Lessons learned on implementing ECDSA on a Java smart card

Tommi Elo
Department of Computer Science
Helsinki University of Technology
P.O. Box 9700
02150 HUT Finland
tommi.elo@hut.fi

Abstract

Many companies have designed and manufactured smart cards, which vary greatly in both the hardware they use and software development environments they provide. Java Card promises to make smart card programming easier, by introducing a common programming language and run–time environment. Also as a member of the Java family, Java Card raises up hopes of easy software portability from PCs to smart cards. Our work demonstrates, that this promise is not fulfilled in the majority of the cases because of different memory models. The portability of code from other JVMs to Java Card is an important issue, which requires, among other things, a systematic method of conserving memory on the card. In our work we developed such an approach by constructing variable interference graphs and graph coloring them to minimize the number of temporary variables. The concrete method is a modified register allocation approach, which has traditionally been used in compiler design. This is followed by numerical performance data of the actual prototype along with the analysis of that data.

Keywords: ECDSA, ECC, cryptography, JVM, Java, smart card, software engineering, porting

1 Introduction

Smart cards are often used as trusted storage and data processing systems to store cryptographic private keys and other valuable information. This means that they are usually used as part of a larger access control or authorization architecture, which are becoming more commonplace. As a results of this, both the usage of smart cards and the corresponding development environments have greatly expanded since their introduction. Java Card promises to be a common well standardized platform for smart cards from different manufacturers. This is supposed to ease compatibility problems between cards and, thorough the use Java, also between other Java platforms and smart cards.

A great deal more software has been written for workstations than for smart cards, therefore many PC developers find themselves in a situation were they would want to reuse existing code base in smart card programming. Java programming language is available for a wide range of platforms including desktops and smart cards. Several differences between these environments complicate the matters despite the use of Java, which promises to ease portability between different platforms through the use of a virtual machine architecture and common syntax.

Programming smart cards is inherently harder than programming in a desktop environment for several reasons: 1) They lack natural input and output. 2) Their processor and memory capacities are limited. 3) The standards are few and not very well followed by the industry. 4) The development environments and languages for the cards have been archaic.

Lack of proper input and output means that other systems such as PC's are needed as essential parts of the development. Having additional complicated systems in the development process tends to make things harder as there are more things that can go wrong. Additionally, more skills and tools are required to get the actual job done.

Today the typical smart card has an 8-bit processor running 4 MHz clock-speed and less than 16 kB of memory. Therefore the typical workstation PC is at least 2 orders of magnitude faster and can natively present and process integers that are 10 million times larger. Perhaps most importantly, also the memory capacities of a PC are three to four orders of magnitude larger. Considering these differences in performance and memory, it should be obvious that programming smart cards is somewhat harder than programming PC's. The programmer has many more limitations that have to be taken into account because of the limited environment.

An fundamental standard for smart cards is ISO7816, which standardizes the smart card and the environment to some extent. It deals mainly with standardizing the communication interface of the card. Unfortunately, it is divided into several parts of which the most sophisticated one is optional (ISO7816-4). The mandatory parts do not set tight requirements for internals of the card, and the optional parts are not very well followed by the manufacturers. Many of the cards claiming to be compliant with ISO7816 actually implement only some mandatory parts of it. Furthermore, while the interface of the commands and the core commands themselves are standardized, many identifying bytes (called CLI-bytes) are vendor specific. These bytes are still essential for calling and executing the actual commands.

Java Card promises to solve many of the problems associated with vendor specific smart card programming environments by defining an API that must be followed by all the complying cards. Like other Sun's Java environments it also specifies an abstract layer on top of hardware, a virtual machine, and common language syntax.

This paper presents the lessons that were learned in a project, whose goal was to implement an Elliptic Curve Digital Signature Algorithm (ECDSA) to a smart card running Java, and evaluate the results. The rest of this paper is organized as follows: In section 2 we present the relevant differences between Standard Java and Java Card environment. In section 3 we describe the project background and assumptions. Section 4 describes the implementation architecture, the memory allocation problem and the solution adopted for it. Section 5 presents numerical performance data of the critical part of the implementation and the analysis of that data. Section 6 gives directions for future work and research, while section 7 concludes the actual lessons of our project.

2 Smart card programming with Java Card

While Java Card does not solve all the problems associated with writing smart card software, it does solve some of them and in that process introduces problems of its own.

One of the main benefits of the Java is its syntax, which is the same in all variations of Java. The common syntax of Java makes it easier for a programmer to write code, as no energy is wasted in the futile effort to learn yet another syntax, which itself is mostly irrelevant in programming. The familiar syntax raises up hopes of easy portability. Unfortunately, here the issue is significantly more complex than Sun would like us to believe. It is

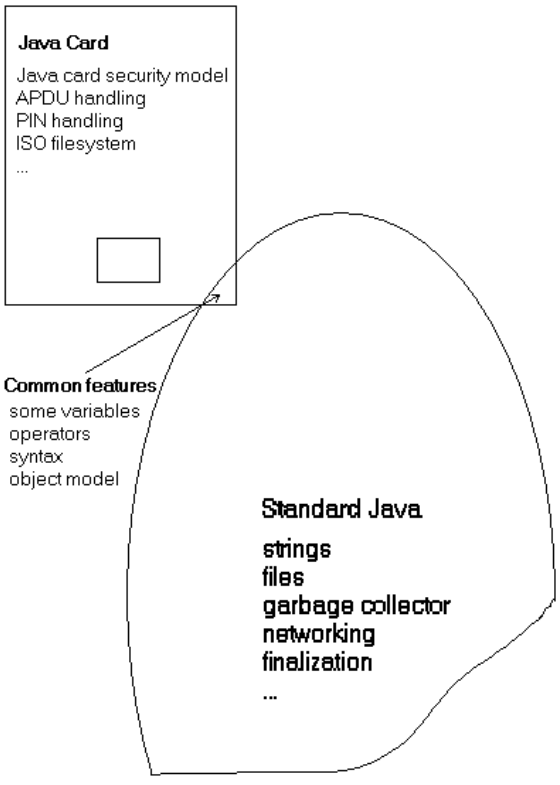

Figure 1. Comparing Java Card and Standard Java as whole development environments.

easy to make a misconception in thinking that Standard Java and Java Card are very similar. This can happen because of the same syntax, similar names and the large amount of marketing hype. In fact, these two environments are relatively far from each other in the programming point of view. The next section presents the main differences and elaborates further on their impact on the programming process.

2.1 Differences of Java Card and Standard Java

Java, as it exists on the card, has many limitations that make the development rather different from that practiced in Standard Java environment. Main differences and limitations are the following [1], [2], [3].

1. No threads
2. No dynamic class loading
3. No garbage collection
4. Missing basic data types (i.e. `long`, `int`)
5. No security manager
6. No cloning
7. No finalization

The lack of threads as well as offering no garbage collection are natural consequences of a more restricted platform, as it would be difficult to fit either of these to the hardware of even the most modern cards. Big differences lie underneath the hood of the computing platform. These differences affect the programming process sometimes rather drastically, as the missing garbage collection and some data types hint.

While in theory it is possible to write a piece of code for the Java Card virtual machine and run the same bytecode on a more sophisticated one, say Standard Java VM, several problems are evident.

1. Current Java Card environments require an additional bytecode compilation phase

   After the compilation of the Java source code to a running bytecode with a Standard Java compiler, an additional compilation phase is needed to make the applet executable on a given card [4]. This compilation (and verification phase) is vendor specific i.e. different vendor's converters produce different bytecode representations from the same source[26].

2. The compatible code is on many cases of little value

   Much of the API of Java Card deals with card specific functions that don't directly have a meaning on the PC side. For example, `PINException` would not mean anything on the PC. Neither would any of the perfectly good security specific Standard Java API calls, when run on the card environment.

Figure 1 illustrates the huge differences Standard Java and Java Card environments have. In fact, as seen from the picture, it is much easier to state the similarities between the two environments: common features are much more rare than unique features of each environment. This means that if the program is to be portable, it must be carefully programmed according to the least common nominator (in this case the most limited JVM i.e. the JCVM.)

2.2 Physical memory architecture of the Java Card

Smart Cards have a type of memory that is usually not found in workstations at all, EEPROM (Electrically Erasable Programmable Read-Only Memory). This memory type is non-volatile, that is its contents are not erased by power loss as is the case with RAM. Volatile types of memory are sometimes also called transient memory and objects allocated in this type of memory, for example RAM, are correspondingly called transient objects.

One can easily understand the need for non-volatile memory on the card. As smart cards are externally powered and clocked power loss can happen very easily. A smart card should still be able to continue consistent operation in such a case after the power is restored.

Most of the memory of current Java Cards is EEPROM and manufacturer's documentation often doesn't even mention the amount of RAM that a specific Java Card has integrated on the chip. Current Java Card specifications state that object are by default allocated in EEPROM. For performance specific implementations such as cryptography this quite an unfortunate choice as write operations to

EEPROM are about 30 times slower than equivalent operations in RAM[27]. Furthermore, there is also a maximum number of writes that EEPROM memory can sustain before becoming non-functional. In the current Java Card memory model compiler and the card environment internally handle the placement of objects between RAM and EEPROM. As Java Cards have different amounts of memory this also means that objects that end up in RAM on some cards will end up in EEPROM on the others. Also a great deal of performance of the actual run time program depends on how well the cards additional compiler and environment are designed.

2.3 "Write once run everywhere" does not work

From the many differences stated above it is clear that "write once run everywhere"-principle that Sun has touted with Java does not work well in practice. This inevitably brings out an important question: What would it mean for it to work as advertised? At least the PC and smart cards environments have little common functionality. Even the two virtual machines have differences let alone language libraries, which are inherently different because of differences in hardware's capabilities such as input and output. Of course, other VMs may have more in common than these two extremes but skepticism seems justified, when judging their usefulness from point of view of code reuse and portability.

Some of the problems of porting software from platform to another, like ubiquitous syntax's, are eased by the introduction of Java. Java, however, produces problems of its own, which can be hard to tackle on many circumstances. In a sense, the easiest parts of the porting are made automatic, as trivial code is most likely to run unmodified on both virtual machines.

Two new problems are introduced to the porting process by the use of Java. First, inclusion of JVM on the card degrades performance, which is already a bottleneck in the smart card environment. Second even worse problem is a lack of any kind of memory control in the card environment where memory is a scarce resource.

Lack of GC makes porting harder
As the philosophy of Java does not allow explicit memory management and the limitations of the hardware of the cards, in turn, does not allow a decent garbage collector, Java Card designers decided to drop memory management, which is indeed a very drastic solution. This means that memory which is once reserved, stays reserved. Unfortunately, it does make porting of applications that use memory carelessly quite hard. As one of the most important features in Standard Java is the garbage collection, one can be almost sure that all applications use memory "carelessly" in the Java Card sense.

In the OS field, operating systems with primitive memory management are considered weak. On the other hand, nowhere is the memory as scarce as on the card. One can justifiably question the design choice of Java Card in this matter. The only good explanation seems to be that including a garbage collector on the card may succeed in the future with reasonable certainty. Hardware support for garbage collectors is an active area of research [5].

3 Project background

In this section we present the temporal order of the porting phase. We begin with a project's background, which is followed by an orderly description of its progression.

3.1 Background

The project of implementing digital signatures on a smart card is part of TeSSA project, which is currently running its third year. TeSSA has been primarily focussing on authorization certificate technologies, such as SPKI, and has concerned itself mainly with privacy aspects of the certification technologies, as well as including the certificates into practical architectures [6], [7].

Since the first year of the TeSSA project, elliptic curve cryptography was one of the points of interest. There were several reasons for this, one of the main ones being the possibility to create smaller certificates which would enable a better support for DNS retrieval of the certificates [8]. Of course, this very same property is quite important when con-

sidering a useful smart card involvement in the certificate infrastructure.

In 1998, a Standard Java implementation of ECDSA was produced [9]. The second phase of the ECC subproject began in May 1999 as a Master's Thesis project of the author. By this time, it had become clear that supporting smart cards in the architecture would offer some key benefits, and on the other hand, it seemed possible that the project had necessary know-how and expertise to build such an implementation.

First, the debugged and finished ECDSA research prototype was ready and working on a workstation Java environment; Sun introduced the Java Card platform and actual cards started to become available. This, in combination with recommendations that elliptic curve cryptography was well suited for smart cards [10], [11] was seen as promising, and it favored the idea of introducing smart card support.

Second, combining authorization certificates with exactly ECC based digital signature algorithm would have an important architectural benefit. Namely, in the elliptic curve based public key algorithms the creation of the key is cheap compared to, for example, RSA based cryptosystems. The creation of the key takes roughly the same time as one encryption operation [12], [13]. In SPKI based certification systems, the ability to locally create keys on the fly is very appealing as spare keys can be used to achieve anonymity. By creating and holding several temporary private keys, users' identities can be protected while still maintaining the ability to give credentials to them [14]. In decentralized authorization systems, users and agents can easily authorize each other without strictly having to use special CAs.

3.2 The prerequisites

The project started with the following prerequisites.

- Implement ECDSA on a commercially available smart card
- The card to be used should be Java Card
- Research the role smart cards have in authorization certificate infrastructures

These prerequisites were not requirements that we actively set but more like implicit assumptions that characterized the project from the beginning. In this paper I concentrate more on the implementation and porting to the Java Card environment and especially the memory allocation on the card.

4 Porting Standard Java ECDSA implementation to a smart card

In this section we present both the problems we faced while porting the existing architecture and the solutions we developed. The architecture of our prototype is described. We also developed a systematic method for memory re-use and describe how it was applied in this project.

4.1 The architecture of the card implementation

In order to implement ECDSA, we need both modular big integer arithmetic and finite field arithmetic as can be seen from Figure 2. While the pure elliptic curve operations can be built on top of many kinds of finite fields[15], there are two basic choices[16]: $\mathbf{F}_{2^m}$ or characteristic two fields, and $\mathbf{F}_p$, 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

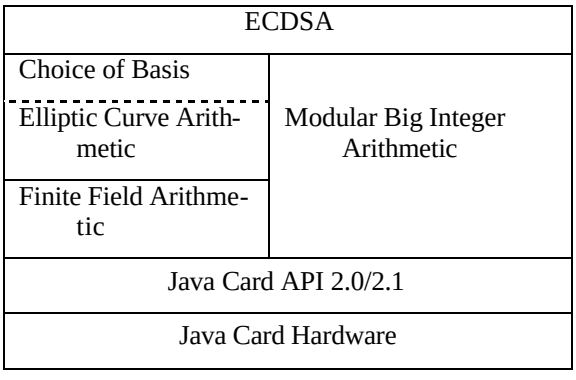

Figure 2. Layer structure of the smart card ECDSA architecture.

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 the fastest operations on any processor. It should be noted that these promises do not necessarily realize when using cards with Java because there is no direct access to hardware.

There are many variations of both of the aforementioned fields; many properties can be carefully chosen to affect either performance or ease of implementation. (See e.g. [19], [21]).

Of these we chose the odd prime field largely for practical reasons. As mentioned before, we had a possibility to use an existing debugged implementation. This implementation was based on the finite field arithmetic being an odd prime field. That existing code base and architecture would not have been of much use, had we implemented ECDSA with a different underlying field. Using the same finite field gave as an opportunity to study the portability issues between PC and smart card environments.

4.2 The main design problem: re-using memory

Our main concern raised naturally from the technical situation of the current Java cards. Our main problem was how to fit the functioning code to only little over 12 kB of memory, and still leave enough space for keys and at least one certificate.

Here we concentrated on finding methods, which could be used in porting the existing ECDSA implementation to Java Card platform. As the existing implementation was a Standard Java application, there was primarily a need for systematic method or formalism that could be applied to translating the memory allocation and usage of the original program from the typically careless reserve-and-forget (workstations) paradigm to a more economical reserve-and-reuse (smart cards) one.

In the layered architecture the lowest layers are usually most performance sensitive. For this reason we started the porting process from the bottom up: Directly above Java Card API is the finite field layer. In the workstation ECDSA it is based on the Standard Java `BigInteger`, which is problematic because it is based on immutable semantics. This fits nicely for GC capable Standard Java but is totally unsuitable for Java Card. So at the very least we had to face the problem of making a mutable version of the `BigInteger`.

A more detailed look at the JDK 1.2 version of `BigInteger` revealed that it called underlying native C library for its operations. We now had two distinct problems stemming from the main design problem of getting the memory to suffice. First, what general method should be used to allocate memory? Second, how do we get the long number arithmetic functionality to the card, which uses mutable semantics? The solutions to these problems are discussed in section 4.3.

Mutable vs. immutable semantics

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 have only limited memory and no garbage collector, we have to explicitly handle the reuse of memory.

Basically, in immutable semantics the value of the object itself can never be modified. Each call to an objects' method allocates and returns a new temporary object, whose reference is assigned to user defined variable. In mutable semantics, exactly the opposite is true. Calls to objects method always modify the object itself; new memory is not necessarily allocated at all and, likewise, no references need to be returned.

It is crucial to understand that the interface syntax is not able to distinguish between mutability and immutability. Memory allocation is a side effect and cannot be revealed by the interface alone. Software engineering community has been focused on the design of interfaces for a long time. The porting problem we encountered is good example of a portability problem, which cannot be solved by solely focussing on the interface design.

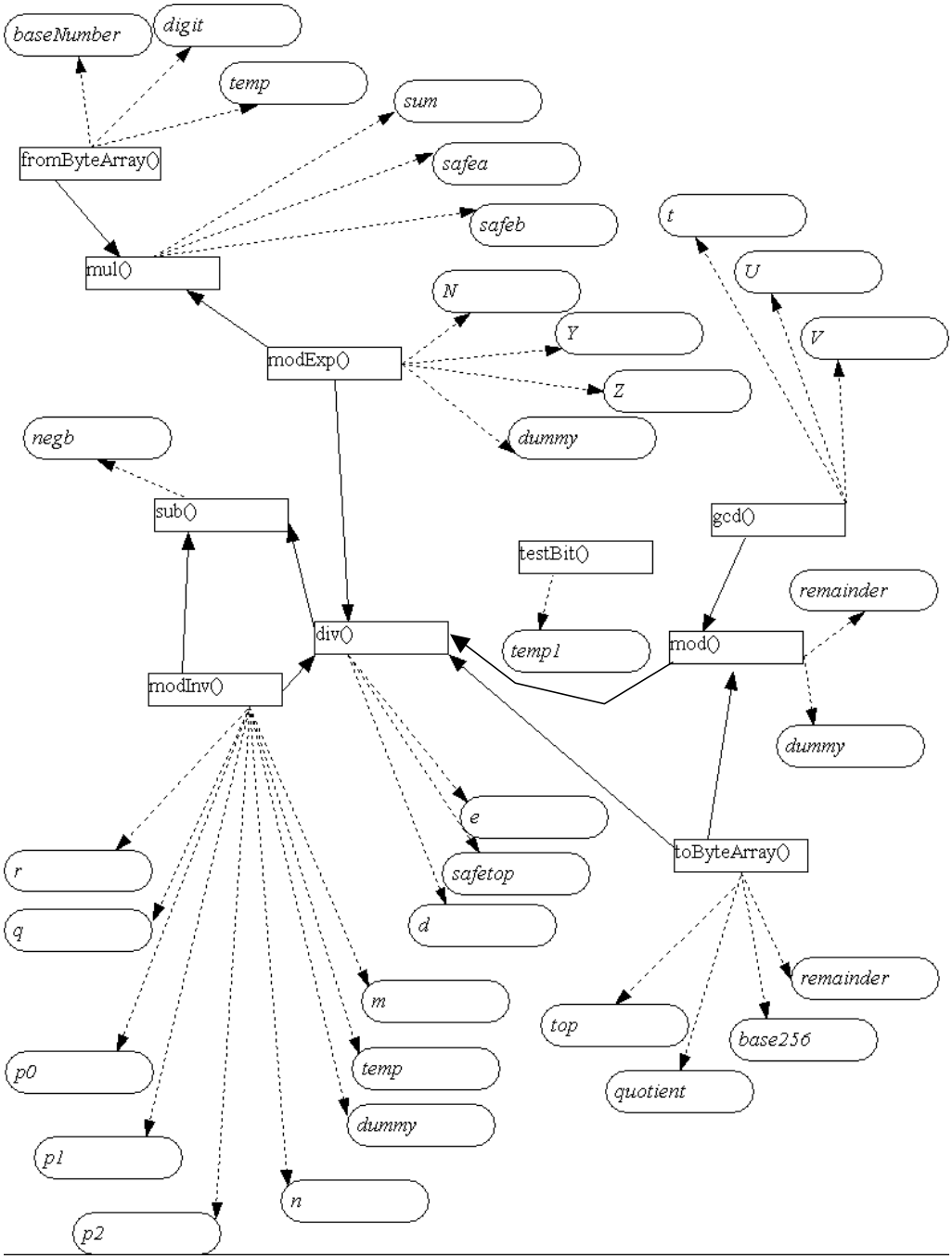

Figure 3. A directed data flow analysis graph of the `MutableLargeInteger` *class. Dashed lines point to temporary MutableLargeInteger objects before optimization. Solid lines point to other methods, which are needed by the method at the beginning of the arrow.*

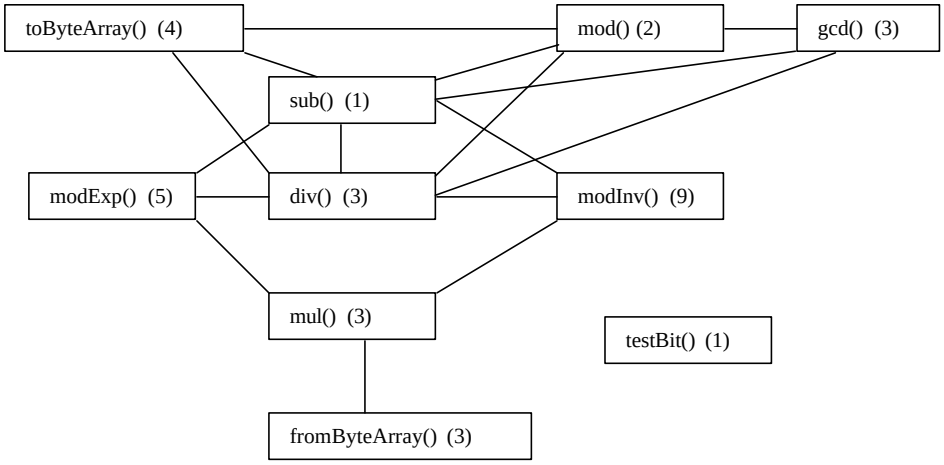

Figure 4. The interference graph of the methods of the `MutableLargeInteger` *class. After the name of the method, the number of* `MutableLargeInteger` *instances is shown in parenthesis. As a method is the basic block in this case, all the temporary variables inside one method are presumed to interfere with each other.*

4.3 Solutions for the memory allocation

As to the first problem of needing a systematic method for porting Standard Java application to Java Card, we used a modification of a so called register allocation approach. In our case, we used variable interference graphs with graph coloring to minimize number of temporary variables. We built variable interference graphs manually by hand and colored a graph of local scopes with a minimal set of temporary variables. That is, the code was first programmed normally and all the variables were named and allocated in local scope. These different local scopes are easily interpreted as basic blocks, which are the nodes of the variable interference graphs. The register allocation problem and its solution methods are well known but traditionally used in compiler design[17]. To our knowledge it has never been applied to minimizing the number of temporary variable allocations in a Java Card implementation.

In our case we took each method of `MutableLargeInteger` as our basic block and analyzed which methods call each other. Based on this high level data flow analysis graph described in Figure 3, we constructed a variable interference graph, which we "colored" using a minimal set of temporary variables. Graph coloring is an NP-complete problem in a general case, but small cases are solvable by hand, and there are good heuristics for solving larger cases with computers. A variable interference graph for one of our prototype's classes is presented in Figure 4.

Using the above method lets us use 15 temporary variables in the case of example in Figure 4, instead of the 34 variables, which would be needed without any optimizations. We get the result of 34 variables by straightforwardly replacing each local scope variable with a global one. (i.e. By just adding the numbers in parenthesis displayed in each node.) This optimization is quite high-level as methods are not the smallest imaginable basic blocks and even larger savings are likely if a more detailed data flow analysis was executed. However, this would require handling graphs that are much larger than the aforementioned example and in practice tool support would be needed to make the optimization.

As for the solution of the second problem, we decided to code a purely Java based long integer package on our own (which is the actually the example class `MutableLargeInteger` presented above). Its design was based on C code by Michael Rosing [18].

4.4 Performance challenge of the implementation

From the very beginning, we also had another major concern, which was speed. As the implementation was done in Java, which is not actually famous of its speed, we decided that an order of magnitude lower performance, acceptable for commercial products, would be sufficient for us. Commercial smart card products employing cryptography try to achieve subsecond performance of operation. This is because a waiting period of one second feels pretty much instantaneous.

As our research objective was mainly to gather experience of programming the cards and study the feasibility of a cryptographic software implementation in such a setting, we decided to first concentrate on producing at least one prototype and on optimizing its performance afterwards. We however planned our approach so that it could be easily modified and optimized in later stages. Most importantly, we created a layered architecture and assumed that concentrating in optimizing the lowest layers would be beneficial.

5 Performance of the prototype

We only measured the performance of the most critical part of the implementation, that is the lowest software layer of our `MutableLargeInteger`. While these figures do not directly measure the performance the end user of the system will see, they are nevertheless important in estimating how fast the whole system can or cannot be made.

	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 2. Performance of `MutableLargeInteger` *on the card as a function of the integer bit size.*

We measured the performance of the lowest software layer, because its functionality is heavily called by the ECDSA layer and therefore it is essential from the performance point of view.

As can be seen from the figures in Table 2, the performance of the prototype leaves much room for improvement considering practical purposes. The execution times grow nearly linearly with field addition and faster in the case of multiplication.

5.1 Performance comparison on workstation environment

Next, we analyze, what are the bottlenecks of the implementation and what can be made to improve it. Lastly, we estimate of how much it can be improved. The conclusions made from the current performance are summarized in the conclusions part.

	JDK 1.2 `BigInteger`	`MutableLargeInteger`
1000 multiplications	<0.1 sec.	0.3 sec.
1000 additions	<0.1 sec.	<0.1 sec.
1000 inversions	1 sec.	44 sec.

Table 1. Performance comparison of the different big integer implementations with the number length of 192 bits.

We ran a series of tests in which we compared the performance of the created `MutableLargeInteger` class with the JDK 1.2 `BigInteger` class. Each of the test was run 1000 times in a tight loop in workstation environment. As can be seen from Table 1, there is no significant difference in performance of addition operation between different implementations. `MutableLargeInteger` multiplication routine is quite much slower than the `BigInteger` one. The limitations of the test setup and timer mechanism do not reveal the exact performance ratio. It is likely that the low performance of the multiplication routine also explains some of the low performance of the inversion routine. The reason for this is that the

inversion routine is already near the optimal and major improvements are unlikely. Inversion routine also quite heavily uses division which spends a significant amount of time calling multiplication. Our division and multiplication routines are not optimal and could be improved [18], [19].

6 Future Work

Many things still remain undone if one considers implementing cryptographic operations for a smart card in software. Here we would like to take the opportunity to suggest directions for future research.

To produce a more widely usable implementation than our prototype, speed of the routines must be optimized. There are several paths to a better performing application. First, there is the most straightforward and least demanding way, which is to select and implement better routines for $\mathbf{F}_p$ arithmetic, such as the ones mentioned in [19].

A well performing software cryptography for smart cards requires more radical changes to the approach. Second possibility is to use optimal extension fields to make calculations faster [20]. Further, the use of hyperelliptic curves and public key cryptography based on them has been suggested by some mathematicians [22]. Hyperelliptic curves are, however, mathematically even more challenging than elliptic curves. Some of the details, which are essential from the implementation point of view, like the representation of the point entities, have not yet been standardized [23].

Yet another approach is to use entirely different kinds of public key cryptosystems. One such system is NTRU which was recently patented by Hoffstein et. al. This system seems ideal for smart card implementations as its code takes up only a very small space of 1500 bytes, and it is suggested to perform much faster than EC based systems [24], [25].

As the Java environment on the card poses a major limit to what optimizations can be done in practice we plan to implement our next ECDSA prototype, together with algorithmic optimizations, on a palm sized computer.

7 Conclusions

While using Java is usually considered to enhance the quality of the code, especially the limitations of memory on the cards are made worse by the use of Java as a programming language. In current versions of Java Card the once reserved memory stays reserved until the given applet is deactivated. On many Java Card platforms, including the Schlumberger Access platform we used, there is no explicit way to allocate objects in RAM. This makes the porting of working Standard Java applications a considerable challenge. Memory must be reused, and every allocation of variables and objects must be carefully considered. The inability to allocate objects in transient memory results in a situation where most objects end up in EEPROM which slows down the execution speed of the applet.

In practice, a systematic method for reusing the variables is required. In our approach we successfully used our own modification of register allocation method and register interfere graphs with graph coloring. Variable interface graphs seem to easily grow so large that they cannot in practice be used without automated tool support. This is also a problem in the case of register allocation: the very nature of the problem, the complexity of it makes it intractable in a general case. Efficient heuristic methods are available and can be successfully used in practice as the research in compiler design has demonstrated. Theory for creating the required tools for solving Java Card memory allocation problems exists. Even following the current standard is possible by incorporating such methods in external tools.

All in all, if we were to begin the same project with our current understanding and knowledge today, we most likely would not select smart cards as our secure tokens. Instead, a more capable device such as a PDA would be chosen. PDAs have an order of magnitude more memory so the memory allocation doesn't pose as inhibiting limits as with smart cards. This is also the direction where we plan to continue our implementation efforts.

While our experiences with smart cards is considered valuable, if faced with a similar design problem today we would more seriously consider using other than Java based cards. Of course, we would still need to keep in mind that we had a fully functional cryptographic module in Java. From a point

of view of commercial implementation, which has to conform to strict performance requirements, the choice would clearly be something else than a Java based platform. As far as research is concerned, implementing performance sensitive algorithms with high level languages is very important as this kind of activity will probably increase in the future.

Software based cryptography for smart cards requires considerable engineering effort even when using elliptic curve methods. For completely Java based cards elliptic curve methods are insufficient in providing speed that could be used in practical applications. New types of public key cryptosystems promise better performance, but empirical research results are still missing.

References

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

[2] *Java Card Virtual Machine specification*, Sun Microsystems.

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

[4] Cyberflex Access Programmer's Guide, Schlumberger, 1998.

[5] J. Morris Chang, Witawas Srisa-an and Chia-Tien Dan Lo, *An Introduction to DMMX (Dynamic Memory Management Extension)*, ICCD Workshop on Hardware Support for Objects and Microarchitectures for Java, Austin, TX. October 10, 1999.

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

[7] Jonna Partanen, Pekka Nikander, *Adding SPKI Certificates to JDK 1.2*, Proceedings of the Nordsec'98, the Third Nordic Workshop on Secure IT Systems, November 1998.

[8] Tero Hasu, Yki Kortesniemi, *Implementing an SPKI Certificate Repository within the DNS*, Poster Paper Collection of the Theory and Practice in Public Key Cryptography (PKC 2000), January 2000.

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

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

[11] Don B. Johnson, Alfred J. Menezes, *Elliptic Curve DSA (ECDSA): An Enhanced DSA.*

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

[13] 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.

[14] Tommi Elo, Pekka Nikander, *Decentralized Authorization on Java Smart – A Software Implementation*, to appear in Cardis 2000, September 2000.

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

[16] Thomas W. Hungerford, *Algebra, Graduate Texts in Mathematics*, Springer-Verlag, New York, Inc, 1974

[17] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley series in Computer Science, 1986.

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

[19] Ian F. Blake, G. Seroussi, Nigel P. Smart, *Elliptic Curves in Cryptography*, London Mathematical Society Lecture Note Series 165, 1999.

[20] D. Bailey, C. Paar, *Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms*, CRYPTO '98, August 1998.

[21] Toshio Hasegawa, Junko Nakajima and Mitsuru Matsui, *A Practical Implementation of Elliptic Curve Cryptosystems over GF(p) on a16- bit Microcomputer*, First International Workshop on Practice and Theory in Public Key Cryptography, PKC'98, February 1998.

[22] A. Menezes, Y. Wu and R. Zuccherato, *An Elementary Introduction to Hyperelliptic Curves*, In: N. Koblitz: Algebraic Aspects of Cryptography. Springer-Verlag, Berlin Heidelberg New York (1998).

[23] Maarit Hietalahti, *A Literature Survey of The Hyperelliptic Curves and Their Use in Cryptosystems*, Technical Report, Helsinki University of Technology, January 2000.

[24] Jeff Hoffstein, Daniel Lieman, Jill Pipher, Joseph H. Silverman, *NTRU: A Public Key Cryptosystem*, IEEE P1363: Protocols from other families of public-key algorithms, Technical Report, October 1999.

[25] *The NTRU Public Key Cryptosystem – Operating Characteristics and Comparison With RSA, ElGamal, and ECC Cryptoystems*, NTRU Communications & Content Security Learning Center, available from http://www.ntru.com/technology/tutorials/operatingchar.htm

[26] Jordi Castellà-Roca, Josep Domingo-Ferrer, Jordi Herrera-Joancomartí and Jordi Planes, *A Performance comparison of Java Cards for micropayment implementation*, Smart card research and advanced applications, Fourth Working Conference on Smart Card Research and Advanced Applications, Kluwer Academic Publishers, IFIP Cardis 2000, September 2000.

[27] Marcus Oestericher, Kseerabdhi Krishna, *Object Lifetimes in Java Card*, Proceedings of Usenix Workshop on Smartcard Technology, May 1999.