to its adversary. Also, it is very important to have an alternative to RSA. Accordingly, the role of ECC has increased in the past significantly. Several standards include now ECC in company with RSA.

Because ECC is able to provide the same grade of security as RSA with significantly smaller keys, it has great potential in a smart card or mobile environment. The authors created the first ECC implementation that is able to deal with cryptographically significant problems on a Java Card. They hope that their research will contribute to a better understanding of ECC. With their smart card implementation they would like to demonstrate the significance of ECC, and the power of Java Card technology, which is now capable of executing more complex operations.

However, the aim of the research was not to create a commercial product. The authors' work could rather be regarded as a prototype, to be a base of a future hardware-supported smart card implementation of ECC.

References

[1] M. Abadi, M. Burrows, C. Kaufman, and B. Lampson. Authentication and Delegation with Smart-cards. Theoretical Aspects of Computer Software: Proc. of the International Conference TACS'91, Springer, Berlin, Heidelberg, 1992.

[2] D. Agrawal, B. Archambeault, J.R. Rao, and P. Rohatgi. The EM Side-channel(s). Workshop on Cryptographic Hardware and Embedded Systems 2002 (CHES 2002), San Francisco Bay (Redwood City), USA, August 13 - 15, 2002, 2002.

[3] I. Zs. Berta and Z. Á. Mann. Elliptikus görbéken alapuló nyilvános kulcsú kriptográfia elemzése chipkártyás és PC-s környezetben. Scientific student circle conference, Budapest University of Technology and Economics, 2000.

[4] I. Zs. Berta and Z. Á. Mann. Smart Cards – Present and Future. Híradástechnika, Journal on C^5, 2000., vol 12, 2000.

[5] P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Electronic purse applet certification: extended abstract. Proceedings of the workshop on secure architectures and information flow, Royal Holloway, 1999.

[6] Certicom. Robert Harley and Team Win 10,000 USD Prize in Certicom's ECC Challenge. http://www.certicom.com, April 2000.

[7] Zhiqun Chen. How to write a Java Card applet: A developer's guide. JavaWorld, July, 1999, 1999. http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard_p.html.

[8] Zhiqun Chen. Java Card Technology for Smart Card: Architecture and Programmer's Guide. Addison-Wesley Pub Co; ISBN: 0201703297, 9 2000.

[9] D. A. Crutchley. Cryptography and Elliptic Curves. Master's Thesis, University of Southampton, Faculty of Mathematical Studies, 1999.

[10] T. Elo and P. Nikander. Decentralized Authorization with ECDSA on a Java Smart Card – A Software Implementation. Cardis2000, 2000. http://www.hut.fi/ telo/publications/.

[11] Csilla Éva Endrődi. Master's thesis on comparison of the ECC and the RSA algorithms. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2001.

[12] L. Györfi, S. Győri, and I. Vajda. Információ és kódelmélet. Typotex, ISBN: 9639132845, 2000.

[13] D. Husemöller. Elliptic Curves (Graduate Text in Mathematics). Springer, ISBN: 0387954902, 1987.

[14] ISO. ISO/IEC 7816.

[15] D. B. Johnson and A. J. Menezes. Elliptic curve DSA (ECDSA): an enhanced DSA. http://citeseer.nj.nec.com/276964.html, 1998.

[16] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. Journal of Computer Security, v. 8, n. 2-3, 2000, pp. 141-158., 2000.

[17] N. Koblitz. Elliptic Curve Cryptosystems. Mathematics of Computation, Vol. 48. No. 177, 1987, 203-209., 1987.

[18] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. Crypto '99, http://www.cryptography.com, 1999.

[19] V. S. Miller. Use of Elliptic Curves in Cryptography. CRYPTO '85, volume 12, 1985.

[20] Stéphanie Motré. Formal Model and Implementation of the Java Card Dynamic Security Policy. Gemplus Research Laboratory Avenue du Pic de Bertagne 13881 Gémenos CEDEX, 2000.

[21] Mullin, Onyszchuk, Vanstone, and Wilson. Optimal Normal Bases in $GF(p^m)$. Discrete Applied Math, 1988, vol 12, 1988.

[22] W. Rankl and W. Effing. Smart Card Handbook. John Wiley & Sons, 2nd edition, ISBN: 0471988758, 1997.

[23] Antoine Requet. A B Model for Ensuring Soundness of a Large Subset of the Java Card Virtual Machine. Gemplus Research Laboratory, Av du Pic de Bertagne, 13881 Gémenos cedex BP 100.

[24] Michael Rosing. Implementing Elliptic Curve Cryptography. 1998, Softbound, ISBN 1884777694, 1998.

[25] B. Schneier and A. Shostack. Breaking up is Hard to do: Modelling security threats for smart cards. USENIX Workshop on Smart Card Technology, Chicago, Illinois, USA, 1999. http://www.counterpane.com/smart-card-threats.html.

[26] Bruce Schneier. Applied Cryptography. John Wiley & Sons, ISBN: 0471117099, 1996.

[27] Tage Stabell-Kulo, Ronny Arild, and Per Harald Myrvang. Providing Authentication to Messages Signed with a Sart Card in Hostile Environments. Usenix Workshop on Smart Card Technology, Chicago, Illinois, USA, May 10-11, 1999., 1999.

[28] Sun Microsystems Inc. Java Card (TM) 2.1.1 Application Programming Interface. Sun Microsystems, Inc. 901 San Antonio Road Palo Alto, CA 94303 USA 650 960-1300, 5 2000. http://java.sun.com/javacard.

[29] Visa. Open Platform Protection Profile (OP3). http://www.visa.com, 2001. Version 0.7.

[30] M. J. Wiener. Performance comparison of public-key cryptosystems. *CryptoBytes*, 4(1), 1998.

[31] J. L. Zoreda and J. M. Oton. Smart cards. Artech House, ISBN: 0890066876, 1994.