DECENTRALIZED AUTHORIZATION WITH ECDSA ON A JAVA SMART CARD

A Software Implementation

Tommi Elo and Pekka Nikander
Helsinki University of Technology
{tommi.elo, pekka.nikander}@hut.fi

Abstract Traditionally, smart cards have been used as secure tokens in identity based access control. That is, a smart card has been used as an intelligent storage of protected cryptographic information, such as a shared secret or a private key in a public key system. The cryptographic information is then used to prove the possession of the card in a secure way either locally or remotely over telecommunication links. In this paper we present a basis for another type of use for smart cards, where smart cards are not used as identification tokens but as authorization tokens. Our approach is based on SPKI-like authorization certificates along with ECDSA based public key cryptography. The ECDSA algorithms provide us the benefits of smaller key sizes, potentially better running times in software-only implementations, and the possibility to create new key pairs on the card in a reasonable time. The latter feature can be used, as we show, to provide additional protection to the user in the form of enhanced privacy. Our current prototype implementation uses the Java Card specification, and we also compare our card implementation with an earlier ECDSA implementation written for a workstation environment.

Keywords: Java Card, Elliptic Curves, digital signatures, ECDSA, public key cryptography, trust management, authorization certificates, SPKI

1. INTRODUCTION

Decentralized trust management and authorization, as promoted by a number of approaches including the PolicyMaker [1], SPKI [2],[3], and TeSSA [4], are based on public key cryptography and semantically rich authorization certificates. All of these approaches are based on the assumption that the acting principals have a secure way of storing any private keys in their possession. However, the problem of the securing

cryptographic private keys has been, and continues to be, a formidable challenge in practical applications of public key cryptography. Fore, in general, security is only as strong as the weakest link in the chain, and keeping private keys truly private presents challenges that cryptography alone cannot solve. In the end, there always seems to be a need for secure storage, whether it's a password in ones mind or a private key locked in a safe. One established solution is to use smart cards as secure storage media.

Elliptic curve cryptography (ECC) has some advantages over more established public key techniques that makes it particularly interesting for smart card software implementations. Mainly, the much shorter key lengths in contrast to traditional methods help in squeezing the cryptosystem to the limited environment of the current cards. Because smart cards are not very fast, the shorter key length and resulting faster running times also favour ECC over finite field public key algorithms. As an additional benefit, the operations required to create new key pairs in ECC based public key systems are computationally cheap when compared, e.g., to the effort needed to create new RSA key pairs. This provides some additional benefits with respect to authorization certificate systems, since systems based on direct authorization tend to use more keys and have shorter key lifespan than more traditional identity based public key infrastructures.

Our aim in the presented work is to provide a smart card platform for authorization architectures that use certificates. A smart card can be used to securely store private keys needed by the authorization certificates. A smart card with an Elliptic Curve Digital Signature Algorithm (ECDSA) implementation can also be used for secure creation of new keys and certificates, and for checking the validity of previously created certificates.

1.1. CRYPTOGRAPHY IN ACCESS CONTROL

The traditional approach to access control is based on a two step process, consisting of strong identification and access right check. A user is first identified by checking that he or she possesses and controls the private key corresponding to a certified public key. This can be done by a challenge-response protocol or be part of a more sophisticated authentication of authorization management protocol. A successful identification results in a name, which is supposed to represent the identity of the user. After the identity authentication check an access control list (ACL) is consulted to see if the operation is allowed for that name. These trust

relationships, along with other bindings pertaining to a typical identity certificate system, are illustrated in Fig. 1. In addition to those relationships already mentioned, there is one more that we want to emphasise, i.e., the link between a person and that person's name. This link is not cryptographic in nature, and is a bit problematic since different persons can have exactly the same names, and it is not necessarily clear what we mean with a name in a distributed setting (see, e.g., [2], [3], [5], [6]).

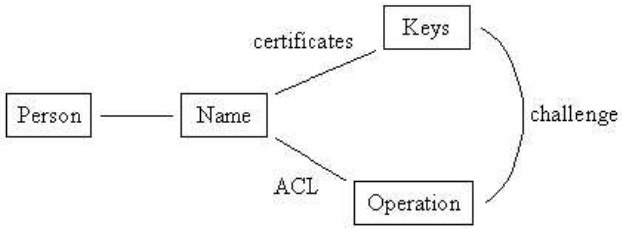

Figure 1 Trust relationships when using identity certificates for access control [5].

1.1.1 Authorization certificates authorize without identification. It is rarely necessary to identify a user in order to make access-control decisions. In authorization certificates, names can be avoided altogether. In Fig. 2, the relationships between entities in an authorization certificate based access control setting are explained. The authorizing party signs a document (certificate) that describes what the possessor of a certain private key can do. Thus, authorization certificates bind operations directly to keys.

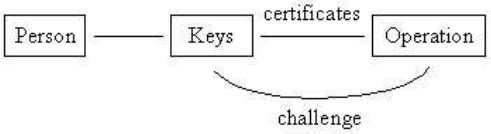

Figure 2 Bindings in the authorization certificate [5].

By employing authorization certificates, it is possible to make access-control decisions that do not require any knowledge of the name or identity[1] of the user. By presenting a certificate and proving possession of

[1] I.e, identity other than the public key, which may be considered as an identity (or name) itself.

the related private key, the user can convince the issuer of the certificate that he or she has the requested right to access the indicated resource. Because it is possible for the user to posses several key pairs, and even create temporary key pairs on the fly, the use of one key cannot be easily related to use of other keys. When desired, this property can be used to build anonymity in the core functionality of the authorization.

1.1.2 Delegation of rights. For the rightful owner, it is usually possible to transfer the rights granted by a certain certificate to some other party. This process is called delegation. In order to be controllable, delegation must be enabled by the authority granting the original certificate, otherwise a certificate is not delegable. This means that an authority can choose either to trust or not to trust the redistribution of rights to a subject. This is, of course, a very different process than copying rights, since the authority can, by checking the delegation chain, make sure that the chain is valid. It might choose to perform additional checks to make sure, for example, that no unwanted parties are involved. It might also check that chain is not too long, and accept it only if it is shorter than some safety length. These additional checks require us trust to the reduction policies of our associates, however.

1.1.3 SPKI authorization certificates. One proposal that uses authorization certificates is SPKI, which is under development in the IETF [2], [3]. An SPKI certificate consists of fields. From the security point of view, five of the certificate fields are important: issuer, subject, delegation, authority, and validity. Formally, an SPKI certificate can be represented as a five-tuple (I, S, D, A, V), where I is the public key of the issuer and S is the public key of the subject for which the rights are given. D is the delegation bit, which, if true, states that the authorization can be further delegated. A describes the authority granted; it is also called tag. Thus, the tag contains the description of the delegated rights. V is the validity field, and may describe a network location where the validity of the certificate can be checked or a period of time during which the certificate is considered valid. Also other types of validity conditions are possible.

1.1.4 Certificate Reduction Certificates. Whenever SPKI certificates are used in any typical setting, they tend to form certificate chains. Basically, a chain of SPKI certificates is an ordered collection of certificates in such a way that the subject of a preceding certificate is identical to the issuer of the following certificate. That is, the grantee of one certificate further grants (some of) its rights by creating another

certificate. If the security policy of the first issuer of such a chain allows, the chain may be replaced with a single certificate, called Certificate Reduction Certificate (CRC). For example, given two certificates (I_1, S_1, $D = true$, A_1, V_1) and (I_2, S_2, D_2, A_2, V_2), iff S_1 equals with I_2, then this certificate chain (of length two) can be reduced to a certificate (I_1, S_2, D_2, A, V) where $A = \bigcap(A_1, A_2)$ and $V = \bigcap(V_1, V_2)$ [3].

A Certificate Reduction Certificate server, or CRC-server for short, is an on-line certificate issuer that is willing to return CRCs in response to certificate chains sent to it. That is, when a CRC-server receives a certificate chain whose first certificate is signed by itself, it verifies the chain according to the SPKI chain verification rules, possibly augmented with its own local policy, and if the verification passes it issues a corresponding CRC and sends it back to the sender of the chain. CRC-servers can be used to enhance privacy.

1.2. ELLIPTIC CURVE CRYPTOGRAPHY ON A SMART CARD

Using elliptic curve cryptography (ECC) instead of more traditional approaches has some relevance when using standard low-cost smart cards. ECC offers smaller key sizes, and the implementations are therefore potentially faster. Smaller keys take less storage, which is important because smart cards have limited memory to be used. Also the faster key generation offered by the ECC makes the secure on-the-fly key generation practical. This is important especially in architectures that offer support for anonymity, which is based on the use of many key pairs.

On the other hand, using ECC has also some obvious drawbacks. For example, ECDLP, the problem on which ECC security is based on, has not been analysed as rigoriously as, e.g., the integer factorization problem, and this may cause some problems in the future. ECC mathematics are somewhat more advanced, and it has not been studied as long or as widely as traditional arithmetics. This makes the implementation harder and more error-prone. The implications of using elliptic curve cryptography instead of traditional DLP or IFP based problems can be summarized as presented in the Table 1.

1.3. PROJECT GOALS

In our project, we explore the possibility of implementing a digital signature algorithm, in software, to commercially available smart cards, and use this implementation for decentralized authorization and trust management. The work is based on a previous implementation of ECDSA that runs on Standard Java platform [8]. We have ported this imple-

ECC Benefits	ECC Drawbacks
Shorter key length	More complicated mathematics
Faster running time of the normal operations	Security of ECDLP has been analysed less than e.g. integer factorization
Smaller storage requirements	
Faster key generation	

Table 1 ECC on a smart card

mentation to a smart card environment and provided the underlying software functionality absent in the standard Java Card. The main goal has been to demonstrate feasibility of the software implementation. A secondary goal has been to research and evaluate a possibility of practical applications with respect to the speed of the routines.

1.4. ORGANIZATION OF THIS PAPER

The rest of this paper is organized as follows. In Sect. 2. we describe the Java Card architecture in general, and its security differences from standard Java. Sect. 3. describes the architecture of our implementation, and outlines the original implementation level goals we had when starting this work. In Sect. 4. we provide the actual details of the implementation, while in Sect. 5. we summarize the lessions attained so far. Sect. 6. outlines our current notions of future work. In the Appendix, we briefly outline the algorithms used in Elliptic Curve Cryptography.

2. JAVA CARD ARCHITECTURE

The Java Card architecture is a limited subset of Java. It defines its own API and virtual machine. The limitations of the current cards won't allow a full-blown virtual machine or other Standard Java capabilities. For that reason, the Java platform architecture was designed with different virtual machines for different purposes.

Java Card API 2.0 consists of the three fundamental packages: `javacard.framework`, `javacardx.framwork` and `javacardx.crypto`. The `javacard.framework` package provides a framework of classes and interfaces for the core functionality of a Java Card applet. The `JCSystem` class acts as a central point of execution. The `AID` class encapsulates

the ISO application identifier, which is used to uniquely match applets. The AID class has methods to get to the `Applet` instance, which all applets on the card inherit. An `APDU` object encapsulates an Application Protocol Data Unit (as defined by ISO), which is the communication protocol between the card and off card entities. Applets can use the services of the `ISO` and `Util` classes. The `ISO` class associates ISO-defined constants to their actual values, while the `Util` class contains various native utility functions such as `arraycopy()`. The `OwnerPIN` class can be used by the applets to query the status of the PIN recognition, and ask the user to authenticate. The `javacardx.framework` package provides classes and interfaces for ACLs, card operating systems, and other vendor specific extensions. The `javacardx.crypto` extension package contains the security classes and interfaces. It isn't necessarily available due to export restrictions.

2.1. JAVA CARD VS. STANDARD JAVA

Java, as it exists on the card, has many limitations that make the development rather different from that practised in other Java environments. Particularly, the lack of garbage collection affects the development process, since smart cards don't have too much memory even to begin with. The specification of Java Card won't allow the use of destructors either, so freeing the memory is not supported. This means that the memory that has been reserved stays that way until the given applet is deactivated. It also means some of the high level nature of Java is lost, since programmers must now constantly watch the usage of memory.

2.2. JAVA CARD SECURITY

Java Card security has not yet been analysed deeply in the literature, but some information is available [11]. Many of the changes from the standard Java specification are done in order to fit Java to the limited environment of the current cards, but these changes also have security implications. Some changes increase security risks while others lessen them. One thing that needs to be addressed is that much of the base security model in Standard Java is totally absent from Java Card [18].

2.2.1 Java Card properties increasing security. The following properties make the Java Card environment more secure than the standard Java environments are.

- *Lack of threads.* The security analysis becomes much easier without threads.

- *Absence of dynamic class loading.* If VM can be confused about the types of objects, the Java security model brakes.

2.2.2 Java Card properties decreasing security. The following properties make the Java Card environment less secure than the standard Java environments are [18].

- *Lack of garbage collection.* Memory leaks are a well known source of numerous security problems.
- *Exception propagation problems.* Uncaught exceptions could lead to a card becoming muted.
- *Multiple applications and applet firewalling.* Attacks between different vendor's applications is a risk.
- *Access to native code.* Any available native methods must be considered to be parts of the TCB.

3. IMPLEMENTATION ARCHITECTURE AND CRITERIA

As already mentioned, we used an existing Standard Java (workstation) ECDSA implementation as a basis for our present work. That implementation uses the JDK1.2 class `java.math.BigInteger` to provide all integer operations. However, that class is an immutable class, which means that every method call returns a new object that must later be garbage collected. This is clearly unsuitable for the card environment, so we implemented our own large integer class (`MutableLargeInteger`), which can reuse the allocated objects.

The card platform we used is Schlumberger Access with 16kB EEPROM. Its Java environment has software support for 16-bit type of Java (`short`) at longest, 32-bit integers (`int`) or 64-bit ones (`long`) are not available. For this reason that we could only use `short` integers in our `MutableLargeInteger` class. This card has no hardware crypto-acceleration, so all the operations are executed on the normal processor of the card.

3.1. ARITHMETICS AND CHOICES FOR THE FINITE FIELD

The implementation of ECC algorithms needs several levels of arithmetic. It makes sense to model these levels as a stack (Fig. 3). The target platform needs to be able to handle normal integer computations, such as bit-wise xor, bit-wise and, and shifting of bit patterns. Since the

register length is much smaller than the size of the bit patterns we operate with, we must be cautious on how the underlying hardware is used. Also the speed of the hardware in integer operations, such as addition and multiplication, is of importance in the making of implementation choices.

The finite field arithmetic operations are at the very low level on the call stack, as one can immediately see in Fig. 3. This means that the processor will likely spend a significant amount of time doing finite field arithmetic; thus, the efficiency of its implementation largely effects the efficiency of the whole architecture [18].

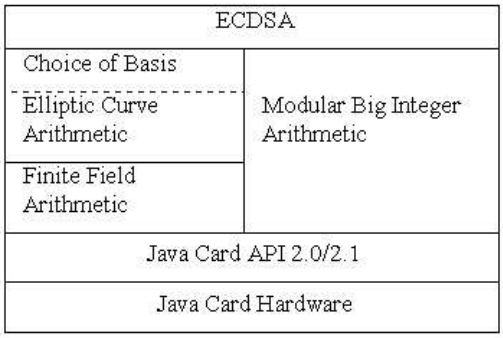

Figure 3 Levels of the arithmetic architectures in a smart card ECDSA.

In order to implement ECDSA, we need both modular big integer arithmetic and finite field arithmetic. While the pure elliptic curve operations can be built on top of many kinds of finite fields [19], two choices are most used: $\mathbf{F}_{2^m}$ or characteristic two fields, and $\mathbf{F}_p$ called the odd prime fields. $\mathbf{F}_{2^m}$ fields promise clear performance advantages on processors that have small registers and relatively unoptimized integer instructions with no access to special big integer hardware. Much of this promise of speed is based on the fact that operations can be implemented only by using basic bit operations such as and-ing and xor-ing, which are typically the fastest operations on any processor. It should be noted that these promises aren't necessarily realized when using cards with Java because there is no direct access to hardware from the pure Java code.

Finite field arithmetic can look quite different depending on what is the choice of the underlying field. In practice, we were faced with the following real alternatives for the purposes of our implementation.

- Odd prime field ($\mathbf{F}_p$)

- Characteristic two finite field with a polynomial base ($\mathbf{F}_{2^m}$)

Of these we chose the odd prime field largely for practical reasons. We had a possibility to use an existing debugged implementation, that was based on the odd prime finite field arithmetic. That existing code base and architecture would not have been of much use, had we implemented ECDSA with a different underlying field. Otherwise the EC arithmetic operations used in the elliptic curve layer would have had to be largely rewritten. This is because they have direct interface connectivity with the lower finite field layer and hence directly call its services.

3.2. PERFORMANCE SENSITIVITY OF THE LAYERS

We assume that the performance of the whole architecture depends on the arithmetics roughly in a level order: the lower the part is in the stack in Fig. 3, the more performance specific it is. As we are concentrating on software, the two lowest parts, namely Java Card HW and its implementation of the Java Card API, are not a major concern for us. Also there is little data of different Java Card manufacturers environments and architectures, that would address their performance differences. The performance comparison would need to develop some test software and run it on different platforms to address the differences in speed. In the future, we intend to use our software also for this purpose.

3.3. THE WORKSTATION ECDSA

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 a class that contains all the needed operations for calculations with points. The `ECDSAPublicKey` and the `ECDSAPrivateKey` are then used as an actual key pair.

For each interface there is a corresponding implementation class. The `ECDSASignature` and `ECDSAKeyPairGenerator` classes have been inherited from the corresponding abstract classes of Java in the `Security` package. The implementation classes like the ones mentioned above are registered in Java and they perform the actual cryptographic operations. Finally, there is the `ECProvider` that has been used to register the new engines so that they can be called through the provider mechanism. As can be seen, much of the structure in the workstation implementation is actually devoted to the provider functionality, that is only an addi-

tional layer for using the ECDSA. Unfortunately, the Java Card API doesn't offer same kind of support for cryptoservices as standard Java. Most significantly, there is no support for providers. For this reason, we stripped off this functionality in the card implementation.

3.4. PLANNING THE IMPLEMENTATION

We started with the assumption that the underlying code should be used if possible, since it represented well debugged and finished code. This would lead to code reuse, which could help in the implementation.

The goals of the implementation were defined as follows.

- **The main goal** is a working implementation of the ECDSA to some of the current Java capable cards, which can support at least 160-bit but preferably higher key length.

- **A secondary goal** is to achieve a performance, that will result in 5-10 second signature operations. And further, a 5-10 second performance of the key generation when using a preset curve.

Commercial applications would require sub-second operations. However, since one of our primary concerns is to evaluate if the implementation is possible using Java, a five to ten fold increase is justifiable, as native low level code can be substantially faster than Java Card code. Thus, the implementation that achieves these goals could be viewed as a proof of concept that commercial quality software cryptography is feasible with the current Java capable cards. It would also provide a case of a successful use of Java Card programming environment in demanding application development.

The motivation for our secondary goal is to be able to create new keys in a time window which is not too wide for most applications. Achieving this would make it possible to create new key pairs on the fly. This would enable us to better support anonymity, an advantage provided by SPKI-like authorization architectures. We can, for example, delegate the authority given to us to another key pair, created inside the card. That is, in doing so we are actually making a certificate chain longer ourselves. We could then give this chain to a public CRC-server, which reduces certificate chains, and the resulting certificate would only reveal the last key in the chain; the one that was generated by us. Now, we could use this certificate publicly and only the CRC-server would know the original key pair that was issued to us.

4. THE CARD IMPLEMENTATION

As described earlier, and depicted in Fig. 3 on page 9, the actual implementation is built using a layer like architecture. At the very bottom, we have the Java Card implementation and API provided by the card manufacturer. On the top of that lie the basic classes providing the needed big integer and other final field arithmetic operations. In our case, as we are using the odd prime field $\mathbf{F}_p$, the big integer and finite field operations are almost identical, thereby allowing us to save space on the card. The implementation issues involved with these low layer operations are explored in detail in Sect. 4.1, while Sect. 4.1.2 and Sect. 4.2. concentrate on the upper layers.

4.1. MUTABLE CLASSES IN OBJECT ORIENTED PROGRAMMING

In Java, the immutability of many classes makes it easier to achieve information hiding principles, which are central ideas in object oriented programming in general. On the other hand, because on the Java Card we only have limited memory and no garbage collector, we have to explicitly handle the reuse of memory. While the mutability itself need not be in direct contrast with the OOP principles, when combined with the need to preserve memory some problems arise.

Now, when there is no way to explicitly reclaim used parts of memory and no garbage collector to do this automatically, any allocated memory must be manually reused by carefully designing the number and usage of temporary objects. This leads to a situation were the code must use the temporary objects in such a way that no side effects occur [18].

4.1.1 Implementation of big integer arithmetics. The actual implementation of the big integer class `MutableLargeInteger` uses half register arithmetic, ported from C to Java [20]. In the half register arithmetic we usually use only half of the bits that fit to a machine register. In our case, a register is a Java `short` -type integer. On the other hand, we plan to experiment also with an implementation that uses full registers instead of half ones.

As already mentioned, our `MutableLargeInteger` is a mutable class. The mutability of the underlying big integer class is very important because that is what makes it possible to reuse objects. This is important to achieve savings in memory usage, which in turn is important because no memory that has been allocated can be freed in Java Card. We have tried to allocate all our temporary objects as static variables. This way, when the program executes, the needed number of temporary objects are

reserved only once for every class and not for every instance as would otherwise be the case.

4.1.2 The ported part of the implementation. The next layer of the ECDSA card implementation consists of the workstation ECDSA implementation with the Java 2 API specific provider mechanism removed. The interfaces themselves are unchanged, but in the card version we have only one implementation class corresponding each of the interfaces. The implementation classes have been modified to use as much static temporary `MutableLargerInteger` variables as possible. The useage of these instances was optimized using the register allocation like reuse algorithm.

4.2. USING CERTIFICATES WITH ON-THE-CARD KEY CREATION

As was already mentioned in Sect. 3.4, it is beneficial to be able create key pairs directly on the card. If all the authorization tokens in the architecture support generation of new keys, it is easier to support anonymity. The ability to create temporary keys fast enough makes it possible to create key pairs on-the-fly basis. When the new key pairs are created on the card, the public key needs to be exported from the card. We might need to be able to send it securely to a public key server, for example. If the computer and the reader we are using are trusted, the task is easy, but usually the hardware the smart cards are connected to are at least partially untrusted. In the case of untrusted hardware, we can create certificates on the card. The public key is then placed in the certificate, which is protected against modification by the cryptographic signature.

According to our initial analysis, the size of the typical ECDSA signed SPKI certificate is under 300 bytes in size. This consists roughly of the two public keys of the issuer and subject (256 bits each), a signature (160 bits), a hash of the signature (160 bits), and additional headers (about 100 bytes). We must also include the size of the validity and tag fields; the size is also a function of the those. The more complex the authority, which is written in the tag field, the longer the certificate becomes. In most cases, the validity field contains only the time limits under which the certificate is valid (30 bytes). Correspondingly, even a cleartext tag-field can be quite simple (e.g. 100 bytes). If RSA keys offering the corresponding level of security were used, only the two keys combined would take more than 500 bytes.

5. EVALUATION AND LESSONS LEARNED

As we indicated in Sect. 1.3. on page 5, our goal was to evaluate the feasibility of implementing ECDSA on commercially available Java Card environments, to study to what extend the existing ECDSA implementation that was written for a workstation implementation could be reused in this project, and to provide smart card support for decentralized authorization systems. While our implementation is still progressing towards its final stages, and the actual implementation is not yet fully optimized, a number of interesting results and observations may readily be stated.

5.1. PERFORMANCE DATA

Table 2 compares the performance of our `MutableLargeInteger` class with the built in JDK 1.2 `BigInteger` in the workstation environment. The benchmarked version of `MutableLargeInteger` uses only card compatible datatypes. No variables are defined in the local scope and the number of temporary objects has been minimized using the register allocation approach.

The figures in the Table 3 tell us that the performance of the current `MutableLargeInteger` prototype on the card leaves room for improvement. As multiplication and inversion are the basic steps of the ECDSA algorithm, we can readily tell something about the performance of the

	JDK1.2 `BigInteger`	`MutableLargeInteger`
1000 multiplications	$\ll$ 0.1 sec.	0.3 sec.
1000 additions	$\ll$ 0.1 sec.	$\ll$ 0.1 sec.
1000 inversions	1 sec.	44 sec.

Table 2 Performance comparison of the different big integer implementations with the number length of 192 bits.

	50 bits	**100 bits**	**192 bits**
Multiplication	15 sec.	39 sec.	137 sec.
Addition	0.7 sec.	2 sec.	6 sec.
Inversions	370 sec.	N/A	N/A

Table 3 Performance comparison of the different big integer implementations with the number length of 192 bits.

whole ECDSA prototype. For example, 192-bit ECDSA needs about 30000 inversions during the signature operation and about half of that for the key generation or checking of the signature.

Looking at figures in the Table 2, we can deduce that inversion operation of our current prototype is roughly 50 times as slow as with the JDK 1.2 implementation of `BigInteger`. The figures are not entirely comparable as JDK implementation uses native methods. It is interesting to note that multiplication is about 30 times slower on our implementation than with JDK 1.2 implementation. We know that our implementation of the inversion routine is not fully optimized, but we presume it is still hard to make it orders of magnitude faster. Our multiplication routine on the other hand is the simplest possible and far from optimal. Multiplication is also heavily used to implement inversion. This would suggest that by better optimizing our multiplication routine, we could also get a much faster inversion.

This optimization alone could give us a ten to fifty fold increase in the speed of the basic finite field operations, which would elevate their performance from hundreds of seconds to seconds or tens of seconds. Unfortunately, ECDSA uses these basic field operations very many times, which suggests that we are still quite far from achieving total performance figures in the tens of seconds class.

5.2. THE JAVA CARD ENVIRONMENT

When considering the Java Card environment, we were faced with a number of dissimilarities that made it relatively hard to apply our previous knowledge of standard Java to the card environment. First, the lack of garbage collection, and any other facilities that would allow direct memory reuse, makes a huge difference between the Java Card environment and any of the other Java environments. With this difference, the whole nature of the software development process changes. For example, instead of the design of UML class diagrams, and therefore the relationships and associations between classes, the focus should be towards estimating the needed memory consumption, which means runtime objects become much more important.

Second, security considerations are almost totally different. The limited execution environment with the much more limited Java Card applet security approach, are behind the basic differences. However, the underlying trust assumptions may make a much bigger difference, depending on the security requirements of the actual application. Basically, the Java Card environment itself must be considered trusted.

Also an interesting point to consider is the case of porting workstation Java code to the card environment. Maybe the most important lession here is that while we were able to preserve the structure of the interfaces, most part of the code needed revisions due to the semantical differences (mutability vs. immutablity). Thus, according to our experience so far, it seems almost inevitable that the write-once-run-everywhere principle, which Sun is touting for Java, definitely does not apply to the Java Card environment. Code written for other Java environments is almost sure not to run without changes, due to the limited nature of Java Card.

To summarize and to further illustrate our Java related experiences, it is instructive to consider the design choices faced when developing our `MutableLargeInteger` class. Basically, we faced the problem of recycling and reusing instance objects; a case of designing object structures. Our current implementation is carefully hand crafted; each algorithm is designed to run in isolation, and to use a minimum number of `MutableLargeInteger` instances. The whole reuse issue is a well known example of the so called register allocation problem usually associated with compiler design. Compilers need to use the registers of the target computer efficiently. Much the same way, we are forced to the reuse of already created objects to get around the memory limitations of Java Card environment. We envisage that the Java compiler could be extended to handle `BigInteger` instances in a same way the compiler already handles `String` intances, and to perform register allocations as a compile time process. In such a case, the underlying implementation might well use mutable objects while the compiler would preserve the illustration of immutable values.

An alternative to the compiler based register allocation approach could be implemented with reference counting and primitive finalization. While the current Java Card architecture supports no memory management whatsoever, it would not be too hard to add primitive reference counting to the environment. This could resemble, for example, the `java.lang.ref.SoftReference` approach. A reference counting approach would allow the finalize method to be called whenever there are no more active references to an object. That method could then return the object to a pool of reusable objects.

Thus, at first glance, it seems that mixing of Java and smart cards may not be on a very solid foundation from the serious programming point of view. Basically, Java is a great language to program with, but if the programmer has very limited amount of available memory, which will always be the case with smart cards, one is forced to reuse objects much the same way that one saves registers in assembly language programming. This is not very convenient, and we argue that some

modifications would be beneficial if Java is to gain more popularity in smart card programming. Enhancing the compilers or even providing some partial form of garbage collection may ease the situation while not requiring a full blown garbage collection mechanism.

5.3. ECC AND DECENTRALIZED AUTHORIZATION

According to our initial evaluations, ECC based keys and certificates seem to offer a number of benefits over more traditional approaches when considering smart cards for decentralized authorization. Although at this point the performance of the implementation leaves room for improvement, it seems that a pure Java software implementation, no matter how optimized, is not yet sufficient on the current Java cards with no special crypto hardware. However, the order-of-magnitude shorter key length in ECDSA makes it possible to store a much larger number of key pairs and certificates on a card than when using e.g. RSA. Furthermore, since the key generation is quite fast in comparision with RSA, it is feasible to create new key pairs within the card, on the condition that the performance of the more basic operations can be made acceptable. Since the card itself must be assumed trusted, the security of the key pairs created on the card may be considered quite good. With the use of suitable Certificate-Reduction-Certificate servers, such keys can be effectively used to provide controlled anonymity, thereby enhancing privacy in the overall system.

6. FUTURE WORK

In the near future, we expect to complete a better optimized version of our implementation. Also an empirical performance comparison of the $\mathbf{F}_p$ and $\mathbf{F}_{2^m}$ in the Java environment would be important as it would clarify how directly traditional performance evaluations can be generalized to Java environments. It has been hypothesized that an optimal normal basis version would be faster than the big integer based one [21], [22], [23]; however, it is not clear weather the optimal normal basis would provide better performance in case of smart cards using Java, since native processor instructions are not readily available in a pure Java environment. Therefore, the argumentation in [17] is not necessarily valid. Despite our continuous efforts to find empirically validated comparisons between these architectures, we have been unable to find any published work of such nature. It would be important to empirically compare these two quite different implementation options, as they are both equally included in at least two EC standards [13], [14].

Appendix: ECDSA algorithms

ECDSA is the EC analogue of the more widely used DSA [12], [13].

Key generation

1 Select an elliptic curve $E(\mathbf{F}_p)$ so that the number of points in it is divisible by a large prime n.

2 Select a point $P \in E(\mathbf{F}_p)$ of order n.

3 Select a cryptographically strong random number (integer) d in the interval $[1, n-1]$.

4 Compute $Q = dP$.

5 The public key is (E, P, n, Q). The corresponding private key is d. Here E is the elliptic curve used, P is the chosen point on that curve, and Q is the public key point.

Signing a message

To sign a message m, the following algorithm applies.

1 Select a cryptographically strong random number k in the interval $[1, n-1]$.

2 Compute $kP = (x_1, y_1)$ and $r = x_1 \bmod n$.

3 Compute $k^{-1} \bmod n$.

4 Compute $s = k^{-1}(h(m) + dr) \bmod n$, where h is the Secure Hash Algorithm (SHA-1).

5 If $s = 0$ then go to step 1. (If then does not exist.)

6 The Signature for m is (r, s).

Verification of signature

1 Obtain the signers public key (E, P, n, Q) securely. Verify that and are in r and s are in the interval $[1, n-1]$.

2 Compute $w = s_1 \bmod n$ and $h(m)$.

3 Compute $u_1 = h(m)w \bmod n$ and $u_2 = rw \bmod n$.

4 Compute $u_1P + u_2Q = (x_0, y_0)$ and $v = x_0 \bmod n$.

5 Accept the signature if $v = r$.

References

[1] M. Blaze, J. Feigenbaum, and J. Lacy, *Decentralized Trust Management.* In Proceedings of the 1996 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, May 1996.

[2] Carl Ellison, *SPKI Requirements, RFC2692.* September 1999.

[3] Carl Ellison, *SPKI Certificate Theory, RFC2693.* September 1999.

[4] Sanna Liimatainen et al., *Telecommunications Software Security Architecture.* Helsinki University of Technology, Available from URL: `http://www.tcm.hut.fi/Research/TeSSA`.

[5] Ilari Lehti, Pekka Nikander, *Certifying Trust.* Practice and Theory in Public Key Cryptography (PKC'98).

[6] Pekka Nikander, *An Architecture for Authorization and Delegation in Distributed Object-Oriented Agent Systems.* Helsinki University of Technology, Doctoral Dissertation, 1999.

[7] Tage Stabell-Kulø, Ronny Arild and Per Harald Myrvang, *Providing authentication to messages signed with a smart card in hostile environments.* USENIX Workshop on Smartcard Technology, May 1999, p. 93-99.

[8] Yki Kortesniemi, *Implementing Elliptic Curve Cryptosystems in Java 1.2.* NordSec 1998.

[9] *Cyberflex Access Programmer´s Guide.* Schlumberger, 1998

[10] *Java Card 2.1 Application Programming Interface.* Sun Microsystems, Inc., February 1999.

[11] Gary McGraw, Edward W. Felten, *Securing Java.* Wiley Computers Publishing, John Wiley & Sons, Inc.

[12] Don B. Johnson, Alfred J. Menezes, *Elliptic Curve DSA: an Enhanced DSA.* Certicom ECC Whitepapers.

[13] IEEE unapproved standards Draft, *IEEE P1363 / D10 (Draft Version 10) Standard Spesifications for Public Key Cryptography.* July 16, 1999.

[14] American National Standards Institute (X9 Committee), American Bankers Association, *Working Draft, AMERICAN NATIONAL STANDARD X9.62.1998 Public Key Cryptography for The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).* September 20, 1998.

[15] Bruce Schneier, *Applied Cryptography: Protocols Algorithms and Source Code in C, Second Edition.* John Wiley & Sons, Inc., 1996, p. 160.

[16] Arjen K. Lenstra, Eric R. Verheul, *Selecting Cryptographic Key Sizes.* The 3rd workshop on Elliptic Curve Cryptography (ECC'99), October 27, 1999.

[17] Aleksandr Jurisic and Alfred J. Menezes, *Elliptic Curves and Cryptography.* Certicom ECC Whitepaper.

[18] Tommi Elo, *A Software Implementation of ECDSA on a Java Smart Card.* Master's Thesis, Helsinki University of Technology, March 2000.

[19] Christof Paar, *Implementation Options for Finite Field Arithmetic for Elliptic Curve Cryptosystems.* A Slide set of a key note speak, The 3rd workshop on Elliptic Curve Cryptography, October 1999.

[20] Michael Rosing, *Implementing Elliptic Curve Cryptography.* Manning Publications Co., 1998.

[21] *The Elliptic Curve Cryptosystem for Smart Cards.* Certicom ECC Whitepapers, May 1998.

[22] Thomas W. Hungerford, *Algebra, Graduate texts in mathematics.* Springer Verlag New York Inc, 1974.

[23] Lasse Leskelä, *Implementing Arithmetic for Elliptic Curve Cryptosystems.* Master's Thesis, Helsinki University of Technology, January 1999.