Implementing Elliptic Curve Cryptosystems in Java 1.2

Yki Kortesniemi

yki.kortesniemi@hut.fi
Helsinki University of Technology

Abstract

Elliptic curves have been studied for a long time, but only for a little over ten years have they been used for cryptographic applications. Compared to other available public key cryptographic systems, they provide shorter key lengths, which are believed to translate to increased speed. On platforms like Java, which at the moment is slower than C, this speed difference is of importance.

The Java Development Kit (JDK) 1.2 includes the definitions for different types of cryptographic services, but only a few algorithms and implementations. In this paper, we propose a way of implementing elliptic curve cryptographic services in JDK 1.2 and discuss our implementation of a signature service.

1. Introduction

The increasing importance of Internet and the multiple client platforms therein have made Java an alluring platform for various applications. Many of them require some form of security and access control, which – along the rest of the application – has to be implemented in Java. However, Java is slower to execute than C – the fastest Just In Time compilers produce code that approaches 40% of the speed of C++ [1]. Therefore, some careful consideration is required in choosing the cryptographic technology to be used.

Elliptic curves have been studied for a long time as a branch of mathematics, but only for a little over ten years have they been considered for cryptographic applications. Their security relies on a modified Discrete Logarithm (DL) problem, which means that many existing algorithms for Discrete Logarithms can be used with minor modifications. The problem is also somewhat harder than regular DL or factorising, so shorter keys can be used. As a result, compared to RSA, elliptic curves can be made faster on platforms with limited computational power.

This speed difference can prove to be very important in many applications. One application we have been studying in our project is to use SPKI certificates for access control [2]. The service provider would provide his resellers with access certificates to the service. These access rights could then be delegated by issuing new certificates. The end user would then present his own certificate whenever he wishes to use the service. To verify that the user is authorised to access the service, the certificate chain has to be dynamically resolved. Resolving the certificate entails verifying the signature on each certificate. In practice, this can lead to several consecutive signature verifications with public key operations – so, a saving in each verification is going to be multiplied.

Another consequence of using elliptic curves could be that faster key generation

can also lead to completely new application areas. If temporary public keys become computationally cheap, they can be more generously used to provide, for instance, anonymity in different operations and thereby to improve privacy.

Our goal has been to hide the implementation details behind an API so that more optimised implementations can later be developed. Possible improvements could be based on using different algorithms, different representations for arithmetics, native implementations of the key parts of the algorithm (with the loss of portability, though), or even switching to hyperelliptic curves.

The rest of this paper is organised as follows: Section 2 introduces elliptic curves, talks about their application for cryptography, and finally discusses their security. Section 3 discusses JDK 1.2 security in general and the cryptography related functionality in more detail. In Section 4, we elaborate on the problems and possibilities of implementing elliptic curves on JDK. Section 5, then, describes our implementation of signatures based on elliptic curves. Finally, Section 6 discusses some ideas for further development and Section 7 presents a conclusion.

2. Elliptic Curve Cryptosystems

Public key cryptosystems are based on mathematical one-way functions that have a trapdoor. One-way functions are easy to evaluate in one direction, but very difficult in the other direction – just like it is easy to make guacamole from an avocado, but it is very difficult the other way. If a trapdoor is introduced, however, the process can be reversed via the trapdoor – and the guacamole can become an avocado again. In a public key cryptosystem the function itself constitutes the public key and the trapdoor information functions as the private key.

Many one-way functions with a trapdoor, can be used in the other direction as well: the information can be encrypted via the trap door and decrypted with the function itself. This makes it possible to use the same system for both encrypting and signing data.

One example of a suitable family of mathematical functions with a trapdoor is the discrete logarithm in a finite field. A finite field $F(q)$ has q elements, where q is a prime number. Typically, these elements are represented as the integers from 0 to $q-1$. In such a field, it is fairly easy to calculate b from the formula

$$b = a^x \bmod q,$$

but if x is kept secret, it is very difficult to calculate x from a, b and q. This problem has been used in several cryptosystems, among others in the DSA.

2.1 Elliptic curves

Elliptic curves are simple equations of the form $F(x, y) = 0$, where F is a polynomial of degree 3. Using the XY-plane, they can be plotted as a gently looping curve. They have been studied for a long time as a means of calculating the circumference of an ellipse (whence the name Elliptic Curve).

In cryptographic applications, elliptic curves are not based on the real numbers, but on a finite field, like discrete logarithms. Again, one possibility is the prime field $F(q)$.

There are several equations to define elliptic curves, but the most common are based on the Weierstrass equations [3]. In the prime finite field F(q) the equation can be stated as

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

where a and b are integers modulo q. In addition, a and b have to satisfy the equation

$$4a^3 + 27b^2 \neq 0 \pmod q.$$

Now, the elliptic curve consists of all the integer solutions *(x, y)* to the defining equation, along with an additional point at infinity.

The number of points (including the point at infinity) on the elliptic curve *E* is called the order of *E* and it is denoted by *#E*.

2.2 Operations on the Elliptic Curve

The discrete logarithm problem in a finite field is based on exponentiating a point to a scalar *e* (repeatedly multiplying by itself *e* times). In elliptic curves, the corresponding problem is based on multiplying a point on the curve with a scalar *k* (adding the point to itself *k* times). To appreciate the difficulty of reversing the scalar multiplication, let us first look at the operation of adding points on a curve.

Geometrically, the operation of adding two points on the curve can be defined as follows:

- Draw a straight line through points *u* and *v* to find a third intersecting point *w*, then
- Draw a vertical line through *w* to find the final intersecting point *z*.

Then, $u + v = z$.

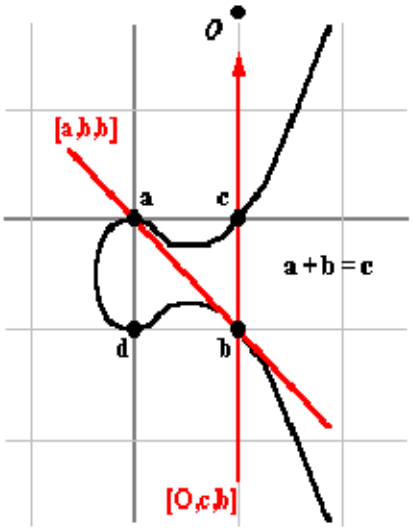

Figure 1: An elliptic curve defined by the equation $y^2 + y = x^3 - x^2$. [4]

As an example, in figure 1, points *a* and *b* are being added together. First, a line is drawn from *a* to *b*. As the line is a tangent to the curve at point *b*, the point *b* actually contains two points and therefore the third intersecting point on this line is also found at *b*. The vertical line through *b* intersects the curve at *c*, which is the answer to *a* + *b*. Now, if we were to repeatedly add point *a* to itself, we would visit all the five points on the curve in the following order (see figure 2): *b*, *c*, *d*, *0*, *a*.

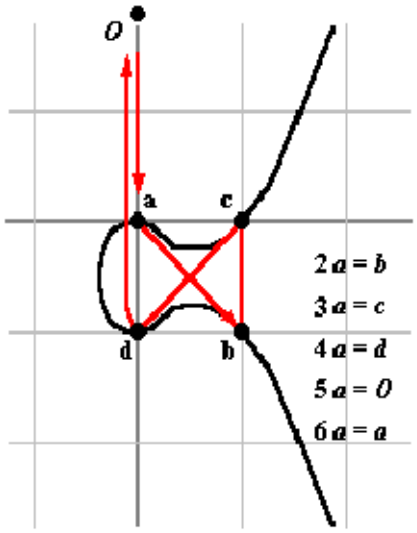

Figure 2: The result of repeatedly adding point a to itself. [4]

In this case, all the points on the curve are visited, but depending on the curve and the starting point, typically only a subgroup of the points on the curve are visited. We call the number of points in the subgroup *r*. We can state this more formally by saying that the order of a point *P* on an elliptic curve is the smallest positive integer *r* such that $rP = 0$. The order of the point always exists and it divides the order of the curve (*#E*). The result of the division *#E/r* is denoted by *h* and is called the cofactor. The cofactor is used to validate the curve parameters. The point *P* is normally chosen so that *h* remains small. For security reasons, the order of the point should be prime, even though the order of the curve does not have to be prime.

If we were to choose a starting point *G* of a sufficiently high prime order and then

find point Q by choosing a random integer d and calculating

$Q = dG$,

it would be difficult deduce d from G and Q. This problem forms the elliptic curve analogue of the discrete logarithm.

We can now publish the curve and the starting point G, which can be used for several secure transactions (with some limitations discussed later). Then, for each key pair we choose a random scalar d, which is the private key, and calculate Q, which is the public key.

2.3 Cryptographic considerations

The security of elliptic curve cryptosystems relies on the difficulty of resolving d from the equation $Q = dG$, if Q and G are known. This is most difficult, when the order of G (that is, r) is prime. The best method known today, the Pollard-λ method, finds d in $O(\sqrt{r})$ operations [5]. Therefore, to equal 1024 bit RSA in security, r should be 160 bits in length.

In practice, it is not trivial to assure that r is prime. However, there exist methods that can be used to minimise the probability of r not being prime. From the security point of view, it should suffice, if the probability of r not being prime is smaller than the probability of an attacker finding d anyway using Pollard-λ.

Further, there exist some special curves that can be reduced to more easily solvable forms thereby reducing the security. Menezes, Okamoto and Vanstone (MOV) [6] proved that an elliptic curve over $F(q)$ can be reduced to the discrete logarithm in the finite field $F(q^B)$ for some $B \geq 1$. This attack is practical only if B is small, so to prevent this, the curve has to satisfy the MOV condition; that is: B has to be high enough. Another group of unacceptable curves are the anomalous curves, whose order $\#E$ equals the order of the underlying field (that is, q). Again, this can be easily verified when the curve is created.

If the same curve and starting point G are used repeatedly, the security is reduced: if the same parameters are used n times, it only takes $\sqrt{n}$ times longer to find all the private keys than to find a single key. Therefore, n should not be exceedingly large. On the other hand, if the curve is chosen so that it is impossible to break even the first key, the other keys can not be broken either.

Finally, it should be noted that the security of elliptic curves in general has not been proven. It is merely assumed that they should be secure, as nobody has yet provided any proof to the contrary. So far, elliptic curves have not been studied as extensively as RSA, for instance, so they could prove to be vulnerable to some new attack. To balance things, it should be noted that even the security of RSA relies on assumptions; it has not been proven either.

3. Cryptography in JDK 1.2

JDK defines and partially implements security related functionality as part of its core API. This functionality is collected in the java.security package and its subpackages. To facilitate and co-ordinate the use of cryptographic services, JDK 1.1 introduced the Java Cryptography Architecture (JCA). It is a framework for both accessing and developing new cryptographic functionality for the Java platform. JDK 1.1 itself included the necessary APIs for digital signatures and message digests.[7]

In JDK 1.2, JCA has been significantly extended. It now encompasses the cryptography related parts of the Java Security API, as well as a set of conventions and specifications. Further, the basic API has been complemented with the Java Cryptography Extension (JCE), which includes further implementations of encryption and key exchange functionality. This extension, however, is subject to the US export

restrictions and is therefore not available to the rest of the world. To fully utilise Java as a platform for secure applications, the necessary cryptographic functionality has to be developed outside US.

3.1 The Java Cryptography Architecture

One of the key concepts of the JCA is the provider architecture. The key idea is that all different implementations of a particular cryptographic service conform to a common interface. This makes these implementations interchangeable; the user of any cryptographic service can choose whichever implementation is available and be assured that his application will still function.

To achieve true interoperability, JDK 1.2 defines cryptographic services in an abstract fashion as engine classes. The following engine classes, among others, have been defined in JDK 1.2:

- MessageDigest – used to calculate the message digest (hash) of given data
- Signature – used to sign data and verify digital signatures
- KeyPairGenerator – used to generate a pair of public and private keys suitable for a specific algorithm
- KeyFactory – used to convert opaque cryptographic keys into key specifications (underlying representation of the underlying key material)
- CertificateFactory – used to create public key certificates and Certificate Revocation Lists (CRLs)
- AlgorithmParameters – used to manage the parameters for a particular algorithm
- AlgorithmParameterGenerator – used to generate a set of parameters to be used with a certain algorithm

A generator is used to create objects with brand-new contents, whereas a factory creates objects from existing material.

To implement the functionality of an engine class, the developer has to create classes that inherit the corresponding abstract Service Provider Interface (SPI) class and implement the methods defined in it. This implementation then has to be installed in the Java Runtime Environment (JRE), after which it is available for use.[7], [8]

4. Adding an Elliptic Curve cryptosystem to JDK 1.2

The JDK 1.2 incorporates the concept of cryptographic engine classes, that implement a particular cryptographic function using a certain standard algorithm. These algorithms have been mentioned in JDK 1.2. The users are then expected to use these standard names when accessing cryptographic services.

4.1 Adding ECDSA to JDK 1.2

One of the first problems in implementing ECDSA is that it is not one of the standard algorithms in JDK 1.2. Consequently, there are no interface definitions for it. In fact, there are no definitions for any elliptic curve cryptosystems in JDK 1.2. This is an omission, as clearly defined interfaces facilitate the inclusion of alternative implementations later.

To solve this problem, we have designed a set of interfaces, which are shown in figure 3. These interfaces have been modelled after the DSA interfaces provided with JDK 1.2. They define the functionality of elliptic curve parameters that are used with the related implementations of cryptographic engines. We propose that they could be included in JDK 1.2.

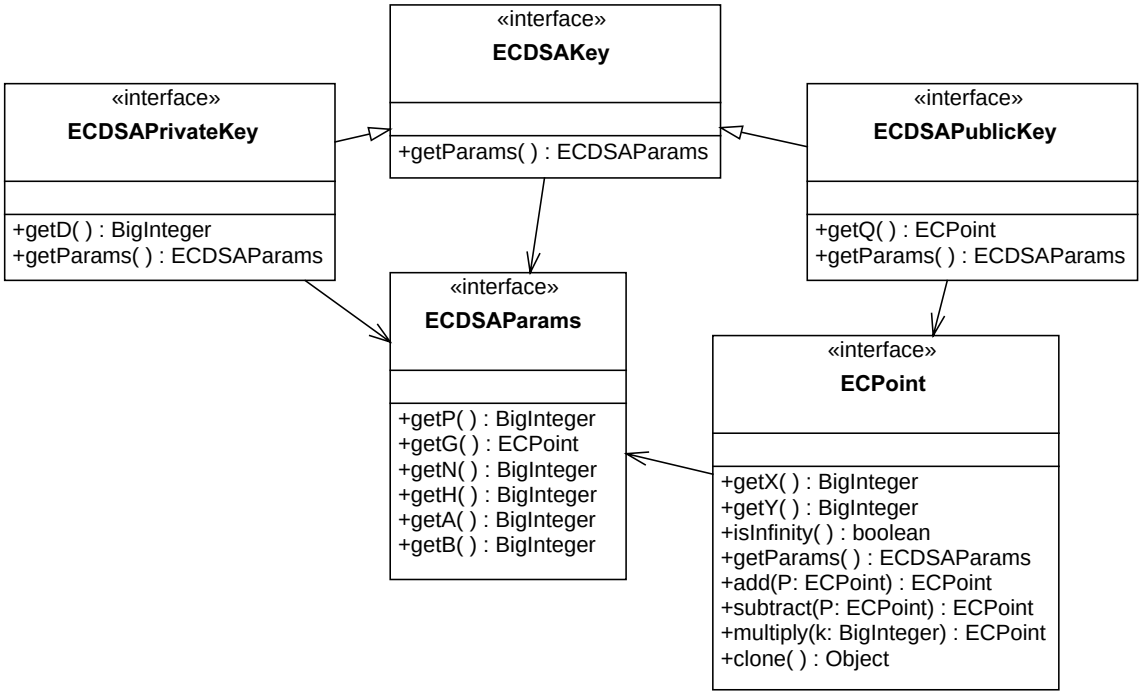

Figure 3: The ECDSA interfaces

The ECDSAParams interface uniquely identifies the elliptic curve used as well as the generator point.

The ECPoint interface is used to represent a point on the elliptic curve. This interface defines an immutable class that contains all the necessary operations for calculations with points.

The ECDSAKey class is used only for type checking. The ECDSAPublicKey and the ECDSAPrivateKey are then used for the actual key pair.

No definitions for the engine classes of ECDSA (key pair generator and signature) are required, as they have already been defined by the JDK 1.2 Cryptography Architecture. There is an abstract class with all required method definitions for each engine class. All that is left is to inherit the engine class and implement the functionality.

On top of all these engine classes, a master class is required to register these services to the runtime environment. This master class has to be inherited from the Provider class in the Security package.

5. Implementing an Elliptic Curve Cryptography Provider in JDK 1.2

In our project we implemented the Elliptic Curve Digital Signature Algorithm (ECDSA). The signature algorithm and all necessary operations were defined in IEEE P1363 and ANSI X9.62 –standards.

5.1 Implementation choices

One implementation choice is the one regarding field on which the curve is to be implemented. The most common options are F(q) and F(2^m). Of these two, the prime field is considered to be slightly more suitable for software implementation, whereas the field F(2^m) should be more suitable for hardware implementations[3]. Therefore, our implementation has been based on the prime field. The field, as well as all the parameters for the curve and generator

point have been taken from the ANSI standard[5]. We are using the 192-bit, 239-bit and 256-bit prime fields. These should result in signatures more secure than the minimum security level mandated by the ECDSA draft, which is a 160-bit prime. Further, they have been used to estimate the performance of a 160-but field.

Another implementation choice is the presentation of point coordinates. It is possible to use either affine or projective coordinates. Projective coordinates can be more effective in very long key lengths, but the effectiveness naturally depends on the implementation. We are using affine coordinates, as our keys are not excessively long.

The mathematics have been implemented using the BigInteger class, which provides arbitrary precision arithmetics. Also, there are some modulo arithmetic related operations as well as primality checking. All these characteristics make BigInteger a good choice for a straight forward implementation like ours, but leave room for performance optimisation. BigInteger is an immutable class, so using it for calculations results in continuous creation of new objects, which can have a significant negative impact on the performance.

5.2 The class structure

Figure 4 shows all the classes used in this implementation. There is one implementation class for each of the interface classes (except for the ECDSAKey class, which is used only for type checking). These provide the functionality defined in the interfaces as well as some methods used for validating the parameters.

Further, there are the ECDSASignature and ECDSAKeyPairGenerator classes that have been inherited from the corresponding abstract classes (SignatureSPI and

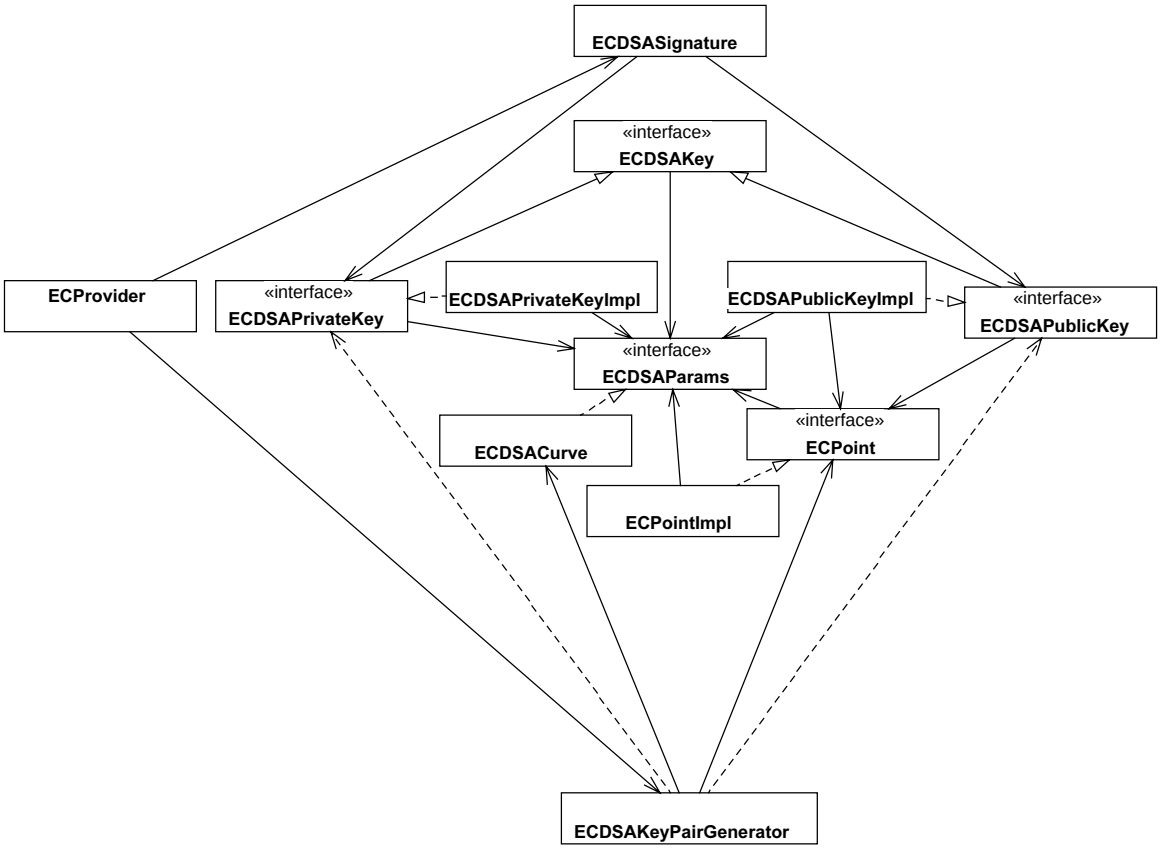

Figure 4: The complete class structure

KeyPairGeneratorSPI in the Security package). The signature class can be used to sign and to verify signatures, and the key pair generator class can be used to create the key pairs used for signing and verifying. Before the signature class can be used, it has to be initialised by passing it the relevant key. At this stage, the key is verified to make sure it is valid. This is particularly important with the public keys, which could arrive from anywhere in the network.

Finally, there is the ECProvider that has been used to register the new engines. It has been modelled after the Provider example in JDK 1.2.

The complete implementation consists of roughly 750 lines of code (empty and comment lines have been excluded from this number).

5.3 Performance

The performance of the ECDSA implementation was compared to the DSA implementation provided by Sun. This implementation comes standard with the JDK 1.2. The results have been summarised in table 1. The numbers were obtained by calculating the averages of 100 iterations of each operation at three different field sizes, extrapolating to a field size matching DSA security and then normalising the running time results to the DSA figures.

Table 1 : Comparison of the relative performances of DSA and ECDSA implementations

Operation	DSA	ECDSA
Key pair generation	1	6
Signing	1	6
Verifying	2	12

As can be seen from the data, our non-optimised implementation is roughly 6 times slower than the Sun DSA implementation. It should be possible to improve the performance significantly by using more optimised algorithms for elliptic curve operations. Another possibility would be to implement the mathematics using long-type variables instead of BigInteger-type. As BigInteger is an immutable class, each operation (key pair generation, signing and verifying) requires the creation of thousands of objects, which can cause a severe performance penalty. Long, however is a basic type and could be reused thus avoiding most of the object creations.

A useful characteristics of ECDSA can be seen from table 2, which contains the original measurement data (again, normalised to DSA figures). The performance of ECDSA scales linearly with the key length, even up to key lengths not provided by DSA. These key lengths can provide security, which in the current understanding is unbreakable in the foreseeable future. A field size of 256 bits is security wise comparable to a symmetric key of 128 bits or an RSA/DSA key of 2304 bits [5, 10].

Table 2 : The running times of ECDSA at different field lengths (in bits) compared to the performance of 1024-bit DSA

	Field length		
Operation	192	239	256
Key pair generation	7,2	11,9	13,6
Signing	7,2	11,9	13,7
Verifying	7,4	11,9	13,7

Another convenient characteristic of our implementation of ECDSA is that it uses a predefined curve for all its operation. Therefore, even the first key generation is as fast as the subsequent ones. The Sun DSA implementation, however, has to create the primes required for the keys during the first key generation. The execution time of prime generation varies considerably, but it is not uncommon to experience a prime generation hundreds of times slower than the actual key generation. Therefore, if the total number of keys cre-

ated is very small, it is possible that the total time for creating key pairs and signing documents is already shorter with the current ECDSA implementation than with DSA.

6. Further ideas

One of the reasons for implementing elliptic curve cryptographic systems was the possibility of increased speed compared to other cryptographic systems. Our implementation is based on BigInteger-class, which is flexible and provides some of the required functionality. However, this flexibility comes at the cost of speed.

Similar functionality could be achieved by implementing a specifically designed class in C. This would improve performance, but would sacrifice portability.

Still another possibility would be to implement the mathematics in Java using enough many variables of the type long for each number. This approach is suitable only if the keys used are sufficiently short, but could result in faster implementation, particularly on platform with native support for 64-bit integers.

Even shorter key lengths could be achieved by switching to hyperelliptic curves. They are more complex in nature, and thus provide the same security with shorter key lengths.[9] The downside is that even less is known about their security than about the security of regular elliptic curves. Hence, they could become susceptible to a new attack.

The use of short keys opens up completely new application areas for cryptography. It would be feasible to implement a smart card capable of generating key pairs and signing and verifying signatures in a reasonable time by implementing the relevant mathematics in hardware. This smart card could then be used for many applications, like electronic money: nano payments would be feasible while still maintaining anonymity – something worth cherishing in this time of electronic surveillance.

7. Conclusions

In this paper, we have discussed the need for a new encryption system for JDK 1.2. Java, as a platform, is not as fast as C, so the choice of cryptographic algorithm is of importance.

Elliptic curves have been used for cryptography since the mid of 1980's, but are yet to gain mass success. However, they have several tempting features: they provide shorter key lengths than competing public key systems, the shorter keys can result in faster implementation and, finally, they are based on a separate mathematical problem from RSA, which could prove valuable should RSA become more vulnerable to attacks in the future.

We are proposing that the Java Cryptography Architecture should be extended to include ECDSA as a standard algorithm and propose a possible set of interfaces to be included in JDK 1.2 to facilitate its implementation.

Further, we have discussed our implementation of a signature service in JDK 1.2 based on the ECDSA standards. This system has successfully been used as a part of our access control system based on authorisation certificates.

Finally, we have presented ideas for further research in this field.

References

[1] T. Halfhill, *How to Soup Up Java, Part 1*, Byte, May 1998.

[2] P. Nikander and J. Partanen, *Distributed Policy Management for Java 1.2*, Proceeding of the 1999 Network and Distributed System Security Symposium, San Diego, CA, Internet Society, Reston, VA, February 1999.

[3] IEEE Draft version 5 of P1363: Standard Specifications For Public Key Cryptography, July 1998.

[4] Elliptic Curve Cryptography, http://world.std.com/~dpj/elliptic.html.

[5] ANSI Working Draft of X9.62-1998, Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)©, April 1998.

[6] A. Menezes, T. Okamoto and S. Vanstone, *Reducing elliptic curve logarithms to logarithms in a finite field*, IEEE Transactions on Information Theory, 39 (1993), 1639-1646.

[7] *Java Cryptography Architecture API Specification & Reference*, http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html, Sun Microsystems, March 1998.

[8] The Java API documentation, Sun Microsystems, July 1998.

[9] Y. Sakai, K. Sakurai and H. Ishizuka, *Secure Hyperelliptic cryptosystems and their performance*, Proceedings of the 1998 International Workshop on Practice and Theory in Public Key Cryptography (PKC'98).

[10] B. Schneier, *Applied Cryptography, 2nd edition*, John Wiley & Sons, 1996.