A Survey of Public-Key Cryptography on J2ME-Enabled Mobile Devices

Stefan Tillich and Johann Großschädl

Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria
{Stefan.Tillich,Johann.Groszschaedl}@iaik.at

Abstract. The advent of hand-held devices which incorporate a Java Virtual Machine (JVM) has greatly facilitated the development of mobile and wireless applications. Many of the possible applications, e.g. for e-commerce or e-government, have an inherent need for security which can be satisfied by methods of public-key cryptography. This paper investigates the feasibility of public-key implementations on modern mid-range to high-end devices, with the focus set on Elliptic Curve Cryptography (ECC). We have implemented the Elliptic Curve Digital Signature Algorithm (ECDSA) for both signature generation and verification and we show that both can be done on a J2ME-enabled cell phone—depending on the device—in times of a few seconds or even under a second. We also compare the performance of ECDSA with RSA signatures and provide some key issues for selecting one protocol type for implementation in a constrained device.

Keywords: Public-key cryptography, elliptic curve cryptography, digital signature, ECDSA, J2ME-enabled device.

1 Introduction

Today the market for mobile communication and computing devices like cell phones and PDAs is growing rapidly. The issue of application development for such a great number of different devices has been addressed by the integration of Java Virtual Machines (JVMs). Most of today's devices conform to Sun Microsystems' Java 2 Platform, Micro Edition (J2ME). J2ME devices allow a fast deployment of mobile and wireless applications. Many possible applications have an inherent need for security, e.g. applications for mobile electronic payment, authentication to access secure networks, digitally signed mobile transactions, secure messaging and digital rights management. Cryptographic secret-key and public-key algorithms can be employed to satisfy this need for security.

It is noteworthy that public-key methods require the presence of a public-key infrastructure (PKI). A PKI allows public keys to be validated and connected to the respective owner. This is accomplished through the generation, provision and revocation of user certificates which is usually done by trusted parties called

C. Aykanat et al. (Eds.): ISCIS 2004, LNCS 3280, pp. 935–944, 2004.

Certificate Authorities (CAs). Once established, PKI services can be provided to all network-enabled devices including hand-held devices.

It is a good practice to implement security systems as open systems. An open system complies with specified, publicly maintained, readily available standards and can therefore be connected to other systems that comply with the same standards. Such systems bear many advantages, e.g. interoperability, flexibility, and public acceptance. For example, an operator of an e-commerce portal may want to enable customers to buy via cell phones. Requirements for transactions are most likely secrecy, protection from data manipulation and non-repudiation. Additionally, the customers could require the e-commerce portal to authenticate itself. All these services can be provided by a set of secret-key and public-key cryptographic methods. If public standards like [1,12,13,14] are used, then the portal can make use of already established PKIs, interact with already existing client application and may even be able to act as a intermediary for other portals.

The usefulness of public-key methods in mobile and wireless applications is evident, but they come at a price. All public-key algorithms require complex mathematical computations. Constrained devices may not be able to offer enough resources to allow an implementation of the required public-key protocols. The work described in this paper has been conducted to examine the current situation in this regard. With our implementations we show the feasibility of digital signature operations in modern-day cell phones. We provide performance timings which have been measured on four different cell phone types for implementations of ECDSA and RSA signatures. The results are subsequently used to derive some general recommendations for mobile and wireless applications.

The rest of the paper is organized as follows. Section 2 gives a short overview of public-key cryptography and highlights important properties of Java-enabled hand-held devices. Section 3 deals with optimizations for Java applications which incorporate public-key algorithms. In Section 4 we provide the results and conclusions from our performance measurements on four different cell phone models. Section 5 gives a short summary of the presented work and also some prospects on the integration of public-key algorithms into constrained computing devices.

2 Public-key Cryptography in Constrained Devices

2.1 Public-key Cryptography

Public-key cryptography is a relatively new topic in the long history of cryptography. It was first proposed in the 1970s by Diffie and Hellman with their key agreement protocol [2] and by Rivest, Shamir, and Adleman with the RSA algorithm [15].

Public-key methods are generally based upon so-called trapdoor one-way functions. These are mathematical functions which are easy to compute in one direction, but are hard to invert. However, the inversion can be facilitated if one is in the possession of some piece of additional information: the so-called trapdoor information. The main current public-key algorithms rely on the hardness of one

of two mathematical problems: integer factorization (IF) or discrete logarithm problem (DLP). The equivalent of the DLP for elliptic curves—denoted as elliptic curve discrete logarithm problem (ECDLP)—is particularly interesting, as no subexponential-time algorithm for solving it has been discovered so far. This fact distinguishes Elliptic Curve (EC) algorithms from other cryptosystems which are based on IF or DLP like RSA [15] and Diffie-Hellman [2]. In contrast to the ECDLP, there are known subexponential-time algorithms for solving both the IF problem and the DLP in conventional number fields.

A general relation exists between the hardness of the underlying problem and the minimal length of the operands, i.e. keys and other parameters, of the public-key algorithm. For a given cryptographic algorithm and a desired level of security the operands must have a certain length. The operand length has a direct impact on the performance and memory requirements of an implementation of the algorithm. The assumed hardness of ECDLP results in shorter operands for EC methods in comparison to other algorithms. In [9], Lenstra et al. provide a thorough comparison of the security of public-key cryptosystems based on IF, DLP, and ECDLP. For instance, a key size of 190 bit for an EC algorithm is approximately equivalent to an RSA key size of 1937 bit under the condition that there will be some progress made towards more efficient solutions of the ECDLP in the future. If no such progress is made, then the required RSA key for equivalent security even grows to over 3137 bit.

2.2 Properties of Java-enabled Devices

If public-key algorithms are to be implemented in constrained devices then EC methods appear to be an attractive option. But all public-key operations require substantial computing resources. Furthermore, despite the many advantages which are offered by Java it has a poor performance in comparison to native code. Therefore, achieving an adequate performance is the biggest challenge of implementing public-key algorithms in Java on constrained devices. In the following we will examine the current situation on Java-enabled devices regarding support for cryptographic algorithms.

The most widely supported mechanism for Java deployment on hand-held devices conforms to Sun Microsystems' Java 2 Platform, Micro Edition (J2ME). The base set of application programmer interfaces (APIs) is defined in the Connected Limited Device Profile (CLDC) which is currently available in version 1.0 and 1.1 [17,19]. The CLDC together with the Mobile Information Device Profile (MIDP) form the Java runtime environment for most of today's Java-enabled hand-held devices. The MIDP is available in version 1.0 [18] and version 2.0 [6]. Applications which conform to MIDP are commonly called MIDlets. MIDP 1.0 provides a thinned down variant of the standard Java API and is implemented today in many mobile devices. MIDP version 2.0 adds limited support for games, media control and public-key certificates. Moreover, secure connections over HTTPS and secure socket streams (based on either TLS version 1.0, SSL version 3.0 or WTLS) are provided. However, there is no access to the cryp-

tographic algorithms which implement the secure connection. Therefore, tasks like data signing cannot be done with the MIDP 2.0 API.

The best solution for the provision of general public-key methods for the application programmer would be the inclusion of the required cryptographic algorithms into MIDP. In this scenario, the computational extensive methods could be implemented efficiently by the Java Runtime Environment (JRE) and could use all the features of the respective device. Unfortunately, so far neither version of MIDP offers support for cryptographic methods. Not even the highly useful BigInteger class from the standard Java API, which facilitates low-level arithmetic for many cryptographic algorithms, is included. However, the Java Specification Request (JSR) 177 [7], which is currently being prepared for first release, proposes the integration of a Security Element (SE) into J2ME devices. Such an SE provides support for secure storage of sensitive data, cryptographic operations and a secure execution environment for security features. The integration of these APIs into J2ME devices will be a big step towards mobile and wireless application security. But J2ME devices without such a support will nevertheless stay in broad use in the next years.

Device-specific APIs for cryptographic operations could be provided by the manufacturer. But the use of such APIs would necessitate different Java MIDlet versions for different devices. So this option is only applicable if there is a known and relatively small number of target devices.

Due to complications with code signing, MIDP 2.0 does not include the possibility for the installation of user libraries. Therefore it is not possible to install a cryptographic library shared by different MIDlets to reduce code size.

The current situation leaves only one practical option for applications which require access to cryptographic algorithms. This option is to bundle the required cryptographic classes with the actual application classes. A drawback of bundling is that it leads to relatively large code sizes. This can cause problems with devices which enforce a limit of the application size and therefore inhibit application deployment.

3 Implementation Issues

As outlined in Section 2, two important factors must be considered for MIDlets which include public-key algorithms: performance and code size. Based on our practical work, we provide some hints for optimization of both factors.

3.1 Performance

The effectiveness of different performance optimizations is dependent on the underlying machine and JRE. However, there is a number of useful general rules which should be regarded. Many articles are available which give hints for achieving better performance in Java. From our experience, some of these hints are more applicable to public-key implementations than others and we try to list the more effective ones in the following.

The most convenient way for optimization is to use the optimization switch of the java compiler (`-O` for Sun's javac). Such an compiler optimization can both increase performance and decrease code size.

An important step of optimization consists of application profiling. Profiling identifies methods, which are frequently called and are worth optimizing. Various tools are available for profiling of Java applications. For example, Sun's JRE has a non-documented switch `-prof` which turns on profiling. All general rules for optimization like object recycling, avoidance of String concatenation, inlining of short methods and replacement of short library methods (e.g. `Math.max`) by local methods should be considered.

Implementations of public-key algorithms often deal with multi-word values and look-up tables. Java arrays seem to be the natural choice, but it should be noted that array accesses introduce a certain overhead for index range checking. If array elements are statically indexed, i.e. the index is a constant value, then it can be favorable to break the array into a number of separate variables. Application wide constants should always be declared as `static final`, which allows for better performance and reduces code size.

Another problem can be posed by lengthy initialization code. Such initializations can be moved to a static initializer block of a class which is loaded at startup. This way the execution of the actual public-key algorithm can be shortened at the expense of a longer MIDlet load time. Another way is to do initialization in a separate background thread. This approach can be very favorable if the MIDlet has to wait for input, as the initialization can be done during this period. In the case of ECDSA signature generation, this strategy can lead to a dramatic performance gain, as outlined in Section 4.2.

3.2 Code Size

The first step to reduce code size is to get rid of unused classes. If all of the MIDlet classes are written by the developer, then he can handcraft them to his specific requirements. In this fashion, the code size can be kept small. When an existing cryptographic library is included, then there are often many unused classes. There are quite a few tools available which can analyze MIDlets and remove unnecessary classes.

Most J2ME devices require the class files and additional resources (e.g. picture files) to be put in a Java Archive (JAR) file. A JAR file conforms to the widely adopted ZIP format and features some additional meta data. Most importantly JAR files provide compression of the bundled files. Compression is optional, but should always be used for MIDlets.

Another possibility for code size reduction is obfuscation. Normally, obfuscation is used to prevent decompilation of Java class files. Obfuscator tools usually replace field, method and class names with shorter ones, remove debug information and compress constants. These measures can result in a substantially smaller bytecode. There are however some issues related to obfuscation which can affect the functionality of a program. Most importantly, class renaming can

prevent the explicit dynamic loading of classes. Obfuscation tools can be configured to perform only certain code modifications and we recommend a careful selection of those modifications.

4 Practical Results

We have implemented the complete ECDSA signature generation as specified in ANSI X9.62 [1] in a Java MIDlet. We have used the parameters and vectors of Section J.2.1 of ANSI X9.62, which are based on an elliptic curve over the binary extension field GF(2^{191}). All our MIDlets bundle the application classes with the library classes which provide the cryptographic functionality. The MIDlets have been built with Sun's Wireless Toolkit version 2.1 and J2SDK 1.4.2.

Four different J2ME-enabled cell phone types have been used to measure execution times for EC point multiplication and ECDSA and RSA signature generation and verification. Timing results for the mid-range devices Siemens S55, Nokia 6610, Nokia 6600, and the high-end Ericsson P900 are provided.

All measurements have been done on the actual devices using their system time features available through the CLDC 1.0 API. Therefore the accuracy of the measured times is not very high. Nevertheless the measurements provide a solid basis for comparison of different algorithms and for estimates of response times of public-key enabled applications. The measurements of different algorithms are given as time for the first execution and—where sensible—as an average of 20 executions. Algorithms with running times of more than 30 seconds are clearly unsuited for that particular device and therefore no averaging was done (RSA signature generation on Nokia 6610 and Siemens S55, ECDSA signature generation on Siemens S55). Note that normally, but not always, the first execution takes longer than the average due to initializations like class loading.

4.1 EC Point Multiplication

The fundamental building block of virtually all EC cryptosystems is a computation of the form $Q = k \cdot P$, which is nothing else than adding a point $k - 1$ times to itself, i.e. $k \cdot P = P + P + \cdots + P$. This operation is called *point multiplication* or *scalar multiplication*, and dominates the execution time of EC cryptosystems. Scalar multiplication on an EC is analogous to exponentiation in a multiplicative group. The inverse operation, i.e. to recover the integer k when the points P and $Q = k \cdot P$ are given, is the elliptic curve discrete logarithm problem (ECDLP). The hardness of the ECDLP is fundamental to the security of ECC as outlined in Section 2.1.

The time which is required for a EC point multiplication determines the overall execution time of an ECC algorithm to a very high degree. Therefore we concentrated our effort on an efficient implementation of the point multiplication. We used Montgomery's method [11] in combination with the fast multiplication (projective version) as described by López et al. [10]. For the arithmetic operations in the underlying finite field GF(2^{191}) we implemented methods described

Table 1. EC point multiplication execution time (in ms)

Device	First execution	Average
Nokia 6610	2.183	2.150
Nokia 6600	984	720
Ericsson P900	578	428
Siemens S55	17.135	17.216

in [4]. For field multiplication we used the left-to-right comb method (Algorithm 4 in [4]) with a window size of 4. Squaring was done with precomputed 8-bit polynomials (Algorithm 7) and inversion with the Extended Euclidean Algorithm for inversion in GF(2^m) (Algorithm 8).

Table 1 lists the measured execution times in milliseconds for a single point multiplication over the finite field GF(2^{191}) on the different tested devices. An EC point multiplication can be realized in under a second to a few seconds on three of the tested devices. However, it can be seen that the performance on the Siemens S55 is not high enough to allow an implementation of EC algorithms with a sensible response time.

4.2 ECDSA and RSA Signature Generation and Verification

The timings for ECDSA and RSA signature generation are given in Table 2, while the signature verification results are listed in Table 3. The key sizes of both implementations have been chosen according to [9] to provide the same level of security. Both algorithms use the hash method SHA-1 for message digesting.

The ECDSA implementation uses elliptic curves over the binary extension field GF(2^{191}) and a key size of 191 bit. For the test runs, the parameters and vectors of Section J.2.1 of ANSI X9.62 [1] have been used. The implementation uses the code of the previously described EC point multiplication. Modular multiplication and inversion in GF(p) are done in the Montgomery domain. We have used Montgomery multiplication with the Separated Operand Scanning (SOS) as described in [8] by Koç et al. and the Modified Kaliski-Montgomery inverse as described in [3] by Savaş et al.

The RSA implementation uses a key size of 1937 bit. The implementation is based on the IAIK JCE micro edition [5], which uses the Chinese Remainder Theorem and Montgomery multiplication. The RSA implementation just serves as comparison for the ECDSA implementation and therefore no special algorithms like Multi-Prime RSA [16] have been examined.

The code sizes of our MIDlets were 53 kB for ECDSA and 60 kB for RSA. These figures refer to the size of the JAR file of the full implementations of signing and verifying including test code. No obfuscation has been used.

ECDSA signature generation has the property that a great deal of precomputation can be done which does not involve the signed data. Most significantly, the EC point multiplication can be precomputed at runtime. If an application has some idle time (e.g. waiting for user input), then the precomputation can be

Table 2. ECDSA and RSA signature generation execution time (in ms)

	ECDSA		RSA	
Device	First execution	Average	First execution	Average
Nokia 6610	2.294	2.266	74.682	N/A
Nokia 6600	860	763	7.125	4.077
Ericsson P900	453	418	3.703	2.725
Siemens S55	18.963	18.117	883.602	N/A

Table 3. ECDSA and RSA signature verification execution time (in ms)

	ECDSA		RSA	
Device	First execution	Average	First execution	Average
Nokia 6610	4.382	4.449	2.825	2.488
Nokia 6600	1.266	1.247	157	139
Ericsson P900	843	854	109	97
Siemens S55	35.277	N/A	30.094	30.661

done in the background in a low priority thread. In this way, the completion of the signing process upon availability of the data to sign becomes negligible. Our tests showed that signing can be done in a few milliseconds.

4.3 Analysis of the Measured Data

It can be seen from the listed timings that the EC point multiplication is indeed the dominating factor for the ECDSA operations. ECDSA signing requires one point multiplication while verifying requires two. Therefore verification takes approximately twice the time of signing. The difference between signing and verification is more dramatic for RSA signatures. RSA public-key pairs are normally selected so that the public key is relatively small while the private key is big. This is the reason that the RSA signature verification is up to 30 times as fast as signature generation (Nokia 6600).

A direct comparison of ECDSA and RSA signatures reveals that ECDSA signing is faster than RSA signing (33 times on the Nokia 6610) and RSA verifying is faster than ECDSA verifying (9 times on the Nokia 6600 and Ericsson P900). ECDSA signing and verifying performs in under 5 seconds on all devices (except Siemens S55). RSA verifying performs in under 3 seconds on all devices (except Siemens S55). RSA signing takes under 5 seconds on the Nokia 6600 and the Ericsson P900 but is very slow on the Nokia 6610, probably due to the lack of an efficient hardware multiplier. Unfortunately, there are rarely any details like microprocessor type, incorporated hardware accelerators (e.g. coprocessors), memory types and sizes, clock frequency etc. publicly available for the particular phone models. This hampers all attempts to interpret the measured results based on these data and to draw conclusions in this regard.

Decisions for a cryptosystem should be based on details of the particular application. Important factors can be the number and ratio of signature generations

and verifications, the presence of waiting times for user input and compatibility with existing systems. For example, the full check of digital signed data requires the retrieval of the whole certificate chain and one verification for each certificate. In this case RSA should be chosen for its faster verification. On the other hand, applications which perform a signature of data which is entered by the user could perform precomputations for ECDSA during waiting times and do the actual signing in a few milliseconds.

The following general recommendations for implementation of public-key cryptosystems in J2ME devices can be derived from our measured performance results:

- If only the verification of digital signatures is to be performed, then RSA signatures should be implemented.
- If only signing of data is required, then ECDSA signatures should be chosen.
- If both signing and verification are required, then the choice should be dependent on the particular application.

5 Summary

In this paper we have outlined the current situation regarding the implementation of public-key algorithms in J2ME-enabled devices. We have presented some practical hints for the optimization of performance and code size. Furthermore we have shown the feasibility of Java implementations of public-key operations on constrained devices regarding both ECDSA and RSA signatures. Our timing results have shown that modern J2ME devices are capable of performing public-key algorithms which offer a high degree of security (191 bit ECDSA, 1937 bit RSA). High-end devices like the Ericsson P900 can even execute ECDSA signature generation and verification in under one second. Furthermore we have compared ECDSA and RSA signature algorithms which can serve as a basis for selecting particular public-key protocols for secure mobile applications.

It is only a matter of time until the growing processing power of mobile devices allows for an easy integration of public-key algorithms into mobile and wireless applications. The adoption of the Security and Trust Services API (JSR 177) [7] by device manufacturers will provide a good basis for application developers to produce secure MIDlets. And such secure applications are vital for building trust of end users and for opening whole new fields of application for mobile computing.

Acknowledgements The research described in this paper was supported by the Austrian Science Fund (FWF) under grant number P16952N04 “Instruction Set Extensions for Public-Key Cryptography”. The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT. The information in this document reflects only the author’s views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

1. American National Standards Institute (ANSI). X9.62-1998, Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA), Jan. 1999.
2. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov. 1976.
3. E. Savas and Ç. K. Koç. The Montgomery Modular Inverse—Revisited. *IEEE Transactions on Computers*, 49(7):763–766, July 2000.
4. D. R. Hankerson, J. C. López Hernandez, and A. J. Menezes. Software implementation of elliptic curve cryptography over binary fields. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 1–24. Springer Verlag, Berlin, Germany, 2000.
5. IAIK Java Security Group. IAIK Java Cryptography Extension Micro Edition (IAIK-JCE ME). Available at `http://jce.iaik.tugraz.at/products/`.
6. JSR 118 Expert Group. Mobile Information Device Profile, Version 2.0. Available for download at `http://java.sun.com`, November 2002.
7. JSR 177 Expert Group. Security and Trust Services API (SATSA), JSR 177. Available for download at `http://java.sun.com`, October 2003.
8. Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
9. A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Key Sizes. In H. Imai and Y. Zheng, editors, *Public Key Cryptography — PKC 2000*, volume 1751 of *Lecture Notes in Computer Science*, pages 446–465. Springer Verlag, Berlin, Germany, 2000.
10. J. López and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES '99*, volume 1717 of *Lecture Notes in Computer Science*, pages 316–327. Springer Verlag, Berlin, Germany, 1999.
11. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, Jan. 1987.
12. National Institute of Standards and Technology (NIST). Recommended Elliptic Curves for Federal Government Use, July 1999.
13. National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES), Nov. 2001. Federal Information Processing Standards (FIPS) Publication 197.
14. National Institute of Standards and Technology (NIST). Secure Hash Standard (SHS), Aug. 2002. Federal Information Processing Standards (FIPS) Publication 180-2.
15. R. L. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.
16. RSA Data Security, Inc. *PKCS #1 v2.1: RSA Cryptography Standard*, June 2002.
17. Sun Microsystems. Connected Limited Device Configuration, Version 1.0a. Available for download at `http://java.sun.com`, May 2000.
18. Sun Microsystems. Mobile Information Device Profile, Version 1.0a. Available for download at `http://java.sun.com`, December 2000.
19. Sun Microsystems. Connected Limited Device Configuration, Version 1.1. Available for download at `http://java.sun.com`, March 2003.