Analysis and Implementation of Offline-Authentication on Mobile Devices

Bachelor Thesis

by

Daniel Baselt

from
Krefeld

presented at the

Chair of computer networks
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

September 2006

Advisor:
Dipl. Inform. Michael Stini

Acknowledgments

A lot of people supported me during my work on this thesis to whom I wish to express my gratitude.

First of all, I would like to thank Michael Stini for his enduring support and sincere encouragement. He provided me with lots of ideas and comments. The numerous hours he spent for discussion were essential to the quality of my work.

I would like to thank Janis Breitmeier, who provided me an early version of the Bluetooth packages for the DOM framework.

And, of course, I appreciate all the efforts of my colleagues and friends who cross-read this thesis, gave helpful comments and advices or lent me their cell phones for strenuous testing of the programs.

The cliparts used in this thesis are simplified versions from the originals created by Juliane Krug, available at [Lib06].

Contents

List of Figures

List of Tables

List of Abbreviations

CA	Certificate Authority
CDC	Connected Device Configuration
CLDC	Connected, Limited Device Configuration
CRI	Challenge-Response-Identification
DOM	Digital Ownership Management
DSA	Digital Signature Algorithm
ECDSA	Elliptic Curve DSA
ECDLP	Elliptic Curve Discrete Logarithm Problem
IAIK	Institute for Applied Information Processing and Communication
ITU-T	International Telecommunication Union
Java ME	Java Micro Edition
Java SE	Java Standard Edition
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
MIDP	Mobile Information Device Profile
OAEP	Optimal Asymmetric Encryption Padding
PDAP	Personal Digital Assistant Profile
PSE	Personal Security Environment
PRNG	Pseudo-Random Number Generator
PKI	Public Key Infrastructure
PKCS	Public Key Cryptography Standards
RNG	Random Number Generator
RIPEMD	RACE Integrity Primitives Evaluation Message Digest
SATSA	Security and Trust Services API for J2ME
SHA	Secure Hash Algorithm

Chapter 1

Introduction

The market of digital content on mobile devices has been rapidly growing since a few years. In the near future, it will become even more important as new commercial offerings besides the market of device customizations emerge. To take full advantage of these developments, a generic trading system for entirely digital content at any place anytime to anyone is necessary. Transactions of this kind are accomplished only with a certain amount of trust in the trading partners and their offerings. This can be established by offline-authentication, which uses several concepts of cryptography to ensure the confidence in a trading partner.

With the introduction of trust via *offline-authentication*, it will become possible to use modern mobile devices for more versatile purposes. Besides music files and device enhancements like ring tones more interesting and even more valuable content can be offered and traded. Feasible are tickets for public transport, sports events and concerts as well as digital collectibles like trading cards. Existing digital rights management systems allow buying and using digital content only in a very restrictive manner. Therefore, a new trading system is in development at the chair of computer networks at the Heinrich-Heine-University Duesseldorf. The *Digital Ownership Management* project will enable the user to show and prove ownership, as well as barter digital content. Using offline-authentication as presented in this thesis will be one of the project's pillars.

1.1 Problem Statement

The intention of this thesis is to analyze and realize offline-authentication on mobile devices. Since authentication algorithms are already specified in various non-mobile scenarios, their suitability to a mobile environment has to be analyzed. As security of the algorithms is of great importance, the cryptographic background has to be discussed. Existing libraries implementing authentication are to be examined regarding the constraints on mobile devices. Their low computing speed and different operating systems demand special considerations. Moreover, algorithms and licenses for the libraries should be available under an open source license. Tools providing cryptographic support using the chosen algorithms and libraries are to be developed for use on mobile devices as well as on desktop computers suitable to the *Digital Ownership Management* (DOM) project. An offline-authentication demonstration program based on these tools has to be realized as a result to proof the maturity of the technology and that stable usage is possible.

1.2 Contribution

The protocols of mobile devices authenticating each other and of a mobile device connecting to a service provider's server were adapted and described exactly with regard to existing authentication algorithms. Then, the most significant digital signature algorithms were analyzed for use with offline-authentication especially on mobile devices. As conclusion, *RSA* and *Elliptic Curve DSA* (ECDSA) were selected for implementation.
Java was chosen in the *Java Micro Edition* (Java ME) as programming language, offering easy platform portability to most mobile devices. Possible cryptographic libraries for the *Java Standard Edition* (Java SE) and the Java ME from diverse developing groups were examined and rated regarding security issues, free availability, speed and the project's documentation. The *Bouncy Castle Cryptographic Library Provider* turned out to be best fitting to these requirements. During evaluation, test series were programed using Bouncy Castle in combination with Java SE and ME to deduce conclusions towards computational speed of RSA and ECDSA.
A library, the *AuthToolkit*, capable of managing the offline-authentication process, as

well as signing and verifying messages using Bouncy Castle has been developed, which works both with Java SE and ME. The library will be integrated into the DOM project to enable mobile devices to authenticate offline and online as well. Based on the *AuthToolkit*, a demonstration application has been implemented consisting of a desktop server in Java SE and mobile devices using Java ME. It shows how offline-authentication works in a real environment.

1.3 Structure

In Chapter 2, it will be discussed what the theoretical foundations are on which offline-authentication is based and how the concept of a *Public Key Infrastructure* (PKI) and *Challenge-Response-Identification* (CRI) contribute to this problem. Afterwards, selected digital signature algorithms which are of reasonable significance to security and speed on mobile devices are analyzed with regard to their usefulness for implementation. Further on, other security factors of a public key crypto system, like message digests and random numbers, are discussed.
Chapter 3 is considered with the implementation on mobile devices. Providers of cryptographic libraries enhancing the security architecture of Java are examined with special regard to their possibilities on the limited Java ME.
Chapter 4 describes the *AuthToolkit* and its development for the DOM project.
In Chapter 5, the integration of *Bouncy Castle*, the best-fitting security provider for the offline-authentication purpose, into the Java development environment is discussed and tests series are made with Java SE and ME to examine speed of the implementation.
Chapter 6 presents a demonstration program for the implementation of the *AuthToolkit*.
Afterwards, the thesis is summarized and an outlook on possibilities of offline-authentication is made.

Chapter 2

On Theory of Cryptography

Authentication in the digital world requires the collaboration of different techniques used in modern cryptography. With identities being digitized, the first intention is to prevent making a perfect binary copy of an identity. Having that in mind, it is absolutely absurd sending someone else your identity, to let her prove its correctness. This chapter will show how it is possible to satisfyingly convince a counterpart from one's identity. Therefore, the next Sections introduce and discuss relevant solutions for creating a system capable of offline-authentication. Usually, the scenarios described in the following need three parties. According to cryptographic habits, they will be called Alice and Bob as the communicating parties and Eve being the evil counterpart. Eve will try to pretend to be Alice or Bob without being discovered.

2.1 Foundations of Authentication

The most simple way to let two people identify each other in a completely digital environment is to agree upon a secret *pass phrase* (see Figure 2.1). When more than two people want to communicate and each one is assigned a pass phrase, everybody has to know every phrase to assure the identity of a counterpart.
Unfortunately, *Eve* discovers *Bob's* secret phrase simply by becoming a new member at the service he uses. As soon as *Eve* receives *Bob's* pass phrase, she will be able to communicate on the system with his identity, exactly as *Bob* would do.

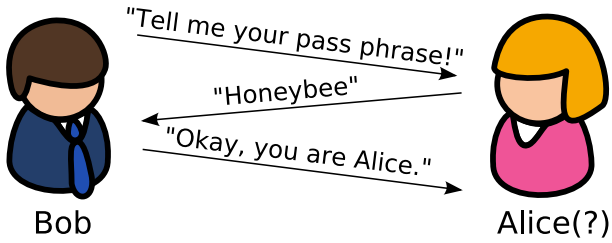

Figure 2.1: Authentication with a Secret Pass Phrase

This can be solved by assigning every possible pairing of users an extra key. A server managing keys and matching users has to store $\frac{n(n-1)}{2}$ keys, with n being the number of users [Buc03], implying complexity of $O(n^2)$. Besides that, all users have to be known to each other, meaning they have to store n keys before any connection can be established, but this is unacceptable for mobile usage. Also key distribution becomes very complicated, regular updates distributing new users' pass phrases are necessary and anybody except the users must not be permitted to access the passes.

2.1.1 Public Key Cryptography

A different approach uses keys to encrypt messages instead of using passwords for identification. It is possible to encrypt and decrypt messages with a special key belonging to one user. This principle is called *symmetric cryptography* (or *private key cryptography*). The drawbacks of key distribution and complexity of key numbers (in $O(n^2)$) are still the same as using passwords. But it paves the way to the use of *public key cryptography* (see Figure 2.2).

Here, in contrast to one shared key for encryption and decryption, two different keys belonging to each other are used. One key is referred to as the *public key*, which can be freely distributed, the other one is called the *private key*, which must not be revealed to anyone. Now, the advantage is that only one key pair is needed for an user instead of one key per possible communication channel, reducing complexity of key management to be in $O(n)$ at the server. Also, the key distribution is alleviated, because it is possible for the server to offer the proper public key to every user requested, although complexity is still in $O(n)$. Less security overhead is necessary, only the correct exchange of a newly

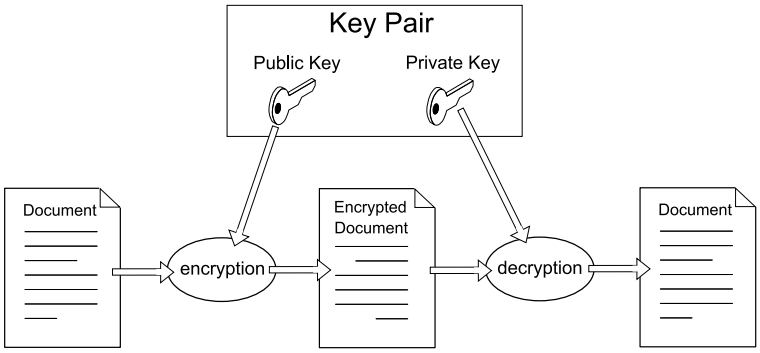

Figure 2.2: Public Key Cryptography

Table 2.1: Comparison of symmetric and asymmetric cryptography

CRYPTOSYSTEM	SYMMETRIC	ASYMMETRIC
Number of keys	$O(n^2)$	$O(n)$
Keys per device	$O(n)$	$O(n)$
Key distribution	private	public

created key pair to its owner and the transmission of the public key to the server have to be assured, as well as that the public directory at the server is read only to users. After the exchange, nobody except the user to which the key pair belongs has to care about the private key. Public key cryptography itself does not offer secure identification. *Eve* may know one of *Alice's* encrypted messages (but not her private key) before *Alice* decides to use it or after she used it and pretend being *Alice* by sending this message, which is called a *replay attack*.

Public key algorithms are based on problems in mathematics which are easy to compute in one direction, but (supposed to be) *hard* to solve in the opposite way (by finding the inverse) without special information, also known as *trapdoor functions*. Hardness means that a problem cannot be computed in polynomial time or better. If a problem was found to be not hard, it would be possible to derive the private key from the public key, which is considered as the *total break* of a public key cryptosystem. For the most important asymmetric algorithms, the underlying problems have neither been proved nor disproved yet. The consequences are discussed in Section 2.2, together with the presentation of chosen algorithms.

To be of comparable strength to private key algorithms, public key algorithms usually need more bits in key length, making public key operations about 10^2 to 10^3 times slower

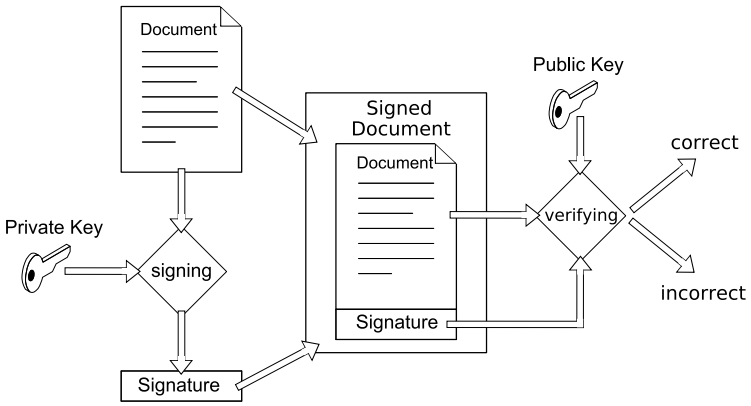

Figure 2.3: Signature and Verification of a Document

than private key cryptography [Sch96]. Usually, this is solved by using a *hybrid cryptosystem*, where not the message itself is encrypted with the public key, but a key of a private key algorithm, which is then used to encrypt messages of the communication session.

2.1.2 Digital Signature Algorithms

When M is the *message space* and C the *ciphertext space*, signing a document $m \in M$ means to perform the *encryption* function $E : M \to C$ on the document with the private key of a public key algorithm and to obtain the signature $c = E(m)$, with $c \in C$. Only the owner of a key pair knows the private key and without it, nobody else can sign the document in the same way.
The signature can be verified by taking the signature and a public key as arguments and compute the *decryption* function $D : C \to M$ to get the encrypted message $m' = E(c)$, with $m' \in M$. If the private and public key belong together, the document itself will be value of the operation, so that $m = m'$ (see Figure 2.3).
To compensate the slowness of public key algorithms, it is not recommendable to sign the whole document, but a hashed version of the document, see Section 2.3 for hashing. This also solves security concerns of some algorithms like *existential forgery* in RSA [Buc03].
Key pairs must not be used additionally to encrypt or sign data besides authentication,

because this is of high risk to security. If *Eve* pretends that she wants to prove *Alice's* identity, *Eve* is supposed to send a random number to *Alice* as challenge. In fact, the random number is the hashed value $h(m)$ of the text m. Alice cannot recognize this and signs the hash value. Actually, when sending the signed hash to *Eve*, *Eve* receives a self-chosen document signed by *Alice* [Buc03]. A solution to this is the use of different hash functions for authentication and encryption, which produce different hash value lengths, so that the length of a hash value determines its purpose.

2.1.3 Challenge-Response-Identification

The *Challenge-Response-Identification* (CRI) protocol is the following:

1. *Alice* wants to identify herself to *Bob*.
2. *Bob* constructs a challenge only *Alice* can solve and sends it to her.
3. *Alice* solves the challenge if, and only if, she knows a certain secret.
4. *Alice* sends her response to *Bob*.
5. *Bob* verifies the solution and accepts *Alice's* identity if her answer was accurate.

Using public key cryptography, Bob would demand Alice to sign a random number r. Alice uses her private key, which is only known to her, to *encrypt* r by computing the *ciphertext* $c = E(r)$. Alice sends the signature c to Bob and he verifies it by decrypting c, he calculates $r' = D(c)$ with Alice's public key and checks, if $r = r'$.

2.1.4 Public Key Infrastructure

When *Bob* wants to validate *Alice's* identity, he needs her public key. Asking *Alice* directly for it is dangerous, because *Eve* can pretend to be *Alice* and send *Bob* a public key matching her own private key. So, to have a trusted third party providing public keys is necessary, where every user may verify the correctness of a key. A CA-server managing the identities is such a trusted third party, called a *Certificate Authority* (CA). It has its own key pair and every member of the CA trusts the CA's signature. Is *Alice* a member at the same authority as *Bob*, he will find her public key at the server's public

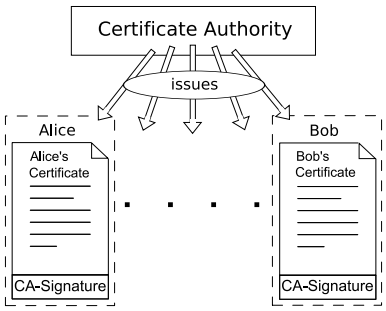

Figure 2.4: A simple Public Key Infrastructure with one level depth

directory read-only, packaged in a digital *certificate* signed by the CA itself. A certificate is a container for a public key and information about the associated user. *Bob* can verify the CA's signature because the CA's public key certificate is known to every member. With it, he can assure that *Alice's* public key specified in her certificate is authentic, otherwise verifying failed. The constellation of a CA with associated members is called a *Public Key Infrastructure* (PKI).
If a key pair becomes invalid because of loss, limitation or noticing illegal operations, a *key revocation* process should be started. Therefore, the key pair is to be marked on the CA's directory (but not deleted) and a new key pair may be created and distributed to the user, depending on the reason of revocation.

2.1.5 Offline-Authentication

With the concepts of CRI and the implementation of a PKI, offline-authentication on mobile devices can be realized. A server providing CA functionality offers the *public key directory*, containing every member's public key certificate signed by the server. A mobile device has to learn the server's public key certificate, as well as its own server-signed public key certificate and private key. The private key should be kept in a *Personal Security Environment* (PSE), an area where only the owner of the key pair has access to. When *Bob* wants to register himself at the CA and has a mobile device of his own acting as PSE to him, cryptographic operations will take place on his device. He generates his key pair on it, so that the private key never has to leave *Bob's* PSE. To let the public key being signed by the CA, he has to securely send his public key to the CA and the new certificate has to be transmitted back to *Bob*. The best way to do this would be, if

Bob went to the CA's administrator personally to exchange information, where also the CA could verify *Bob's* identity. Because this is not always feasible, another possibility is postal communication, as being more expensive but emphasizing Bob's interest to become registered. Then, postal identification procedures like *POSTIDENT* [Pos06] can be realized by postal corporations.
Subsequent, *Bob* receives his CA-signed public key certificate from the server. When not happened during the communication described above, this may even happen using insecure channels, because now *Bob* can check his certificate on the CA directory anytime.
Alice, being also registered with her mobile device at the CA, wants to prove her identity and her CA membership to *Bob* without connecting to the CA-server at this very moment (*offline-authentication*). Therefore, she establishes a connection to *Bob's* mobile device using Bluetooth, infrared, direct cable or a comparable short range communication technology. This type of connections complicate compromising by attacks like the *Man-in-the-Middle-Attack* [Sch96]. *Bob* sends to *Alice* a challenge in form of a random number, which has to be signed by *Alice* with her private key. Afterwards, she sends the signed number and her own CA-signed certificate to *Bob*, who

1. verifies *Alice's* CA-signed certificate with his copy of the CA's public key certificate and
2. verifies the signature of the random number with *Alice's* public key contained in her certificate.

That way *Bob* is assured about *Alice's* identity (see Figure 2.5).
Because neither *Alice* nor *Bob* have reasons to trust each other before the offline-authentication has taken place, *Bob* has to identify himself to *Alice*, too, the same way as described above. When both have made sure that they are indeed connected to the person they expected to, authentication is done and the actual transactions can begin.

2.2 Evaluation of Digital Signature Algorithms

As discussed in Section 2.1.1, public key cryptography is essential to the concept of offline-authentication. The key pair of the Certificate Authority is used to sign users' certificates and each service member uses his or her pair for signing random numbers.

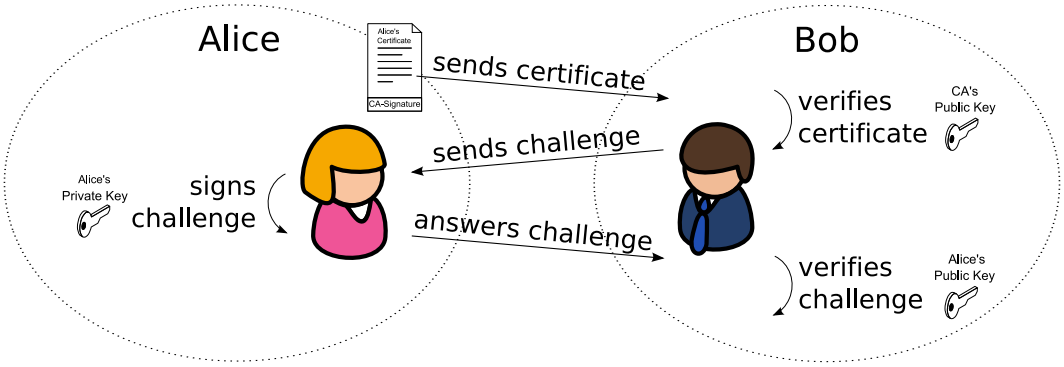

Figure 2.5: Protocol for Offline-Authentication

Therefore, a public key algorithm must be chosen, that is able to work as a digital signature algorithm.
The algorithms discussed in this Section are evaluated considering the demands of cryptography on mobile devices. The criteria are, in order of importance:

Security Ideally, keys for authentication will be valid for years, representing an union with the associated digital identity. Security aspects of the algorithms are possible vulnerabilities of the underlying mathematical techniques and appropriate key lengths to avoid breaking key pairs via brute force in an estimated period of time.

Availability An algorithm should be open source to be able to examine its implementation. Although software patents are not applicable in the European Union, an algorithm should be free to use worldwide.

Speed With each new mobile device generation, processing units become faster, but mobile devices still are rather slow compared to desktop computers. Cryptographic calculations demand operations on very large numbers, while specialized coprocessors are usually not available. The level of security needed for offline-authentication depends on the value of the service for which authentication is intended. The stronger the cryptography, the longer operations last and the more the mobility factor of offline-authentication decreases. So speed of different levels of security has to be observed.

Only three public key algorithms are capable of signing besides encryption: RSA, Rabin and ElGamal [Sch96]. A fourth one, the *Digital Signature Algorithm* (DSA), cannot be

used to encrypt. These algorithms are discussed in this Section. A special variant of the DSA, the *Elliptic Curve DSA* (ECDSA), is described in an own Subsection because of its attributes, which are especially suitable to mobile devices.

2.2.1 RSA

RSA is an abbreviation for the three inventors of this algorithm: Rivest, Shamir and Adleman. It was the first public key cryptography algorithm developed and it is the most frequently used today. And besides it can be used for encryption and signing, it is easy to implement. Due to its popularity, RSA is well-investigated and to all possible attacks discovered yet (e.g. *Chosen-Ciphertext-Attacks* or *Low Exponent Attacks*), countermeasures can be taken [Sch96].
To generate a key pair, Bob chooses two large (more than 100 bits length) prime numbers p and q, $p \neq q$, and computes

$$n = p * q.$$

n is called the *RSA-modulus*.
Additionally, he chooses a natural number e, the *encryption key*, with

$$1 < e < \varphi(n) = (p-1)(q-1) \qquad \text{with } \gcd(e, \varphi(n)) = 1$$

and computes a natural number d, the *decryption key*, with

$$1 < d < \varphi(n) \qquad \text{and} \qquad d * e \equiv 1 \bmod \varphi(n).$$

In other words

$$d = e^{-1} \bmod ((p-1)(q-1)),$$

which can be computed with the *Extended Euclidean Algorithm*, see [Rot05]. The public key is (n, e) and the private key is d.
To encrypt a message $m \in M$ (M being the message space and $0 \leq m < n$), with a public

key, Bob computes

$$c = m^e \bmod n, \qquad \text{with } c \in C \text{ and } C \text{ is the ciphertext space.}$$

If $m \geq n$, m can be split into k subsequent blocks m_i, $i \in 1,\ldots,k$, where each block's length is less than n. Decryption with the private key is computing

$$m' = c^d = (m^e)^d \bmod n.$$

The security of RSA is supposed to be based on the hardness of the factorization problem [Buc03], meaning factoring large numbers. But it is unknown yet if factoring the large modulus n of a key pair is the only way of getting the cleartext message m from the encrypted message c and the public key (n,e), with e being *coprime* to $\varphi(n)$, meaning that $\varphi(n)$ and e have the greatest common divisor 1 [Buc03].
RSA in this form is vulnerable to chosen-message-attacks. To prevent this, it is recommended to follow the recommendations of the *Public Key Cryptography Standards* (PKCS) devised by the *RSA laboratories* [RSA05]. PKCS#1 describes methods of hardening RSA against the above mentioned attack sufficiently. RSA is free to world-wide use since the year 2000, as the patent held by the RSA Security Inc. [RSA06] in the United States expired.
Evaluating the costs of RSA, encrypting requires an exponentiation modulo n. Modular exponentiation is performed by a series of modular multiplications. With a smaller exponent e encryption is sped up, but when the exponent is too small, so-called *Low-Exponent-Attacks* are possible [Buc03]. A common value for e, recommended by X.509 [Gro02], is $(2^{16}+1)$, which is a smart choice because it takes only 17 multiplications to exponentiate. It is possible to choose the same value for e for all key pairs as long as d differs. Decryption is also an exponentiation modulo n, but this time the exponent d has to be about the same size as n. Otherwise, the system becomes insecure. The complexity of usual RSA encryption and decryption implementations is in $O(k^2)$ to $O(k^3)$, with k being key bit length, and key generation is in $O(k^4)$ [RSA05].
Key generation is necessary only when a new client joins to a service or an existing key expires. Signing and verifying speeds are acceptable, see table 2.2.1 and the results of the speed tests in Chapter 5. Recommended key lengths are dependent on the estimated computing capacity of modern computers to break a key pair. Today, key lengths are at minimum 1024 bits and if a key pair should be valid for several years, it would be best to use a 2048 bits modulus.

Table 2.2: RSA Speeds for different modulus lengths with a small public key of 8-bit measured on a SPARC II [Sch96]

STRENGTH	512 BITS	768 BITS	1024 BITS
SIGN	0.16 sec	0.52 sec	0.97 sec
VERIFY	0.02 sec	0.07 sec	0.08 sec

RSA's unproved hardness is its greatest disadvantage. But this can be balanced with two arguments: Its widespread use and (because of that) the big interest of cryptanalysts. The more people are interested in the security of RSA (and with it factoring large numbers), the bigger is the chance that a breakthrough will become public. The contrast to this would be, if a secret service or a company's research team broke RSA and everybody else would still think RSA being secure. But being analyzed for over thirty years, the "risk" of breaking RSA is very small.
Taking all this into account, usage of RSA is very interesting to offline-authentication. In combination with hash functions (see Section 2.3), known weaknesses like the *existential forgery* and *RSA multiplicativity* become impracticable [Buc03].

2.2.2 Rabin

The Rabin public key cryptosystem, named after its inventor Michael O. Rabin, is closely related to the RSA cryptosystem. Its security is also based on the factorization problem, but in contrast to RSA, it is provably equivalent to the factorization problem and therefore considered secure [Sch96]. Encryption with Rabin is a little more efficient than with RSA, while decryption is about the same costs. The disadvantage of Rabin is its vulnerability to *Chosen-Ciphertext-Attacks*. These are attacks where *Eve* has temporary access to the decryption machine and chooses ciphertexts to get the corresponding messages. With them, *Eve* can compute the private key and totally break Rabin. This is why Rabin has only few significance in practice. In the offline-authentication scheme, the attack can be accomplished by encrypting a message with the public key and then sending these randomly seeming bytes as challenge to sign. The answer to the challenge *Eve* receives is the decrypted message that can be used to break Rabin. Therefore, it is not reasonable to use Rabin for offline-authentication.

2.2.3 ElGamal

ElGamal's security is based on the hardness of the *discrete logarithm problem*. It describes the difficulty of calculating discrete logarithms in a finite field. To create a key pair, a prime p and two random numbers g and x have to be chosen, with g and $x < p$. By calculating

$$y = g^x \bmod p,$$

the public key is (y, g, p) and the private key is x.
Signatures of a message $m \in M$, M is the message space, are made by *Bob* choosing a random number k being *coprime* to $(p-1)$. It is very important that k is random and is never used twice, otherwise *Eve* can recover the private key x [Sch96]. *Bob* computes

$$a = g^k \bmod p$$

and uses the *Extended Euclidean Algorithm* to solve the following equation for b:

$$m = (x * a + k * b) \bmod (p-1).$$

Now, (a, b) is the signature of M; k must be kept secret.
Alice verifies the signature by confirming that

$$y^a * a^b \bmod p \equiv g^m \bmod p.$$

It is necessary for m being a message digest (Section 2.3), otherwise messages and appropriate signatures (a, b) can be deduced from another [Buc03]. ElGamal is based on the *Diffie-Hellman key agreement protocol* and the operations to sign and verify are very similar to it [Buc03]. No patents cover the use of it. ElGamal signing requires computing one modular exponentiation and one *Extended Euclidean*. The calculation of a and y^a are message-independent, they can be computed and stored before the actual verification. Then, ElGamal is faster than RSA (with only one exponentiation), but the computations have to be kept secret on the mobile device.
Verification demands three modular exponentiations, two more than RSA. But the verification process can be altered so that computations equivalent to one exponentiation are necessary [Buc03]. Key sizes of ElGamal are about the same compared to RSA for an equal level of security. The ciphertext is double the size of the corresponding

message, but this does not matter to offline-authentication on mobile devices as long as only message digests and random numbers of relatively short bit lengths are computed. One interesting advantage of ElGamal is the possibility to implement it in any cyclic group other than the prime residue group modulo a prime. If a way is found to calculate discrete logarithms in $(\mathbb{Z}/p\mathbb{Z})^*$, with p prime, then ElGamal can be reimplemented in another cyclic group, where discrete logarithms are still hard to solve [Buc03]. Such a group are elliptic curves over finite fields, discussed in the following Section about an ElGamal derivative, the *Digital Signature Algorithm*.

2.2.4 The Digital Signature Algorithm

The so-called *Digital Signature Algorithm* (DSA) is a variant of the ElGamal signature algorithm. It is used in the *Digital Signature Standard* proposed from the United States' *National Institute of Standards and Technology* (NIST) and specified in FIPS 186-2 [NIS00].
To generate a key pair, Bob chooses a prime number q of 160 bits length and a prime p with $2^{511+64t} < p < 2^{512+64t}$ for $t \in \{0,1,\ldots,8\}$. q is to be a divisor of $(p-1)$. Then Bob computes

$$g = h^{(p-1)/q} \bmod p, \text{ with } h \in \{2,3,\ldots,p-2\} \text{ and } h^{(p-1)/q} \bmod p > 1.$$

He chooses a number x with $0 < x < q$ and determines

$$y = g^x \bmod p.$$

The public key is (p,q,g,y) and the private key is x.
To sign a message m, Alice generates a random number $k < q$. Afterwards, she computes the hash $H(m)$ from m with a one-way hash function. The standard specifies use of the *Secure Hash Algorithm* discussed in Section 2.3. Furthermore, she calculates

$$r = (g^k \bmod p) \bmod q,$$
$$s = (k^{-1}(H(m) + xr)) \bmod q.$$

The signature is the pair (r,s). Bob verifies the signature by computing

$$\begin{aligned} w &= s^{-1} \bmod q, \\ u_1 &= (H(m) * w) \bmod q, \\ u_2 &= rw \bmod q, \\ v &= ((g^{u_1} * y^{u_2}) \bmod p) \bmod q. \end{aligned}$$

By checking that $v = r$, the signature is verified [Sch96].
Because of modular exponentiations of fixed 160 bits length, DSA is faster than ElGamal, which needs modular exponentiations of the length of the module p. Moreover, DSA can be sped up analogue to ElGamal by precomputing the message-independent values r and k^{-1}.
The security of DSA, as an ElGamal variant, is also based on the discrete logarithm problem. One way to attack DSA is the *existential forgery*, which can be prevented by checking

$$1 \leq r \leq q-1 \text{ and } 1 \leq s \leq q-1$$

before verification of a signature, another way are algorithms alleviating computation of the logarithm problem. The best known algorithms like *Shanks* or *Pohlig-Hellman* [Buc03] still need more than $\sqrt{q}$ steps to solve a logarithm problem of q bits. Because $2^{159} < q < 2^{160}$ in DSA, at least 2^{79} operations are necessary, which can be considered secure [Buc03].
A U.S. patent attributed to David Kravitz, a former NSA employee, covers DSA, but it was made available world-wide royalty-free. Although DSA is efficient, secure and freely available, it is not implemented in the offline-authentication toolkit, but a variant of it, elliptic curve DSA described in 2.2.5, which implements DSA in another cyclic group and offers even more advantages.

2.2.5 Elliptic Curve DSA

Cryptographic algorithms basing on the discrete logarithm problem can be improved by implementing them in the cyclic group of elliptic curves over finite fields. By choosing

a certain curve, it is possible to apply speed enhancements utilized in $(\mathbb{Z}/p\mathbb{Z})^*$, making signing and verifying faster, but preventing the application of algorithms simplifying the logarithm problem, like *Shanks* or *Pohlig-Hellman* [Buc03].
The advantage of this is smaller key sizes to gain the same security level compared to non-elliptical public key cryptosystems (see table 2.2.5). The downside are more complex mathematical computations for generating signatures, so that implementations of elliptic curve cryptosystems in spite of smaller keys are not necessarily faster.
Elliptic Curve DSA (ECDSA) is a variant of DSA based on the *Elliptic Curve Discrete Logarithm Problem* (ECDLP). The risk of developing new algorithms breaking the ECDLP is reduced, because elliptic curves were of interest in mathematics a long time before the application to cryptography was considered.
Elliptic curves do not represent curves or even ellipses in the common sense, but are points solving the equation

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

Selecting an appropriate curve requires a good understanding of the mathematics of elliptic curves and is troublesome to implement. Therefore, the NIST published recommended domain parameters of elliptic curves to use [NIS99]. The choice of a certain field is significant for overall performance, so for implementation, finite fields of odd characteristic ($\mathbb{F}_p$, where $p > 3$ is a large prime number) and fields of characteristic two, $\mathbb{F}_{2^m}$, are considered best [LD00].
To create a key pair, the domain parameters (q, FR, a, b, G, n, h) of the underlying curve E have to be chosen:

- q specifies a prime power,
- FR describes the method of representing field elements $\in \mathbb{F}_q$, with $q = p$ or $q = 2^m$,
- a, b are two field elements $\in \mathbb{F}_q$ specifying the equation of the curve,
- G is the base point $G = (x_G, y_G)$ on $E(\mathbb{F}_q)$,
- n is a prime of the order of G,
- h is an integer, which is the cofactor $h = \#E(\mathbb{F}_q)/n$.

Table 2.3: RSA and ECDSA key lengths for equivalent security in [bits] [NIS06]

ALGORITHM	RSA	ECDSA
STRENGTH	1024	160 - 223
	2048	224 - 255
	3072	256 - 383
	7680	384 - 511
	15360	512+

The private key is a random integer $d \in [1, n-1]$ and the public key is $Q = dG$.
Alice signs a message m by selecting a random integer $k \in [1, n-1]$ and calculating

$$r = x_1 \bmod n, \text{ where } (x_1, y_1) = kG,$$
$$s = k^{-1}(H(m) + dr) \bmod n,$$

where $H(m)$ is the hash value of m. If $r = 0$ or $s = 0$, she has to select another random number and compute the signature again, otherwise, the signature is the pair (r, s).
For verification, *Bob* checks that r and s are integers in $[1, n-1]$. Then he hashes the message m to obtain $H(m)$ and calculates

$$w = s^{-1} \bmod n,$$
$$u_1 = H(m)w \bmod n,$$
$$u_2 = rw \bmod n,$$
$$(x_1, y_1) = u_1 G + u_2 QA.$$

The signature is valid if $x_1 = r \bmod n$.
ECDSA over $\mathbb{F}_p$ is considered secure with key lengths of 192 bits, in contrast to DSA, which needs key lengths up to 1024 bits or RSA with 2048 bits. The short key lengths are an advantage on mobile devices regarding memory consumption and computing time, as far as the elliptic curve algorithms are optimized well in implementation to benefit from that. Test series with Bouncy Castle using ECDSA concerning performance are described in Chapter 5. ECDSA is not covered by any patents.

2.3 Cryptographic Hash Functions

A message digest h is a preferably random fixed-length representation of a message m of arbitrary length, created by a one-way hash function $h = H(m)$. One-way hash functions are hash functions with additional characteristics. While it is easy to compute h from m, the other way finding m from h is hard. Also, it is hard to find another message m' to m, such that $H(m) = H(m')$. Message Digests are used by digital signature protocols to shrink the number of bytes to sign, because signing is slow and message digests, ranging from 160 bits to 512 bits, are usually shorter than the original message. Several signature protocols even need hashing for security reasons (see DSA in 2.2.4).
Reducing a message to a shorter representation, comparable to a fingerprint of the message, may enable Eve to find two random messages m and m' with $H(m) = H(m')$, also called *birthday attack*. If it is hard detecting two *random* messages with the same hash value, the algorithm is called *collision-resistant*. Using a non-collision-resistant algorithm for digital signatures alleviates the search of different messages with the same hash value, so that it cannot be determined, which message was intentionally signed.

A popular family of hash functions, the MD (short for *Message Digest*) family, is designed by Ronald Rivest, with the fifth one, MD5 from 1991, being the newest and strongest. While it is mainly used to create checksums of files downloadable from the internet or storing passwords in databases, it is considered insecure due to its hash length of only 128 bits, allowing birthday attacks. In 2005, Lenstra, Wang and de Weger showed two different certificates (see 2.5) with the same hashsum [LWdW05].

Another family of hash functions is the SHA family (short for *Secure Hash Algorithm*), proposed by the NIST as a Federal Information Processing Standard (FIPS), see [NIS02], and being technically similar to the MD family. The most common representative is SHA-1, which is often used in a variety of cryptographic applications. Its predecessor, SHA-0, has already been proved to be vulnerable against collision attacks, reducing its complexity from 2^{80} to 2^{39}, so it can be considered broken [WYY05], while SHA-1's complexity has been reduced from 2^{80} to 2^{63} [Sch05]. This makes SHA-1 not insecure yet in signing applications, because finding a document with the same hash value as another already signed document is still not feasible, but better attacks are expected to come. The family was enlarged by the SHA-2 algorithms SHA-224, SHA-256, SHA-384 and SHA-512, each creating according hash lengths, while the variants SHA-0 and

SHA-1 create 160 bits hashes. The SHA-2 group is technically very similar to SHA-1: Although no weaknesses of SHA-2 have been found yet, they are expected to come. Nevertheless, the longer hash sums provide secure use for the next years, as long as no new critical mathematic weaknesses in the SHA family are discovered. All SHA-2 algorithms are covered by a U.S. patent held by the NSA, while SHA-1 is free to use.

A third group of hash functions is the *RIPEMD* (*RACE Integrity Primitives Evaluation Message Digest*) family. The first member of the group, the original RIPEMD, is considered insecure due to collisions found for MD4, on which RIPEMD is based [DBP96]. Therefore, RIPEMD-128 was developed, producing a 128 bits hash length like RIPEMD, but seeming collision resistant yet. A stronger variant is RIPEMD-160, computing 160 bits output, while two other successors, RIPEMD-256 and RIPEMD-320, only reduce the risk of collisions. The algorithms are developed in an open community and none of them is covered by patents [Bos04]. Compared to SHA-1, RIPEMD-160 is faster, but not as widespread and well-investigated as NIST's algorithm. Since security is more important to offline-authentication, use of SHA-1 is recommended.

2.4 Random Numbers

Digital signature algorithms use randomly chosen key pairs for signing operations. If *Eve* discovers, that *Alice's* key pair was not randomly picked, but after a certain procedure, she may be able to reconstruct it and sign messages exactly like *Alice* does. So, determinable random numbers may lead to the collapse of even the best public key system. Ideally, random numbers have to be chosen by a *Random Number Generator* (RNG) being completely random, but as far as the numbers are picked by a deterministic working mobile device, only a *Pseudo-Random Number Generator* (PRNG) is at hand, which lets numbers only seem random, although they are reproducible. There exist different approaches to create the impression of real randomness with a PRNG, which seem to work good as long as the methodology of the random source is not reconstructible. A common idea is to take the day's time and date value as starting point and apply techniques to it to diversify the resulting value, like hashing it with a one-way hash function. This obscures the original time value, making it impossible to recreate it without further knowledge about the time interval from which the number was chosen.
Other variants evaluate key pressing intervals or mouse motions of the user, but each is

traceable to *Eve*. A possibility that seems to work well and may be worth implementing is the recording of atmospheric noise sent by a radio and picked up by a mobile device's microphone [Sim05]. The Java programming language offers the class `SecureRandom`, which offers a cryptographic strong PRNG, meaning that it is considered seeming random "enough" for cryptographic use.

2.5 Certificate Formats

The offline-authentication scheme presented in 2.1.5 requires a standardized method of exchanging identification information between the authenticating parties, including unique identifiers and public keys. Therefore, *digital certificates*, representing verifiable links between identifiers and the corresponding public keys, are exchanged. A popular certificate format used to describe the contained information is *X.509* [IT05], specified by the *International Telecommunication Union* (ITU-T), which stores data in an XML tree and is very flexible to the content. In a closed environment, where certificates are not used for other means than offline-authentication, a self-designed format containing only minimal structure, like simple separation of concatenated information, may also be chosen, because of faster parsing on mobile devices with limited computing power.
The following information should be contained in a certificate, following to [Buc03]:

- the certificate's unique ID number
- the CA's name
- *Bob's* unique identifier
- his public key
- the public key's algorithm name
- beginning and ending date of the certificate's validity
- information about restricted use of the public key to certain applications

Using a proprietary certificate format for offline-authentication, some entries may not be necessary. If only one algorithm is used, its name does not have to be mentioned. Also, if only one CA exists, its name is known to every user. The application to use the certificate with is clear, too. Additional information is needed, when the certificate is to be signed, like the name of the signer, the corresponding certificate's ID and the signing algorithm. See Section 4.3 for the implementation in the *AuthToolkit*.

Chapter 3

Implementation of Offline-Authentication in Java ME

3.1 Introducing Java ME

The *Java 2 Micro Edition* (Java ME) from *Sun Microsystems* [Mic06a] is a derivate of the *Java 2 Standard Edition* (Java SE) with focus on the special demands of limited devices. Like the Java SE Bytecode, every program compiled in Java ME Bytecode can be interpreted by a Java ME virtual machine, called KVM (K for kilobyte) instead of Java SE's JVM. Since mobile devices are less powerful than desktop computers, they are referred as limited devices, meaning their CPU is rather slow and their memory is small compared to machines supporting the Java SE. The Java ME consists of three parts: *configurations, profiles* and *optional APIs*.

The *configuration* is the base part, determining a subset of Java SE's API. Which configuration is supported by a device, depends on its hardware capabilities. Two different configurations exist: The *Connected Device Configuration* (CDC) used on faster PDAs or set-top boxes and the *Connected, Limited Device Configuration* (CLDC), implemented often on cell phones. Because of the wide spreading of Java ME compatible cell phones, offline-authentication has to be runnable on the CLDC as the lowest common denominator.

Profiles expand the API with device-specific user interfaces and storage functions; cell phones are usually compatible with the *Mobile Information Device Profile* (MIDP), which will be used in the following Sections, and PDAs use the *Personal Digital As-*

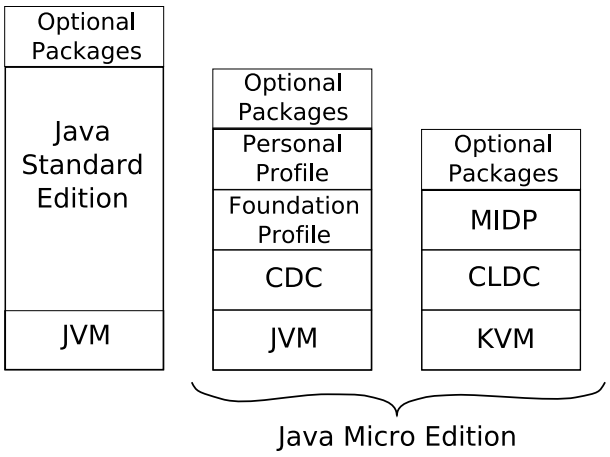

Figure 3.1: Java SE and Java ME Stack Components

sistant Profile (PDAP). Profiles for the CDC are more flexible and offer a more complex API, but they will not be target of development in this thesis.
Optional APIs may be on top of the profiles, like a Bluetooth or 3D gaming API for a specific MIDP compatible device. Figure 3.1 provides an overview, please refer to [LK05] to get a deeper understanding of Java ME.

3.1.1 The Wireless Toolkit

Sun offers the *Sun Java Wireless Toolkit*, that allows developing software for Java ME. The most comfortable way to install it, when already using Sun's *Netbeans IDE* [Mic06c], is to download the *Netbeans Mobility Pack*, which includes the *Wireless Toolkit*. This way, the extra functionality is integrated into the existing IDE. Attention has to be paid to the provided mobile device emulator for testing programs in development. It may work much smoother or slower than the real devices and, like the various desktop computers able to run the Java SE, ME devices vary enormous in speed from one model series to another, which is especially important when it is necessary to compute complex cryptographic algorithms.
Moreover, vendors of Java ME compatible devices do not implement every part of the Micro Edition with the necessary accurateness, so applications may run flawlessly in the emulator, but produce different errors from one phone model to another. During development of the AuthToolkit demonstration application, it showed that especially *optional APIs* like Bluetooth are affected by this. As an approach to this problem, vendors like

Nokia or *Sony-Ericsson* offer own Java ME emulators, which indeed behaved differently when confronted with the application. But creating platform independent solutions is made difficult by this.

3.1.2 Java ME Profile and Configuration Versions

The Java ME's CLDC configuration with MIDP profile provides only a small subset of the bigger Java SE, which affects complete parts of the Java language a high level language programmer is accustomed to. For example, the CLDC 1.0 does not support floating point numbers at all, while they are implemented in CLDC 1.1, which makes the newer version recommendable.
Many mobile devices already support the CLDC 1.1 together with MIDP 2.0, which is the targeted platform for development in the following Sections. Unfortunately, still the complete API for cryptographic computations is absent in these versions.

3.1.3 Java Cryptography Architecture in Java SE and ME

To understand Sun's two different cryptographic modules for the Java language, a glance at history is helpful. From version 1.1, Java is bundled with the module *Java Cryptography Architecture* (JCA), a library with interfaces for signing and hashing. In Java 1.2, another API was introduced: the *Java Cryptography Extension* (JCE), offering functions for several symmetric and asymmetric algorithms. But due to U.S. export regulations in force at that time, cryptographic source code was treated the same way as weapons and the export of the *Java Software Kit* to other countries was prohibited. Therefore, Sun split the JCA and the JCE and put all classes affected by U.S. export laws into the JCE and the non-prohibited into the JCA, enabling delivering Java with the JCA outside the U.S. - without the JCE. The laws changed in 1999, making it possible to bundle Java version 1.4 with a weaker JCE for export. Weak in this context refers to the maximum key length of the supported algorithms, allowing to easily break encryption by brute force. Today, after another change of the regulations in 2004, the only difference between the weak and strong JCE is a file containing security policies. U.S. law permits downloading and replacing the file with one from Sun's website to gain "unlimited strength" support.
The JCA offers a set of classes in the package `java.security` for transparent im-

plementation and use of cryptographic functions like encryption or hashing, which enables other developers than Sun to design own cryptographic service providers. The programmer does not need to worry which certain *cryptographic service provider* is used. In Java ME, Sun excluded the JCA from the CLDC to downsize the consumption of disk space on mobile devices, but some providers created special mobile editions of their frameworks that also regard the technical circumstances on mobile devices.

3.2 Java Security Provider

Several security providers exist which enhance Java's cryptographic abilities. Only the more widespread are discussed here to benefit from a bigger number of testers, which guarantees greater code maturity. The criteria to evaluate the security providers are pretty much the same as for the digital signature algorithms in Section 2.1.2: Besides the availability of RSA and ECDSA, all implementations of discussed algorithms in Chapter 2 should be flawless regarding security aspects, the sources should be open (also to review the implementation), as well as free of charge and the provider should run fast. Ideally, a separate mobile edition exists that reimplements Java ME's lack of necessary data types for cryptography like `BigInteger`.

3.2.1 Sun JCE

Sun's JCE is the standard *cryptographic service provider* in Java SE and can be found in the package *javax.crypto*. It supports a variety of symmetric algorithms, but only a few asymmetric like RSA, while ElGamal, DSA and ECDSA are not implemented. The source code is not available and it cannot be integrated into mobile projects because of lacking classes like `java.math.BigInteger` and `java.math.SecureRandom`.

3.2.2 SATSA

The *Security and Trust Services API for J2ME* (SATSA) by Sun realizes parts of the JCE as optional API adding cryptographic functionality to Java ME [Mic06d]. It implements

the digital signature algorithms RSA and DSA, but no elliptic curve cryptography. By supporting *smart cards* as security elements, cryptographic operations can be performed in a trusted environment on mobile devices. Unfortunately, devices are not necessarily supplied with smart cards (or the programmer does not has access to them, like a SIM card in a cell phone), nor any cryptographic hardware, so the functionalities have to implemented in software, since the cryptographic provider has to be runnable on every Java supporting mobile device. Although SATSA is free to use, its source code is not available.

3.2.3 Bouncy Castle

The *Bouncy Castle Crypto API* [otBC06], developed by the *Legion of the Bouncy Castle*, is a completely free and open source cryptographic service provider. It is a *Clean-Room-Implementation* of the JCE (meaning, that it supports the same interfaces, but is developed from scratch) and supports every algorithm asked for - and even a lot more. A *lightweight version* for limited mobile devices is available, too, which realizes the missing `BigInteger` and `SecureRandom` for use with Java ME.
The downside of Bouncy Castle as being a voluntary project is, that there is no support for the package except a very poor class documentation. Without searching for further documentation on the internet and raking around in some of the rare books covering Bouncy Castle like [Hoo05], it is very hard to get it to work (see Section 8.1). While Java SE offers the well-documented JCA interfaces for use with Bouncy Castle, the lightweight ME version requires direct work with the Bouncy Castle classes, involving examining the source code for understanding the gearing of the package.

3.2.4 Cryptix

The *Cryptix* project [Cry04], used in the *iClouds* project of the *Darmstadt University of Technology* [SH04], is another *Clean-Room-Implementation* of the JCE. Although it supports many algorithms, it lacks support for elliptic curves and offers no Java ME version. Since support and development on the project seem to have stopped in 2004, it cannot be considered an option for ambitious cryptographic applications. A subproject

for elliptic curves, *Elliptix*, is in a pre-alpha state since 1999 and has not been updated anymore.

3.2.5 IAIK

The *Institute for Applied Information Processing and Communication* (IAIK) of the *Graz University of Technology* [Gra05] provides a *Clean-Room-Implementation* of the JCE that is free of charge for research and educational purpose, but not for commercial use. A separate lightweight mobile version, the *IAIK JCE-ME*, is available. The IAIK JCE has a lot of different security algorithm schemes implemented and an optional API for elliptic curve cryptography is offered. The main advantage over Bouncy Castle is the availability of support for IAIK's products, the drawback of the commercially orientation is the closed source. IAIK advertises the speed of the algorithms as a design focus.

3.2.6 Conclusion

Bouncy Castle and the IAIK JCE are both very interesting, but have downsides that have to be evaluated. Unfortunately, open source and free of charge appear to be a contradiction to a good documentation and product support. The lack of documentation of the Bouncy Castle project can be countervailed with voluntary reports on the internet, but it is not guaranteed, that the project will be continually developed. IAIK on the other hand keeps its sources under lock and key. Since free sources weigh more to the demands of the offline-authentication project, Bouncy Castle has been chosen for implementation. Because speed of Boncy Castle's algorithm implementations can only be guessed, speed measurements are taken in Chapter 5.

Chapter 4

The AuthToolkit

The *DOM* project requires secure offline-authentication for scenarios like offline bartering. Therefore, the *AuthToolkit* project has been developed as result of this thesis to support authentication and signing with Java ME and SE. With it, offline-authentication of mobile devices is made possible by using short-range communication connections, that can also be used for flow of application data after authentication (or before, if necessary). Cryptographic operations in the *AuthToolkit* are performed by the *Bouncy Castle lightweight API*, which is implemented as security provider. Algorithms for digital signatures that were to be integrated are RSA and ECDSA, both with the SHA-1 hash function. The *AuthToolkit* can be used as base for other cryptographic extensions of the DOM *project*, like data encryption.
Since it has to be evaluated, whether mobile devices available now and in the near future will support offline-authentication based on this software with acceptable speed, the testing applications *BCSpeedSE* and *BCSpeedME* were written. They are presented with testing results in Chapter 5. A demonstration application showing the possibilities of the AuthToolkit has been developed and is presented in Chapter 6.

4.1 Recommendations

Since the *AuthToolkit* will be used in the *DOM* project, general recommendations set for *DOM* have to be fulfilled by the *AuthToolkit*, too. It has to use cryptographic libraries free of charge and open source, which is complied by the Bouncy Castle security

provider and RSA, respectively ECDSA, together with SHA-1. For flexibility toward cryptanalytic progress and because of unknown speed of the Bouncy Castle provider, both algorithms have to be implemented.
Besides offline-authentication, the *AuthToolkit* should be applicable to any scenario where authentication is demanded. Hence the project performs authentication by using arbitrary input and output streams to other devices, which also allows online-authenticaion. Moreover, the *AuthToolkit* is designed generic to allow integration into projects for Java ME and SE, making authentication between any Java supporting machines possible.
The components integrated into the *AuthToolkit* comprise signature methods, certificate objects and the *Challenge-Response-Identification* for authentication as described in Section 2.1.3.

4.2 Structure

The *AuthToolkit* is realized as a mobile library, enabling the implementation into Java ME as well as into Java SE projects. Offline-authentication as presented in this thesis needs digital signatures and certificates, this is why the project provides three packages: `signature`, `certificate` and `authentication`.

Package `signature`

The package `signature` offers an interface and three classes implementing it:

Signer This is an interface for arbitrary signature algorithms, that classes being part of the signing package should conform to. It offers functions for key pair generation, signing and verifying generic messages. Implementations of `Signer` allow digital signature operations not limited to certificates but any content.

RSASigner An implementation of the `Signer` interface for the RSA algorithm. For key generation, it allows specifying the key length, the prime number certainty (see Section 5.2 for an explanation) and the public key, but also offers “default” values sufficient for most applications on mobile device: 1024 bits and a certainty of $(\frac{1}{2})^{25}$.

RSAOAEPSigner Mainly equivalent to `RSASigner`, enhances `RSAOAEPSigner` it with the *Optimal Asymmetric Encryption Padding* (OAEP) encoding, making RSA secure against *chosen ciphertext attacks* [BR95]. The downside of this method are effectively smaller block sizes, which slows down signing of longer hash functions like SHA-512 by a factor of 2, depending on the length of the RSA modulus.

ECDSASigner A third implementation of the `Signer` interface, which integrates the ECDSA algorithm. When generating keys, prime curves with 192, 239 and 256 bits are available. These curves are *prime192v1*, *prime239v1* and *prime256v1*, as defined in *ANSI X9.62*. Like the `RSASigner` default values, a default curve is available (192 bits).

If other signature algorithms become necessary for the *DOM* project, they can easily be added by another implementation of the `Signer` interface.

Package `certificate`

The `certificate` package contains classes for managing certificates. Only one format is used in offline-authentication and therefore the class `AuthCert` has been developed. An `AuthCert` instance represents a certain certificate as defined in Section 4.3 together with the corresponding signature and offers access to the certificate's fields.

Package `authentication`

The `authentication` package copes with the process of authenticating devices. The class `AuthManager` initializes and processes the authentication cycle between two devices. Calling an AuthManager object and passing it a connection's `InputStream` and `OutputStream` is the only action a developer has to perform to implement authentication. The process performs without any need of user interaction until completed, which can be checked by reading status fields of the AuthManager. Then the exchange of application data can begin.
AuthManager is adapted to work with Java ME peculiarities like the `Display` class as user interface and uses the Bluetooth `LocalDevice` and `RemoteDevice` to get

information about devices. When these objects are not passed to AuthManager, the commandline is used as output and Strings describe the connected parties. This enables authentication also with Java SE and with any connection where streams are available, for example WLAN.

For detailed information on every method of the classes, *JavaDocs* have been created and are located on the CD in the folders of the *AuthToolkit* project.

4.3 Implementation of Certificates

The certificate format used by the AuthToolkit is not according to X.509 to minimize the computation time on mobile devices. Although parsers are available that process the standardized X.509 format, the flexibility it offers is not needed for the offline-authentication purpose and so the management overhead can be avoided. Hence a simple separation symbol (a new line) between each entry is defined and the specified entries are:
C (Country), ST (State), CN (Common Name), unique identifier, public key, according key algorithm, the public keys length, a “valid from” date and a “valid to” date, the hash function used to sign the certificate, as well as the signature itself.
The order of the entries is fixed and can be seen in the sample in Figure 4.1.

4.4 Authentication Protocol

The flow of messages for offline-authentication implemented into the authentication package of the *AuthToolkit* project follows the considerations of Section 2.1.5. The difference is an extension to authenticate not only one party to another but both to each other. The actual information exchanged is shown in Figure 4.2.

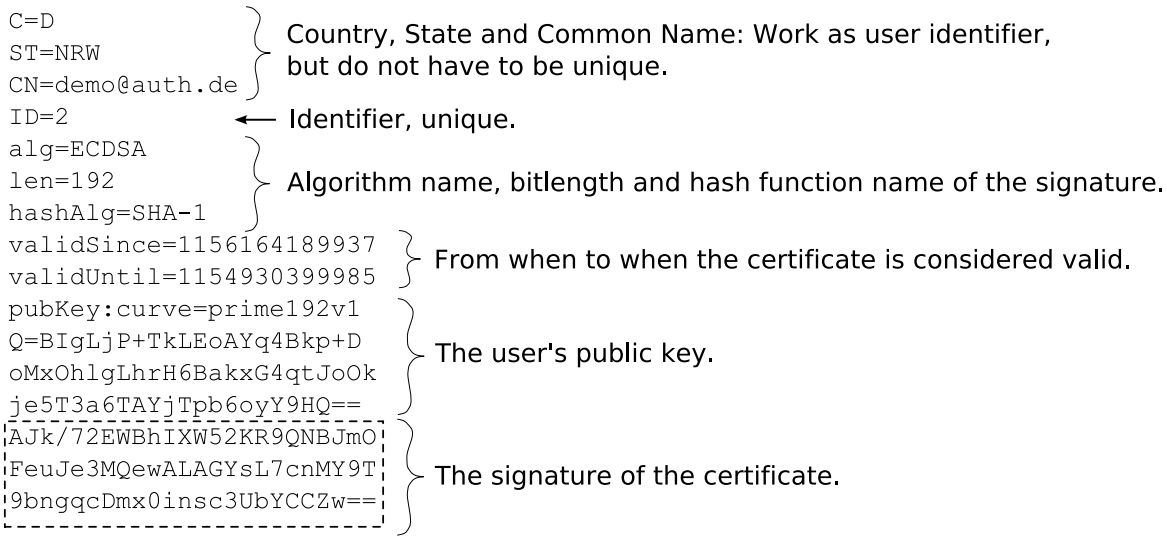

Figure 4.1: Sample certificate representing the structure used in offline-authentication certificates.

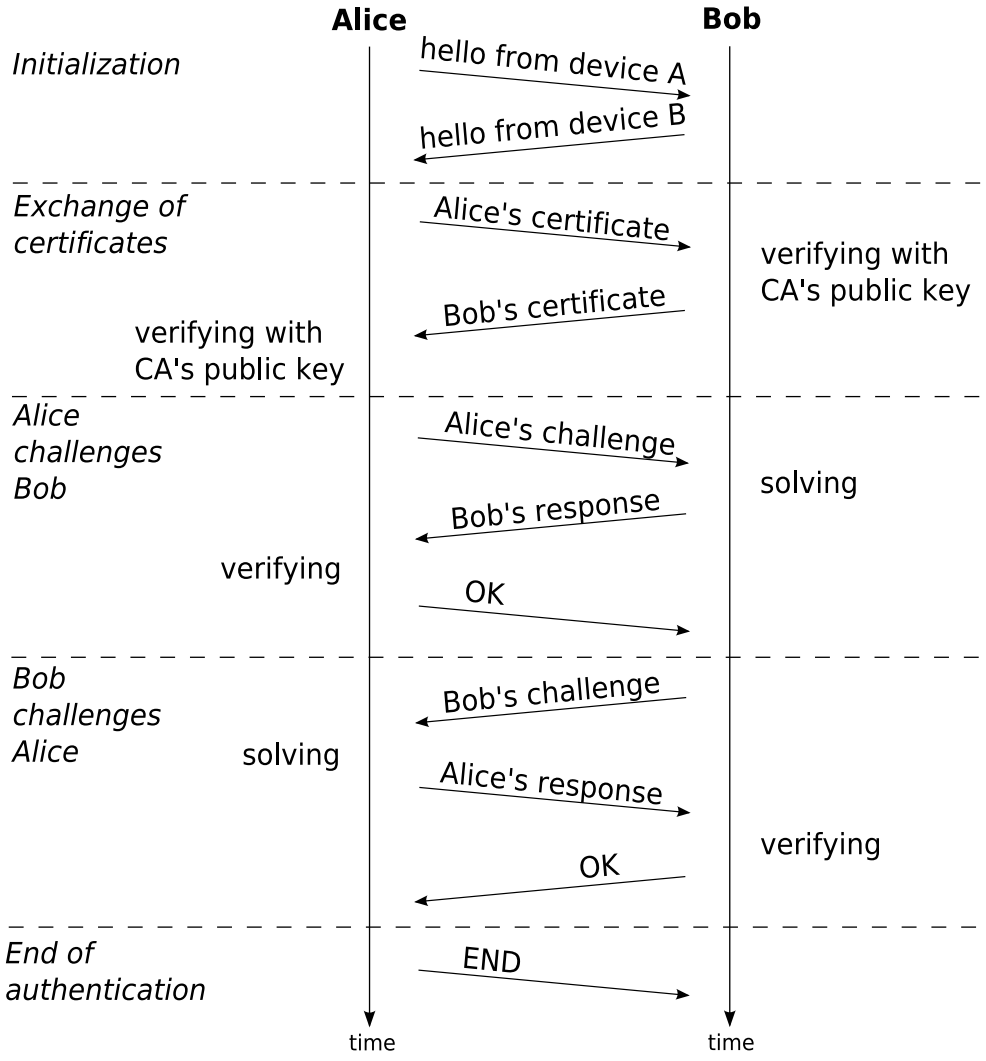

Figure 4.2: Authentication protocol used in the AuthToolkit in case of positive authentication. If certificates or responses have been proved to be wrong, the noticing party sends *BYE* to signalize failure.

Chapter 5

Test series on Java SE and ME

To estimate the expectable speed of offline-authentication on desktop computers and mobile devices, two applications were developed for measuring the execution speed of RSA and ECDSA in various key lengths together with several SHA hash functions.

5.1 Desktop Computers and Java SE

The testing project for desktop computers is called *BCSpeedSE*. When started, all testruns are processed successively until finished. Then a new file will be created in the application's directory, called `BCSpeedSE_[date].txt`, where minimum, maximum, average values and the standard deviation of each part of every test series is stored.
The sequence observed in one test series is always the same:

1. A random message of fixed length is hashed.
2. A key pair is created.
3. The message is signed.
4. The signature is verified.
5. A verification failure is stored.

The verification result is mainly important for RSA due to the certainty of a key pair being created from prime numbers. If the numbers are not prime, verification fails.
Each sequence shown above is repeated one hundred times to straighten out conspicuous slow or fast results. The number of repeatings is not enough for absolutely reliable results, but points out roughly the way Bouncy Castle is able to cope increasing computing

complexity.
The settings observed combined each RSA at 512, 768, 1024 and 2048 bits with SHA-1, SHA-256, SHA-384 and SHA-512, as well as ECDSA at 192, 239 and 256 bits with the SHA variants. The RSA and ECDSA steps represent conventional values from lower to higher security. The different SHAs shall show effects of different message digest bit lengths to the signing and verifying procedures. With RSA, no differences are expected, because even SHA-512 is only as long the block length of the shortest modulus.
The RSA public key value is fixed for each repeating at $2^{16}+1 = 65537$, which is small compared to the modulus lengths (512+ bits) and therefore, operations on the public key are expected to be faster than on the private key. The prime certainty is set to 25, which means that a found number is prime with a certainty of $1-(\frac{1}{2})^{25}$, which is the upper end of values usually implemented.

The tests were run on an *Asus A8N-SLI* mainboard with an *AMD Athlon64 3500+* CPU running at 2200MHz and one gigabyte DDR memory at 400MHz. This represents a mid-range end-user system in mid 2006.

Results are presented in Figures 5.1 through 5.5 with so-called candlesticks. The boxes of each stick show the standard deviation from the average value, which is shown by a horizontal line in the middle of each box. The vertical lines at the top and bottom of the boxes indicate the measured maximum and minimum values.

The RSA key generation results shown in Figure 5.1 are logarithmically scaled. But still the creation times for longer keys cannot be approximated by a straight line, proving that use of RSA makes sense only when key generation is necessary very seldom.
Figure 5.2 describes signing and verification times of RSA. While verification at different key lengths takes about constant time because of the fixed public key at $(2^{16}+1)$, signing becomes slower with a power of more than two, which was expected in Section 2.2.1.
ECDSA key generation measurements are shown in Figure 5.3. The measured times grow about linear compared to key sizes. It should be regarded, that the key length steps are not equally caused through lacking availability of more convenient curves in Bouncy Castle. Of importance is also that already 192 bits of ECDSA are considered about as strong as RSA 1024, as shown in table 2.2.5.
Signing and verifying with ECDSA results are described in Figure 5.4. The signing

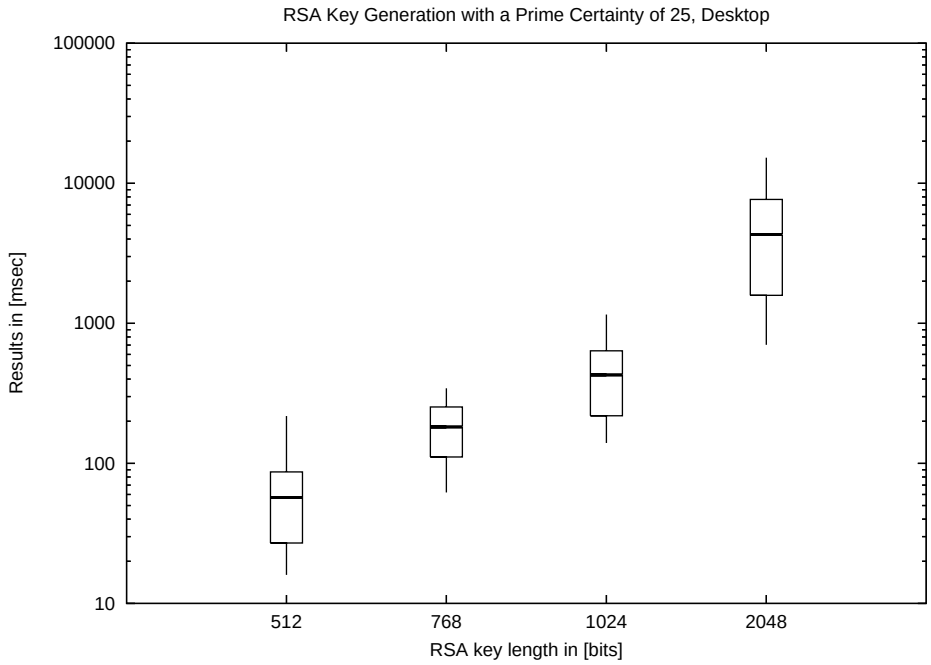

Figure 5.1: RSA key generation on desktop, logarithmic scale

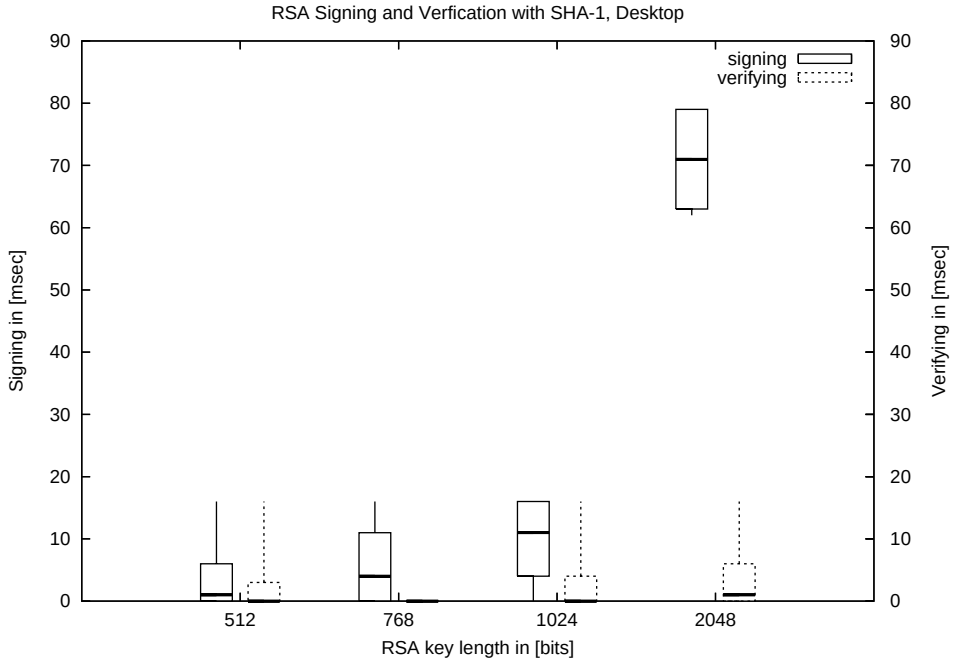

Figure 5.2: RSA with SHA-1 signing and verification on desktop

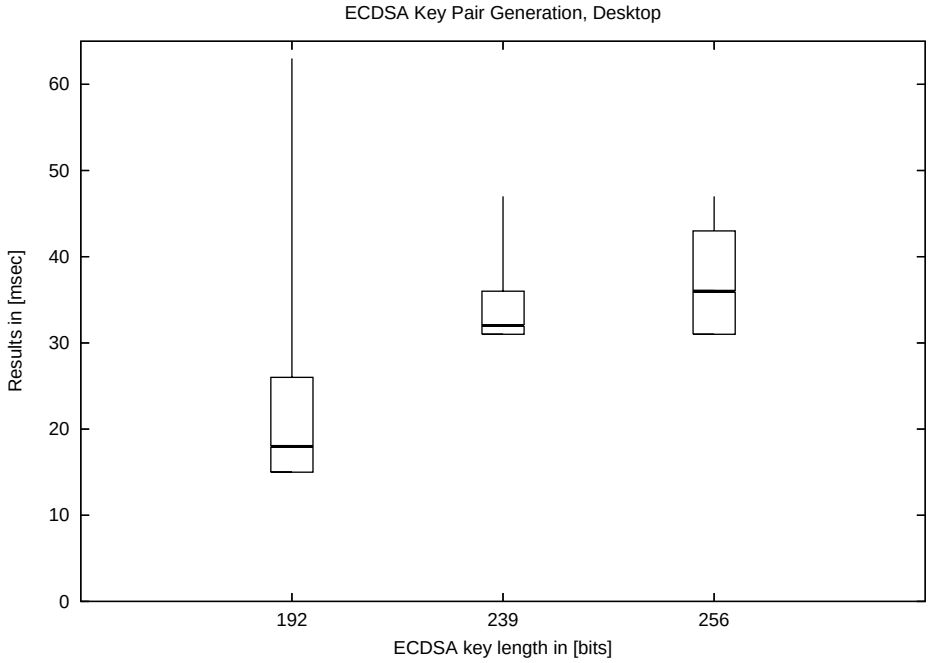

Figure 5.3: ECDSA key generation on desktop

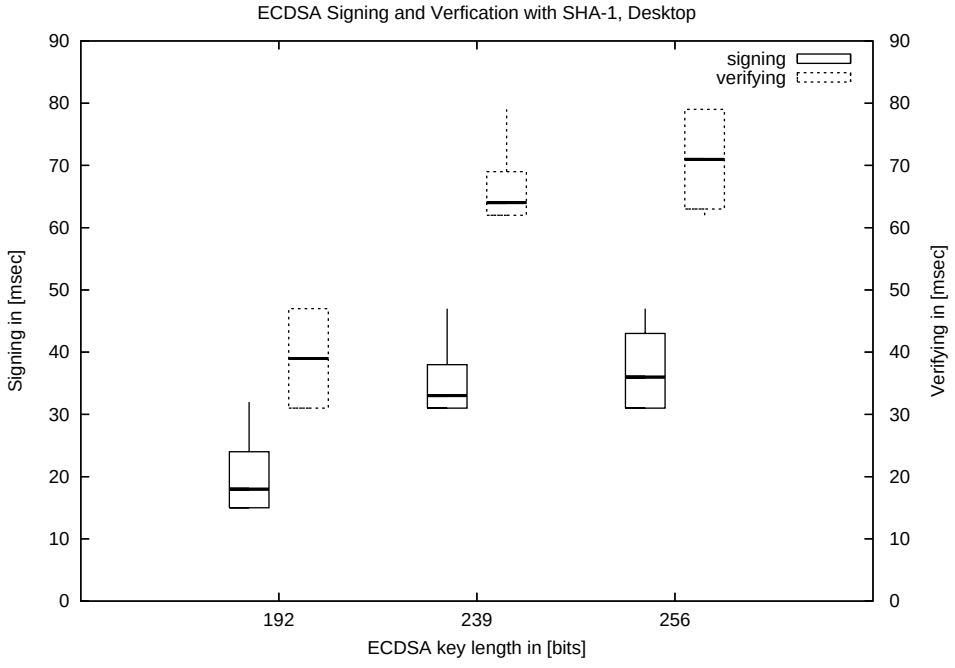

Figure 5.4: ECDSA with SHA-1 signing and verification on desktop

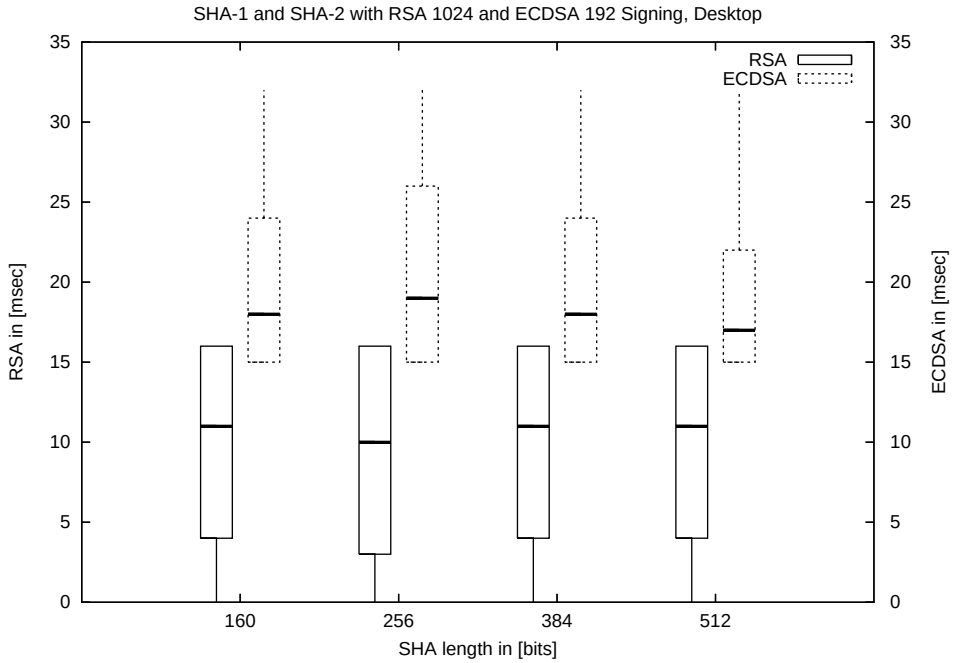

Figure 5.5: RSA 1024 and ECDSA 192 with different SHA algorithms on desktop

times measured take about the same time as key creation, while verification is slower by a factor of two, although it grows linearly also.
In Figure 5.5, measured times of RSA 1024 and ECDSA 192 are shown with different SHA bit lengths. As expected, there appear no differences and the chosen SHA algorithm on desktop computers does not has to be a matter of speed.

The results show that a computer of this class is able to compute even the most complex operations (finding RSA key pairs of 2048 bits length) in about five seconds on average. Noticeable outliers appear at RSA signing with 2048 bits, which indicates behavior when even higher security levels become necessary. Nonetheless, the conclusion is that key lengths on more powerful desktop systems should be chosen from upper security levels, since performance is of no concern there.
The enormous advantage of shorter ECDSA keys compared to RSA is clearly shown at key generation. But also that signing and verifying is slower than RSA even at these short key lengths has to be concerned, since key generation usually is performed much less frequently than signing and verifying.
Which algorithm (and which strength) to choose should be made dependent on the mobile devices observed in Section 5.2.

5.2 Mobile Devices and Java ME

The testing application for mobile devices is called *BCSpeedME*. When started, it demands the user firstly to choose how often each test is ran (1, 20, 50 or 100 times) and then offers a list with possible combinations of algorithms and hash functions. In contrast to *BCSpeedSE*, it is not allowed to run all test combinations in a row without user interaction, since this would take way too long to finish. This eased the problem that the processing units were needed sometimes for other means during testing (like when the phone is called), what adulterated the results.

The mobile devices used for testing are the cellphones *Nokia 6600* released in 2003, *Siemens S65* from 2004 and *Sony-Ericsson M600i* from 2006. All three are modern mobile devices able to run Java ME applications and to communicate via Bluetooth and GPRS, which are integrated in the offline-authentication demonstration application presented in Chapter 6. The *Nokia 6600* implements only CLDC 1.0 and therefore lacks floating point support. But because it is owned by the computer networks chair and was available for testing, *BCSpeedME* was developed to run on this phone, too. The most uncomfortable adaptation was to let the standard deviation be displayed squared, because calculating the square root is not possible with CLDC 1.0. Unfortunately, it hung up during tests regularly, so only a few tests were ran upon it.
The other phone of the chair, the *Siemens S65*, hung up reproducible when trying to compute results when more than one test series were ran in a row. Allowing only one row of tests to run and then showing results solved this problem, so that all tests could also be measured on this phone.
The *Sony-Ericsson M600i*, as being the newest of the devices available for testing, made no adaptations necessary.
The results are presented by candlesticks, which are explained in Section 5.1. All figures show the measured times of the *Sony-Ericsson M600i*, which turned out to be the fastest and most stable device.

A look at the RSA key generation measurements in Figure 5.6 shows the same growth as RSA on the desktop in Section 5.1, although about one hundred times slower. This shows that the higher key lengths should only be considered, if key creation happens very seldom in the application. Keys used for authentication may be valid for years, so even 2048 bits should be taken into account.

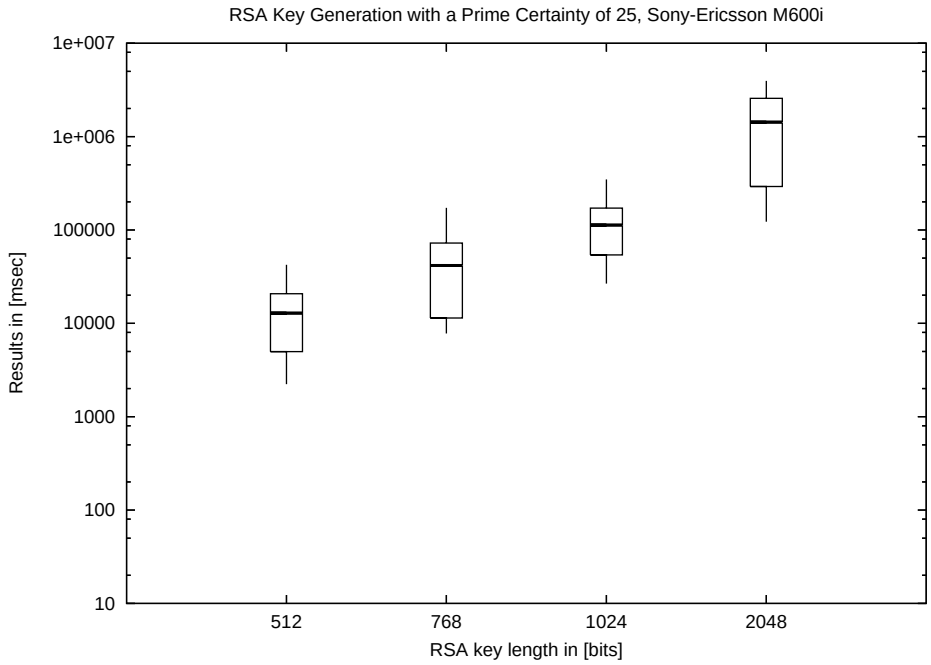

Figure 5.6: RSA key generation on Sony-Ericsson M600i, logarithmic scale

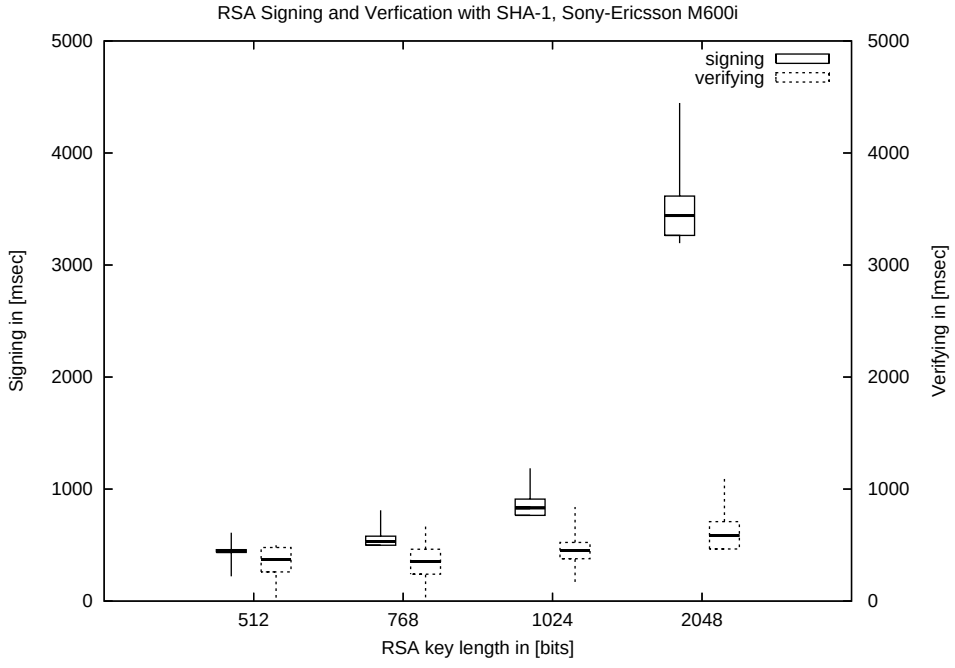

Figure 5.7: RSA with SHA-1 signing and verification on Sony-Ericsson M600i

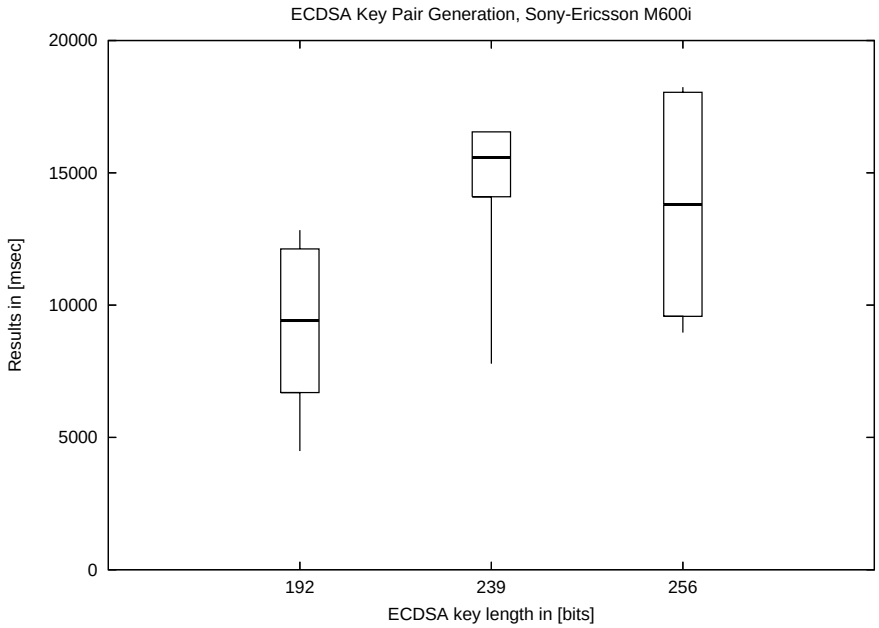

Figure 5.8: ECDSA key generation on Sony-Ericsson M600i

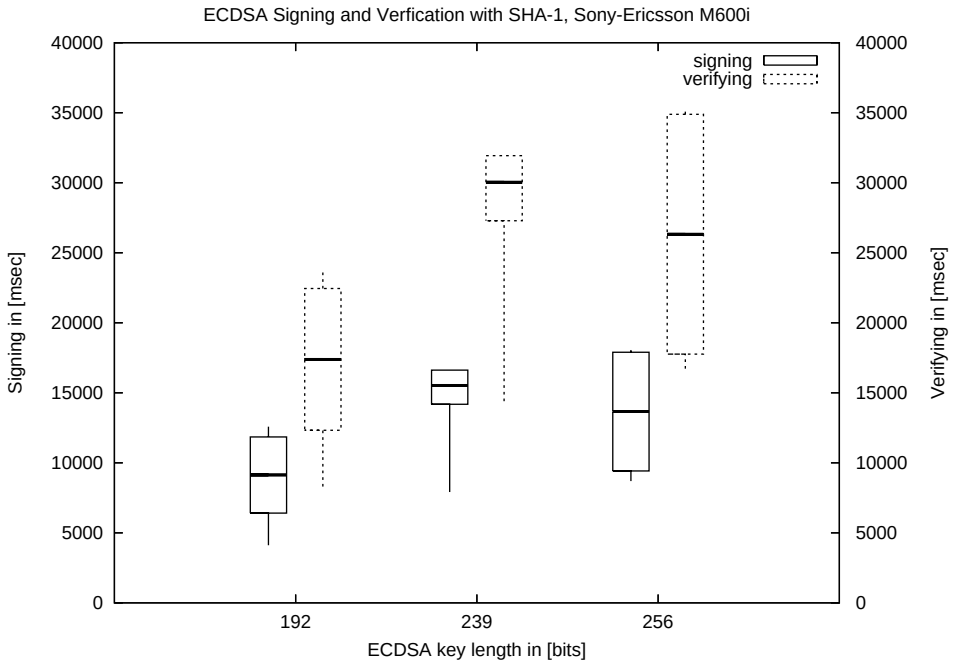

Figure 5.9: ECDSA with SHA-1 signing and verification on Sony-Ericsson M600i

RSA signing and verification, shown in Figure 5.7, is a matter of a few seconds and much faster than with ECDSA (see Figure 5.9), which takes ten to thirty seconds.
In Figure 5.8, ECDSA key pair generation measurements are described. The key pair creation happens in less than 15 seconds and is nearly as fast as ECDSA signing and twice as fast as verifying. The implementation of ECDSA 239 is slower on average compared to ECDSA 256. Therefore, ECDSA 256 should be preferred to ECDSA 239, when a higher security level than ECDSA 192 is necessary.

Compared to the results of the desktop computer in Section 5.1, the mobile results are slower by a factor of one hundred to one thousand times, depending on the specific operation. Especially key generation is very slow on the mobile devices. The other operations need seconds, instead of milliseconds as on the desktop, to finish.
The differences between RSA and ECDSA show that the algorithm to choose finally depends on the emphasis key generation in contrast to verification and signing in an application has. In offline-authentication, key pairs need only be computed once per device, so RSA is a better option as long as ECDSA's speed is not further improved in Bouncy Castle.

Different hash functions, shown in Figure 5.10, do not seem to have an effect on measured RSA 1024 times, so the decision can be made upon desired security level. With ECDSA 192, average values with SHA-384 and SHA-512 raised significantly compared to SHA-1 and SHA-256, which may be caused through splitting longer message digests into more blocks that have to be processed.

Some of the RSA verifications failed, which is supposed to be caused by the way RSA keys are generated in Bouncy Castle. A key is a randomly picked number, which is only estimated to be a prime number. The *certainty*, to which a number should be prime, can be specified when generating keys. If a chosen key pair is made of a non-prime number, signing works but verification fails. All tests above have been made with a certainty value of 25, meaning that a number is prime with a certainty of $(1-(\frac{1}{2})^{25})$. The documentation of the `BigInteger.isProbablePrime(int certainty)` in *Java SE* mentions execution time proportional to the value. If the influence of the certainty to speed in *Bouncy Castle* also behaves like this can also be tested with *BCSpeedME* by choosing the option *RSA Certainty* at start. Then, RSA 512 test series (like above, each series consists of 100 repeatings) with different prime certainties are measured.

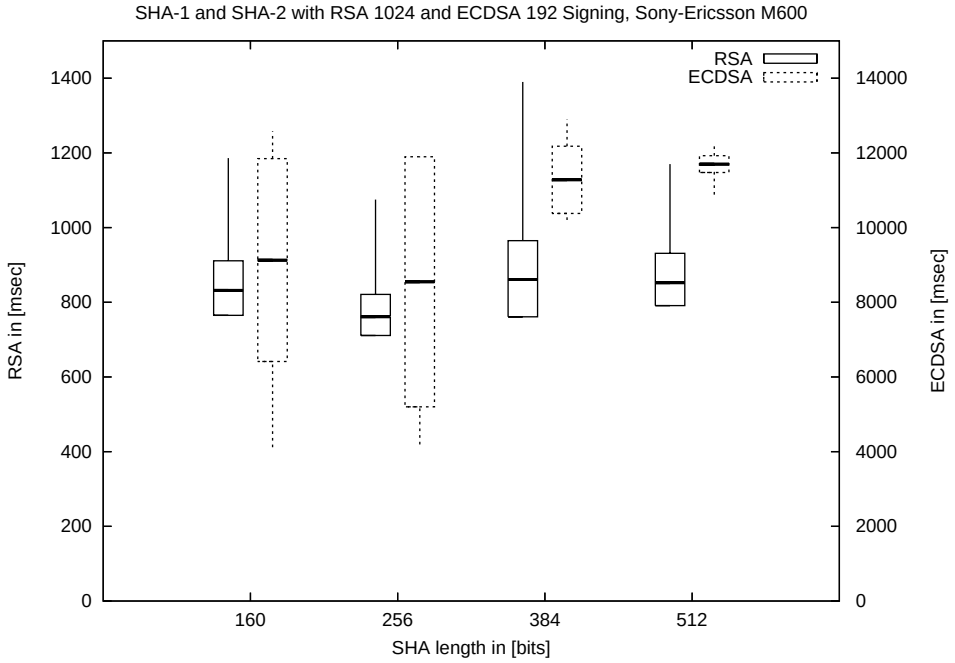

Figure 5.10: Different SHA algorithms with RSA 1024 and ECDSA 192 on Sony-Ericsson M600i

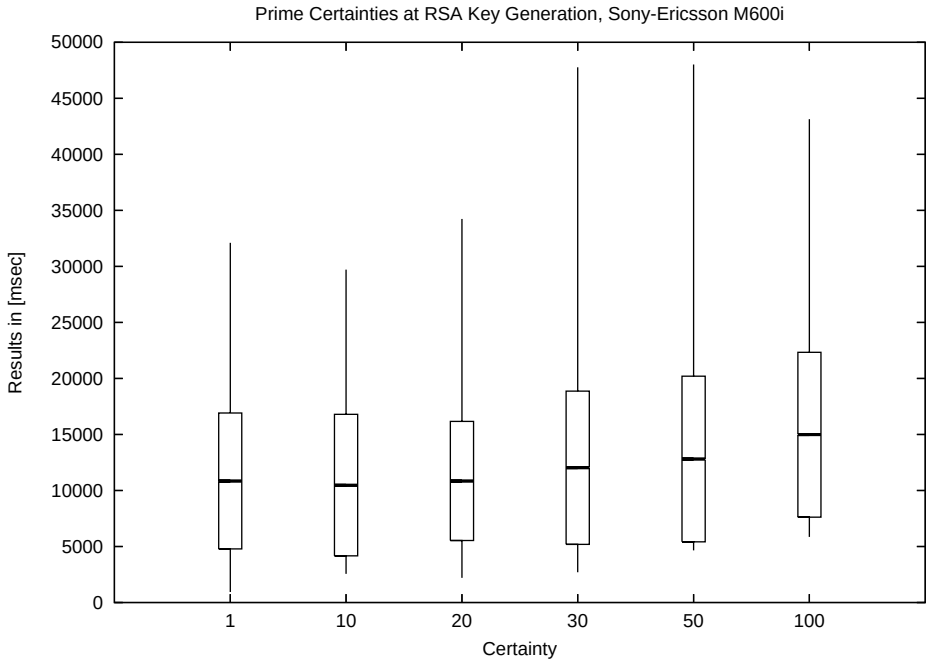

Figure 5.11: Prime Certainties at RSA 512 on Sony-Ericsson M600i

The results in Figure 5.11 show, that the influence of a greater prime certainty is much less than proportional increase at RSA 512. Unfortunately, even one verification with a certainty of 100 failed, where a number is prime with a probability of $(1-(\frac{1}{2})^{100})$. So the reason for failure has to be somewhere else. To cope with that, applications using the Bouncy Castle RSA algorithms should make a "test-run" after key creation: A random number has to be signed and its signature verified afterwards. If the result is good, the key pair is considered okay. Otherwise, another key pair should be created.
With this method, a certainty of about 10 is enough to receive a working key pair in a feasible amount of time.

A final note about the big differences between minimum and maximum values, respectively the width of the standard deviation. Key pair creation is, as already mentioned above, a process of choosing a random number and then testing it's primality. According to that, a prime number can be found fast, even the first number chosen may be prime, but it can also take many attempts.
Signing and verifying processes are better predictable, they happen within one magnitude. These differences are presumably due to the bit representation of the keys and messages, which specifies how many low level multiplications and additions to make when calculating RSA exponentiations or ECDSA additions.
Another factor that appears on mobile devices much stronger than on the desktop is the operating system and other applications running in background, which consume significant amounts of memory and processing time. Although other applications were shutdown when possible before starting *BCSpeedME*, those influences are hardly calculable, although surely significant.

Chapter 6

Offline-Authentication Demonstration Application

Implementing *offline-authentication* requires a mobile client, called *AuthClientMEDemo*, and a certificate management server, the *AuthServerDemo*. The server is aimed at non-mobile computers and therefore will be based on Java SE with the standard Bouncy Castle Crypto API as security provider, while the mobile part will be developed using Java ME and the lightweight Bouncy Castle Crypto API.
These projects show how the *AuthToolkit* can be used for the offline-authentication purpose. The *AuthServerDemo*, acting as certificate authority, manages known user certificates, while the mobile clients register themselves at the server and can execute an offline-authentication process. The server implements the *AuthToolkit* in the same way the clients do, so the Bouncy Castle lightweight API is sufficient to the server.

6.1 The *AuthServerDemo*

The *AuthServerDemo* is a straight forward implementation as a proof of concept for a server managing users and their certificates as PKI. Mobile devices need a root certificate and a personified certificate for offline-authentication, so the *AuthServerDemo* is owner of the root certificate and offers the service of certificate creation to members connecting to it. The certificates are generated with information provided by the clients. When creating a root certificate at first start of the server, the algorithm to use can be chosen: RSA or ECDSA. After initialization, waiting for clients begins to supply them

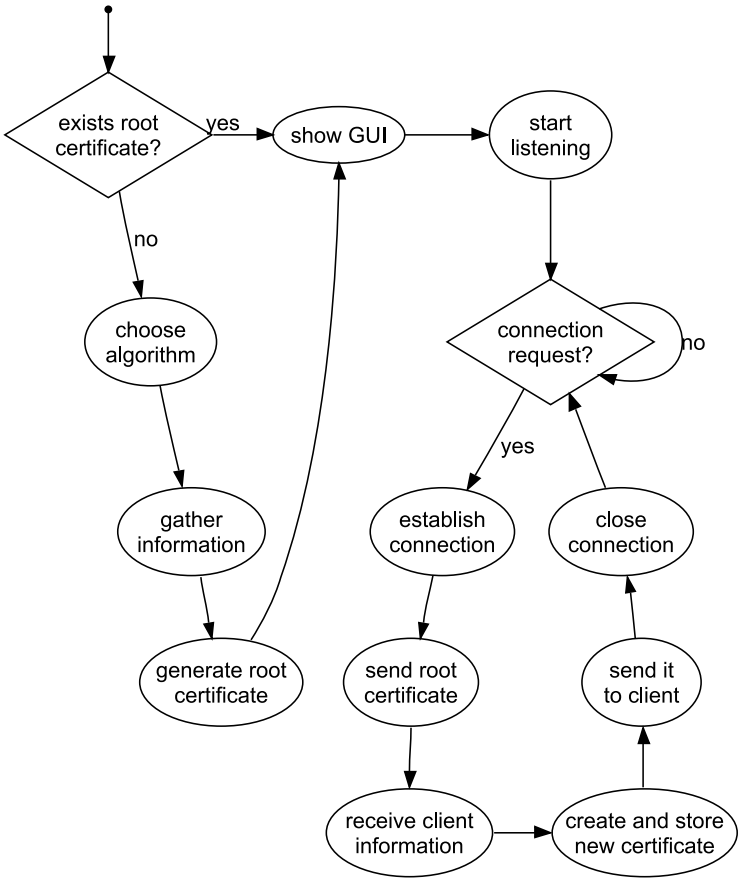

Figure 6.1: Flowchart of the *AuthServerDemo*

with certificates (see Figure 6.1), which have unique serial numbers.
The certificates are stored each in a single file in the working directory of the server. Connecting a database to the server for certificate management is simply implementable but not part of this thesis.
Another possible extension is a logic unit for calculating validity of certificates. This concerns out-dated certificates as well as withdrawn ones. Certificate revocation with offline-authentication has not been analyzed in this thesis and is currently considered unsolved. An approach to handle this may be blacklisting, although this implies lists which can become very large and have to be distributed and kept up-to-date on the mobile devices.
The GUI is realized with Java's Swing API and consists of two tabs, one for showing log messages and another for the certificates stored, which are presented in a table to show their content and signature.

6.2 The *AuthClientMEDemo*

The *AuthClientMEDemo* is the second application of the demonstration designed to run on mobile devices. The implemented functionality comprises finding another device also running an *AuthClientMEDemo* and performing offline-authentication with this device. Before this is possible, the mobile devices need unique certificate, which can be obtained by connecting to the *AuthServerDemo* described in Section 6.1.

Certificate Management

The private key and all certificates are stored in the application's record store, which is managed by Java ME. Other Java applications do not have access to this store for security reasons. But the record store is somewhere on a physical memory, so it is likely possible to access this storage in some way. Therefore, at least the private key should be encrypted with a pass phrase in a serious application to block insights, which is out of scope of this thesis.

User Registration

New users receive their certificate by connecting to the server via TCP/IP over GPRS. This registration is made simple to clarify the proceedings. Neither the user identity is checked as recommended in Section 2.1.5, nor is a secure channel used to exchange certificates. Both is regarded as necessary to a serious implementation.
The procedure is costly to the user and requires online connectivity, but has be done only once for receiving a unique certificate. It simulates a simplified exchange of the CA's and user's information.
Whether RSA or ECDSA is used by the server and clients has to be decided at the first start of the server, see Section 6.1. The client certificates are generated on the mobile devices during server registration, see Figure 6.2.

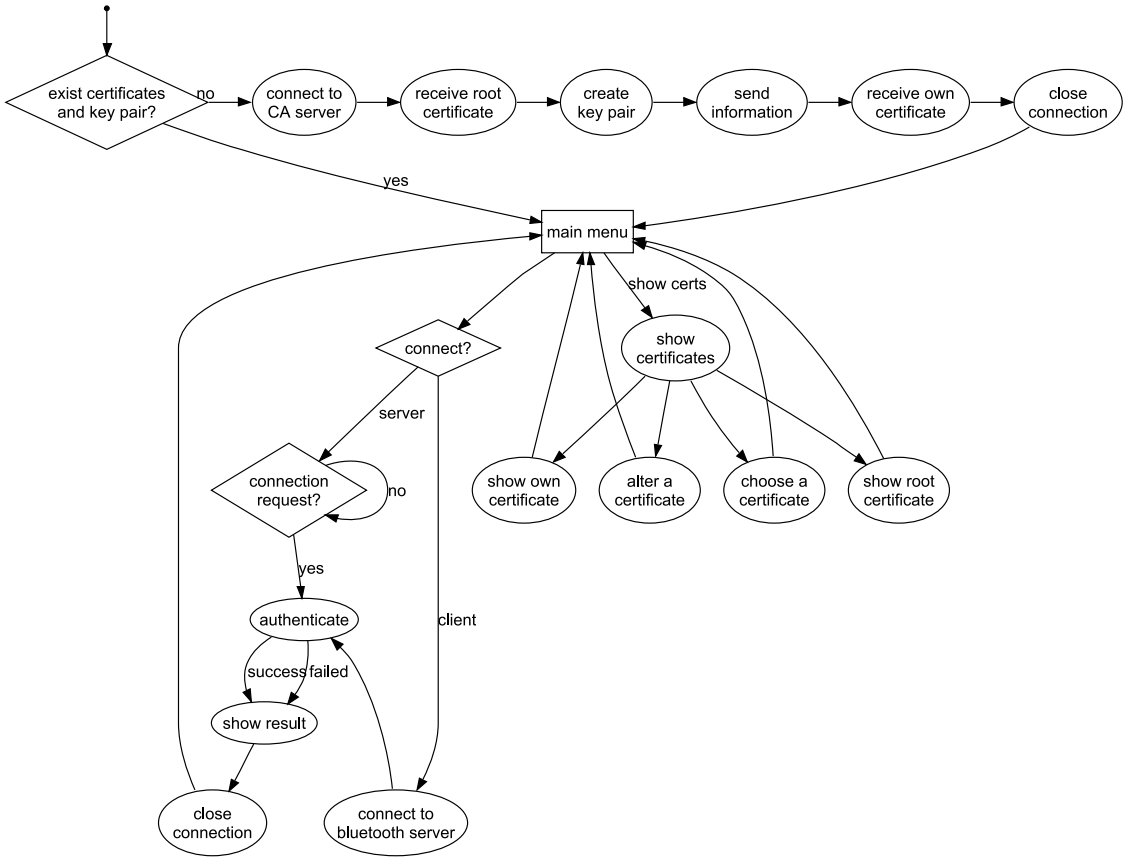

Figure 6.2: Flowchart of the *AuthClientMEDemo*

Figure 6.3: The authentication process in *AuthClientMEDemo*

User Interface

The GUI is a collection of Java ME displays, which indicate the status of the application. A main menu offers the possibility to connect to another device for authentication or to show known certificates. The latter is implemented as a list, where the user may choose to verify the selected certificate, set it as current or alter the common name of the certificate and then set it as current.
Besides these functions, the user's original certificate and the root certificate can be shown.

Offline-Authentication via Bluetooth

When two clients own a certificate, client-client connections are possible, which is implemented over Bluetooth as one possibility of a freely available short range communication. The protocol implemented for offline-authentication is presented in Section 4.4, see also Figure 6.3. The user is informed about the progress of authentication on the device's display.
The decision to use Bluetooth was made because WLAN is not yet supported by most regular cell phones and infrared is often not supported anymore and not suitable to the envisioned scenario of detecting people.
Developing the Bluetooth connection turned out to be the most difficult part of the application. Although the Bluetooth framework of the *DOM* project was still in development, it worked very well on the Java ME emulators from Sun and Sony-Ericsson. So they were chosen for implementation into the *AuthClientMEDemo*. Unfortunately, there appeared problems on real devices regarding Bluetooth inquiry and opening connections. Only two devices of the brand *Sony-Ericsson K750i* could be reproducibly connected, but even there various errors occured. Therefore, the Bluetooth part has been redeveloped on base of the *BluetoothDemo* supplied with the *Sun Wireless Toolkit*, which seems to be the only implementation able to pair real mobile devices. The difference to other projects are mainly a more intense usage of threads.

Cheating Authentication

There are two possibilities implemented in *AuthClientMEDemo* to test the authentication process against deceits: By setting another certificate as current one, the user can change her “identity” for testing the resistance of offline-authentication. The process has to fail, because the private keys to other certificates are unknown.
The other option is to change the common name of a certificate. Here, the certificate itself is altered and so the signature of this certificate will not withstand the verifying process.

Chapter 7

Conclusion and Outlook

The *offline-authentication* scheme presented in Section 2.1.5 enables the *Digital Ownership Management* project to securely verify service members to each other that have never met before without the need to set up a costly server connection. Because of the underlying *Public Key Infrastructure*, the system is scalable to fit even large numbers of users. The implemented *Bouncy Castle Crypto API* supports the special demands on mobile devices and covers the cryptographic algorithms RSA and Elliptic Curve DSA, both proved to be very useful for mobile purposes during theoretical analysis in Chapter 2 and the test series in Chapter 5. The level of security they offer with the recommended parameters will suffice offline-authentication for the next couple of years, although mathematic and technological progress have to be closely observed.

The *AuthToolkit* developed during this thesis serves as foundation to offline-authentication, but may also be used for any other digital signature application suitable to the DOM project.

Speed measurements on cell phones showed that *Elliptic Curve Cryptography* has already passed the border to be used on mobile devices. If the next generation of cell phones offers advancements in calculation speed or the Bouncy Castle implementation gets more optimized, it may become first choice in every application in need of authentication or digital signatures.

The *AuthServerDemo* and *AuthClientMEDemo* are simplified examples for the realization of offline-authentication, but can be easily adjusted to fit the needs of the *DOM* project.

7.1 Outlook

The next step is to include the *AuthToolkit* into the *DOM* project. There, a connection to a database management system will be integrated for handling user accounts server-sided with extended certificate validation logic units.
The client devices will also be enhanced by the *AuthToolkit* to support secure authentication to other service members. For this use, methods for secure access and storage of the private key have to be developed.
If *DOM* applications benefit from data encryption, libraries basing on the Bouncy Castle security provider can be developed easily with the experience gained during this thesis.

Chapter 8

Appendix

8.1 Setting up Bouncy Castle

In this section setting up the Bouncy Castle security provider in a Java SE environment is explained. The installation is not easy there because of missing documentation and requires a couple of steps. With Java ME, usage of Bouncy Castle is much more simple, as described afterwards. The description expects *Microsoft Windows* as operating system, although all steps also apply to Linux, only the installation directories and folder slashes have to be adapted ("\" => "/"). As programming environment, *Sun Netbeans* is recommended and described in the following, because it supports the *Sun Wireless Toolkit*.

1. Assuming that a *Java Runtime Environment* (JRE) and a *Java Development Kit* (JDK) are already set up on the system, *NetBeans* has to be installed. The newest version, *Netbeans 5.0*, can be obtained from [Mic06c]. Since applications for mobile devices are to be developed, the *Mobility Pack 5.0* and the *Sun Java Wireless Toolkit* should also be installed, which are both availably online from Sun, too.
2. Now, Bouncy Castle has to obtained from [otBC06]. Version 1.32 is used for implementation of the applications in the following sections, but at the time of writing, Bouncy Castle 1.33 is already out and should be chosen, because it is mainly a bug fix release without changed functionality. The needed file is called `bc-prov-[version].jar`. It has to be copied to the directory `\lib\ext` of the JRE *and* the JDK's Runtime:
   For example `C:\jre\lib\ext` and `C:\jdk\jre\lib\ext`.

3. Then, a list entry into the file `java.security` has to be made, residing in `\lib\security` in both the JRE and the JDK directories. The "X" has to be replaced with the actual number in the list:
   ```
   security.provider.X= \
   org.bouncycastle.jce.provider.BouncyCastleProvider
   ```
   The position in the list is nearly arbitrary, it may be put anywhere but the first place (Sun's JCE has to stay there). A sequential numbering is important for Java to work with the providers.
4. To undo the key length restriction (see 3.1.3), a "unlimited strength JCE Policy" file from [Mic06b] has to be put into `\lib\security`. Bouncy Castle can now be used with Java SE.
5. For use with Java ME, simply put the file `cldc_classes.zip` containing the lightweight API somewhere *Netbeans* can find it, for example refer to it in a project's classpath as library.
6. An obfuscator, a program for renaming classes for obscuring sense of the source code and removing unreferenced classes is crucial for Bouncy Castle on Java ME for two reasons: Firstly, the size of the package is about 500kb, which is already too much for many mobile devices, and secondly, it tries to define `BigInteger` and `SecureRandom` in the `java.*` namespace, which is prohibited by MIDP devices. *Netbeans* is supplied with the *ProGuard* obfuscator, which has to be configured to maximum obfuscation in the project's settings for Bouncy Castle to work.

8.2 Source Code

All packages discussed in this thesis are supplied with the thesis on a CD, where also the source code's documentation and an electronic version of the thesis is located.

Bibliography

Bos04 Antoon Bosselaer. The RIPEMD-160 page. http://homes.esat.kuleuven.be/ bosselae/ripemd160.html (*as seen Sep 12th 2006*), 2004.

BR95 Mihir Bellare and Phillip Rogaway. Optimal Asymmetric Encryption - How to Encrypt with RSA. http://www-cse.ucsd.edu/users/mihir/papers/oae.pdf (*as seen Sep 12th 2006*), 1995.

BSS99 Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1st edition, 1999.

Buc03 Johannes Buchmann. *Einführung in die Kryptographie*. Springer, Berlin, 3rd edition, 2003.

Cry04 Cryptix. http://www.ntua.gr/cryptix/ (*as seen Sep 12th 2006*), 2004.

DBP96 Hans Dobbertin, Antoon Bosselaer, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. http://homes.esat.kuleuven.be/ cosicart/pdf/AB-9601/AB-9601.pdf, 1996.

Gra05 TU Graz. IAIK JCE. http://jce.iaik.tugraz.at/ (*as seen Sep 12th 2006*), 2005.

Gro02 IETF Network Working Group. http://www.ietf.org/rfc/rfc3280.txt (*as seen Sep 12th 2006*), 2002.

Hoo05 David Hook. *Beginning Cryptography with Java*. Wiley, 1st edition, 2005.

IT05 ITU-T. Recommendation x.509. http://www.itu.int/rec/T-REC-X.509/en (*as seen Sep 12th 2006*), 2005.

LD00 Julio Lopez and Ricardo Dahab. An Overview of Elliptic Curve Cryptography. http://citeseer.ist.psu.edu/333066.html (*as seen Sep 12th 2006*), 2000.

LFB+00 Peter Lipp, Johannes Farmer, Dieter Bratko, Wolfgang Platzer, and Andreas Sterbenz. *Sicherheit und Kryptographie in Java*. Addison-Wesley, 1st edition, 2000.

Lib06 Open Clip Art Library. http://openclipart.org/ (*as seen Sep 12th 2006*), 2006.

LK05 Sing Li and Jonathan Knudsen. *Beginning J2ME*. Springer-Verlag, New York, 3rd edition, 2005.

LWdW05 Arjen Lenstra, Xiaoyun Wang, and Benne de Weger. Colliding X.509 Certificates. http://eprint.iacr.org/2005/067 (*as seen Sep 12th 2006*), 2005.

Mic06a Sun Microsystems. Java ME. http://java.sun.com/javame/ (*as seen Sep 12th 2006*), 2006.

Mic06b Sun Microsystems. JCE Unlimited Strength Jurisdiction Policy. http://java.sun.com/javase/downloads/index.jsp (*as seen Sep 12th 2006*), 2006.

Mic06c Sun Microsystems. Netbeans. http://www.netbeans.org/products/ide/ (*as seen Sep 12th 2006*), 2006.

Mic06d Sun Microsystems. Security and Trust Services API for J2ME (SATSA); JSR 177. http://www.netbeans.org/products/satsa/ (*as seen Sep 12th 2006*), 2006.

NIS99 NIST. Recommended Elliptic Curves for Federal Government use. http://csrc.nist.gov/CryptoToolkit/dss/ecdsa/ (*as seen Sep 12th 2006*), 1999.

NIS00 NIST. FIPS 186-2 Digital Signature Standard (DSS). http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf (*as seen Sep 12th 2006*), 2000.

NIS02 NIST. FIPS 180-2 Secure Hash Standard. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf (*as seen*

Sep 12th 2006), 2002.

NIS06 NIST. Recommendation for Key Management - Part 1: General. http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf (*as seen Sep 12th 2006*), 2006.

otBC06 Legion of the Bouncy Castle. Bouncy Castle Crypto API. http://www.bouncycastle.org/ (*as seen Sep 12th 2006*), 2006.

Pos06 Deutsche Post. Frequently asked questions: POSTIDENT. http://www.deutschepost.de/dpag?lang=de_DE&xmlFile=6394 (*as seen Sep 12th 2006*), 2006.

Rot05 Jörg Rothe. *Complexity Theory and Cryptology*. Springer-Verlag, Heidelberg, 1st edition, 2005.

RSA05 RSA Laboratories. http://www.rsasecurity.com/rsalabs/ (*as seen Sep 12th 2006*), 2005.

RSA06 RSA Security. http://www.rsasecurity.com/ (*as seen Sep 12th 2006*), 2006.

Sch96 Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 2nd edition, 1996.

Sch05 Bruce Schneier. New Cryptanalytic Results Against SHA-1. http://www.schneier.com/blog/archives/2005/08/new_cryptanalyt.html (*as seen Sep 12th 2006*), 2005.

SH04 Tobias Straub and Andreas Heinemann. An Anonymous Bonus Point System For Mobile Commerce Based On WordOfMouth Recommendation. http://www.informatik.tu-darmstadt.de/GK/participants/tstraub/publications/straub_heinemann_acm-sac_2004.pdf (*as seen Sep 12th 2006*), 2004.

Sim05 Kent Simonsen. J2ME Security. Master's thesis, University of Bergen, 2005.

WYY05 Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient Collision Search Attacks on SHA-0. http://www.infosec.sdu.edu.cn/paper/sha0-crypto-author-new.pdf (*as seen Sep 12th 2006*), 2005.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 12.September 2006 Daniel Baselt