DIPLOMAMUNKA

Juhász Sándor Ferenc

Debrecen
2008

Debreceni Egyetem
Informatika Kar

STRONG AUTHENTICATION WITH NFC TECHNOLOGY

Témavezető:
Dr. Fazekas Gábor
Egyetemi docens

Készítette:
Juhász Sándor Ferenc
Programtervező matematikus

Debrecen
2008

Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

I would like to express thanks to all who supported and helped me while I was writing this dissertation. First I am grateful to my supervisor at my home university, Dr. Gábor Fazekas for his support during this time. I would like to thank all my former teachers at the University of Debrecen for the valuable and in-depth knowledge they taught me on various classes during the past years. I would especially like to express my thanks to Dr. Attila Pethő for the interesting and valuable courses on cryptography. These courses helped me a lot during my dissertation project.

I gratefully acknowledge my debt to my teachers and supervisors at Jyväskylä University of Applied Sciences, especially Juha Peltomäki, and Joni Leskinen. Most thanks should go to Petteri Weckström, my project manager who was always very supportive during the meetings and helped to form this project.

Last but not least I should thank my family and friends for their patience while I was writing the dissertation.

1 INTRODUCTION

During my studies at the Jyväskylä University of Applied Sciences I participated in the LASSO project as a researcher and software engineer. The LASSO project is hosted by the Jyväskylä University of Applied Sciences, and managed by project manager Petteri Weckström. My task was to research and develop a strong authentication system using the Nokia 6131 NFC phone. Near Field Technology (NFC) is a contactless communication protocol which enables mobile phones to communicate with wireless smart card readers, and to read Radio Frequency Identification (RFID) tags. The technology will be introduced in 2.3 in greater detail.

1.1 Business idea

Authentication means the process of proving one's identity to an authority. Common examples of authentication are: showing an ID card to a police officer, or entering a secret password when accessing a computer.

In the past decade more and more services became available to all through the Internet. Many of these services process sensitive data: banking transactions are done on-line, confidential messages are sent, etc. When using these services, it is crucial that the users should be able to prove their identity, while the system should limit the possibility of illegal access to the minimum.

A common characteristic of these systems is that an illegal use by a third party can cause considerable damage: money can be lost, sensitive data can be stolen, etc. The most commonly used authentication method, the password, does not provide enough security in these applications: passwords are written down, keylogger applications installed by viruses can log the password while it is typed in, or someone just watches from behind shoulders.

A better solution can be to use multiple tests while authenticating the user. In the well known example of ATM machines, the customer should *have* a bank card, and should *know* a secret PIN. For highly restricted access, biometric information like fingerprint, iris scan is also requested. None of these artifacts are sufficient by themselves, only if all

of them are provided. This type of authentication is called strong multiple-factor authentication.

Carrying one more plastic card in an already thick wallet can be uncomfortable. Can a mobile phone be used to securely access services? This is the topic of this thesis.

1.2 Requirements

The initial requirements of the system were very brief. The main goals of the thesis were the following:

- Designing a system which lets users authenticate to an application running on a desktop PC.
- Strong multi-factor authentication.
- The authentication system should involve the user's Nokia 6131 NFC phone, and a contactless reader connected to the PC.
- The phone and the reader should communicate using NFC technology.
- The architecture of the system should be extensible, to be used through the Internet in various applications.
- Pilot implementation which demonstrates the possibility of implementing a strong authentication system using the technologies mentioned above.

1.3 Schedule of the project

In August 2007, the research was started by reading books and lecture notes on cryptography, computer security and computer science. The main goals were to investigate

- The capabilities of the phone,
- Possible ways of communication with the PC through Near Field Communication
- Different types of authentication
- Vulnerabilities during authentication
- Authentication protocols

- Choosing the architecture of the implementation

The summary of the research can be found in the second chapter of the thesis.

After considering the results of the research, the architecture was decided on, and made a research on the hardware elements which should be acquired to implement the system. The hardware arrived in the beginning of December. Meanwhile the software was already being developed using emulators. In January the software was integrated with the hardware, and the development was finished at the end of January.

As there was not much concise information on using these tools and technologies together, every information had to be found one bit a time. There were many challenges and undocumented features. This was the reason why in the third chapter of the thesis, a tutorial-like description of the technologies and toolkits was written.

At the end of January 2008, the demo environment was successfully finished. Nevertheless, the product is still far from perfect: there are many ideas, and improvements which can make the system more secure, and more useful for everyday use. These improvements are summarized in the fourth chapter.

2 THEORETICAL BACKGROUND

2.1 Authentication

Authentication was the heart of the project, so this field was started with. The most valuable resources in the research were the books by Matt Bishop: Introduction to Computer Security, and the Handbook of Applied Cryptography. Besides them on-line available lecture notes on computer security and cryptography were read. During the research exactness was striven at. When a definition is taken from one of the sources, it is identified by a citation.

2.1.1 Entity Authentication

Definition 1 (Bishop 2002) *Authentication is the binding of an identity to a subject.*

The subject mentioned in the definition can be an external entity trying to prove its identity to a verifier, or a message the identity of which should be confirmed. The first kind of authentication is called *entity authentication*. Messages can be also authenticated during *message authentication*, meaning the data origin is authenticated while the data is sent. As the thesis is concerned only with entity authentication, message authentication will not be discussed here.

Definition 2 (Menzes et al. 1996) Entity authentication *is a process whereby one party (the claimant) is assured (through acquisition of collaborative evidence) of the identity of a second party (the verifier) involved in a protocol, and that the second has actually participated (i.e., is active at, or immediately prior to, the time the evidence was acquired).*

The authentication process involves at least two parties: the *claimant*, and the *verifier*. Sometimes a *trusted third party* can also be involved. Sometimes, according to the conventions used to describe cryptographic protocols, the claimant is called Alice (A), and the verifier is called Bob (B).

During the entity authentication, using some kind of protocol, the claimant provides some kind of information to the verifier, which can be used to authenticate the claimant. This information can come from different artifacts of the claimant:

What you know : a secret that the claimant remembers, like a password or a PIN.

What you have : some information from a physical device the claimant possesses. (magnet card, smart card, phone, etc.)

What you are : data received using some biometric scans like fingerprint or iris scans.

These areas of information are called *authentication factors*. If information from more than one of these areas is used during the authentication process, the term *multi-factor authentication* is used.

After the authentication information has been received from the claimant, the verifier should compare this information to some complementary information stored about the user. This complementary information can be used to verify if the claimant is the one he claims to be. The result of this verification can be the *acceptance* or *rejection* of the claimant.

The objectives of each authentication protocol is summarized in (Menzes et al. 1996) as the following:

1. In the case of honest parties A and B, A is able to successfully authenticate itself to B, i.e., B will complete the protocol having accepted A's identity.
2. (*transferability*) B cannot reuse an identification exchange with A so as to successfully impersonate A to a third party C.
3. (*impersonation*) The probability is negligible that any party C distinct from A, carrying out the protocol and playing the role of A, can cause B to complete and accept A's identity. Here negligible typically means "is so small that it is not of practical significance"; the precise definition depends on the application.
4. The previous points remain true even if: a (polynomially) large number of previous authentications between A and B have been observed; the adversary C has participated in previous protocol executions with either or both A and B; and multiple instances of the protocol, possibly initiated by C, may be run simultaneously.

These are exactly the requirements needed to be fulfilled in this project. As the phone will be used in many applications, the claimant's secret is revealed as little as possible, while the probability of an unauthorized access should be minimalized.

The process of entity authentication can be formalized, with mathematical expressions in an *authentication system*.

Definition 3 (Bishop 2002) *An* authentication system *is a set of five sets:* $Auth = \{A, C, F, L, S\}$

- A *Set of authentication information sent by the claimant*
- C *Complementary information what the verifier stores for validating the authentication information.*
- F *Set of complementary functions that generate the complementary information from the authentication information.* $f \in F, f : A \rightarrow C$
- L *Set of authentication functions which verify identity.* $l \in L, l : A \times C \rightarrow \{true, false\}$
- S *Set of selection functions which enable the entity to create or alter the authentication or complementary information.*

2.1.2 Passwords

In the following paragraphs will be examined what security considerations should be taken when designing an authentication system, and how are these considerations correlate with the most commonly used authentication system: passwords.

Passwords are character strings memorized by users, so they are one of the authentication protocols which use the 'What you know" group of authentication information. During password authentication, Alice authenticates herself by using her username, and proving this by sending her password to Bob. Bob then checks that the password matches to the local copy of Alice's password and if so, grants access. On the verifier's side the passwords are usually not stored as clear text, but they are hashed with a cryptographic hash function like SHA-1 or MD5. During the identification the verifier calculates the hash value of the claimant's password, and compares the hash value to the local copy.

Password authentication systems have a time-invariant property, which means the passwords, and hence the whole authentication protocol is time-invariant: it is the same every time the user authenticates successfully. Once the conversation is intercepted, it can be replayed successfully. Because of this property, the password authentication system is called *weak authentication.*

Although the drawbacks of passwords are well known, password authentication is still the most common authentication method, and still used in many services of the Internet: FTP, Telnet, and in many web applications.

Formally the system can be defined like this:

$Auth = \{A, C, F, L, S\}$

$A = \{(u, p) \mid u \in U, p \in P\}$	U is the set of usernames, P is set of password strings Alice (or anyone else) can send to Bob.
$C = \{pwd(u) = h(p)\}$	Bob stores the hashed values of the passwords for each user.
$F = \{h\}$	The hash function, for example MD5, or SHA-1.
$L = \{L(p)P \rightarrow \{true, false\} : h(p) = pwd(u)\}$	The password is hashed and the hash value is compared to the local copy using the equivalence relation.
S	Not relevant in our case.

There are many problems with this authentication system. See (Bishop 2002) and (Menzes et al. 1996) for details.

- Passwords should be memorized. People choose easy to remember passwords, which can be easily guessed. If a password is hard to remember, it will be written down. Many people give the same password to many accounts. If the password is stolen, the intruder will have access to many services.
- Exhaustive password search. In this case the adversary systematically tries different passwords. If the password space is reasonably big, this "brute force" attack takes very long time to succeed.
- Directory attacks. In a directory attack an adversary gets access to the password file, and tries to defeat the authentication mechanism by trying to determine the password of the user by searching a large number of possibilities.

  Directories of common words and their hashed values are freely available. By accessing the password file, the adversary can search for weak passwords with this dictionary, trying the most common passwords first.

  If the difference of the probabilities of passwords is lowered, the entropy of the password space is increased. If the entropy is maximized, the efficiency of a directory attack can be lowered to the minimum, to the efficiency of the exhaustive password search.

  The entropy of a password space can be lowered by using constraints for passwords like specifying lower limits for password lengths, specifying the mandatory character sets, etc.
- Eavesdropping. Passwords can be intercepted as clear text during typing (simply by watching or with keylogger software), during network delivery, or in the memory during verification. In this case, the full secret of Alice is revealed.
- Replay attack. As passwords are time-invariant, an eavesdropper can impersonate his victim trivially by simply replaying the authentication protocol conversation.

2.1.3 Challenge-response authentication

To defend the authentication system against replay attacks, the time-invariant property of the authentication process should be eliminated. The solution to achieve this is using *challenge-response authentication*, or so called *strong authentication*.

The concept of this authentication protocol is that during every session Bob presents Alice a different challenge that only the true Alice (or someone having Alice's secret) can answer. Alice sends her answer to Bob, who verifies that it is correct. During the process Alice communicates as little information as possible about her secret.

Definition 4 (Bishop 2002) *Let user U desire to authenticate himself to system S. Let U and S have an agreed-on secret function f. A challenge-response authentication system is one in which S sends a random message m (the challenge) to U, and U replies with the transformation $r = f(m)$ (the response). S validates r by computing it separately.*

As the challenge always changes, the active eavesdropper adversary will have no chance to conduct a successful authentication no matter how long he eavesdroppes successful authentication sessions. To break the authentication process he has to find out the secret function f.

Challenge-response authentication protocols can be grouped depending on what cryptosystems they use.

- **Symmetric cryptosystems**. To authenticate, the two parties have to decide on a symmetric key shared by only two of them. Usually a trusted third party, an *authentication server* is used to distribute the symmetric keys between the entities. This architecture is used by the Needham-Schroeder symmetric key authentication protocol, and its implementation, the Kerberos protocol.
- **Public-key cryptosystems**. The claimant demonstrates that she has the private key by one of the following methods:
  - By decrypting a message which was encrypted using the claimant's public key.
  - By digitally signing a message.

By using public-key cryptosystems, the verifier has no direct information of the claimant's secret, so the verifier cannot impersonate the claimant like he could using password authentication. During challenge-response authentication sessions, information can leak out. Although these pieces of information are hard to use, they still manifest some of the secrets of the claimant. A different kind of interactive authentication protocols, called *zero knowledge proofs* eliminates this information leakage. Using

zero-knowledge proofs neither an eavesdropper nor the verifier will receive any more additional information on the claimant's secret than they already knew before the session.

Zero knowledge proofs are an intensively researched and hence very popular field of computer security these days. As these methods are very computation-intensive, and they are cannot be easily implemented in mobile devices, this study will not concentrate on them.

After considering both possibilities for strong authentication, he public-key cryptosystem concept was chosen. The symmetric solutions should need a trusted third party, which would make the piloting more complicated.

2.1.4 Digital signatures

The concept of digital signatures is related to the hand-written signatures used worldwide. A traditional signature binds a person to a document. Digital signatures take an indirect approach and bind documents to *signing keys*. Signing keys are owned by persons, and have to be kept secret. If a signing key is compromised, it can be used to sign documents in the name of the original owner. Later anyone who wants to verify the signature can do that using publicly available verification information for the signature.

Traditionally the digital signature systems use assymetric cryptography, which is defined below:

Definition 5 (Bishop 2002) *An assymetric cryptosystem is a set of* $\{M, D, d, e, D_d, E_e\}$ *where*

M	*Message space, set of message m-s.*
C	*Ciphertext space, set of enciphered message c-s.*
$d \in K$	*The secret, decryption key.*
$e \in K$	*The public, encryption key.*
$E_e : M \to C$	*The encryption function.*
$D_d : C \to M$	*The decryption function.*

Where E_e *has the following property: it is computationally infeasible given a random ciphertext* $c \in C$ *to find a message* $m \in M$ *such that* $E_e(m) = c$*. Eg. the function* E *is a one-way trapdoor function.*

Examples of assymetric cryptosystem are the RSA cryptosystem and the Elliptic curve cryptosystems. These cryptosystems are traditionally used to encrypt messages using the publicly available encryption keys in a way that only the recipient having the private key can decrypt them.

Definition 6 *An assymetric cryptosystem is* commutative *if* $D_d(E_e(m)) = E_e(D_d(m))$.

An example for commutative assymetric cryptosystem is the RSA cryptosystem. This property of the RSA cryptosystem will be explained later.

This means that Alice having her public key can encrypt messages with it. These ciphertexts can only be decrypted using Alice's corresponding public key. No one has Alice's private key, or can create ciphertexts for decryption with Alice's public key. This means if Bob can decrypt a message with Alice's public key, then he can be sure that the message was encrypted by Alice's private key. This is exactly what digital signatures are all about.

Definition 7 *A digital signature system is a set of* $\{M, S, d, e, D_d, E_e\}$ *where*

M	*Message space, set of message m-s.*
S	*Signature space, set of signature s-s.*
$d \in K$	*The signing key*
$e \in K$	*The verification key.*
$S_d : M \to S$	*The signature function.*
$V_e \subseteq M \times S$	*Verification predicate*

Where V_e *has the following property: it is computationally infeasible given a random signature* $s \in S$ *to find a message* $m \in M$ *such that* $V_e(m, c) = true$.

2.1.5 Authentication using digital signatures

The authentication protocol presented in this chapter can be found in (Menzes et al. 1996). A very good description of the protocol, emphasizing the design considerations taken when it was developed can be found in the lecture notes of Prof. M. J. Fischer, held on November 7, 2006 at the Yale University.
(`http://zoo.cs.yale.edu/classes/cs467/2006f/lectures.html`) The following subsection is based on this lecture.

The basic protocol can be summarized like this:

	Alice		Bob
1.,		$\overset{r}{\leftarrow}$	Choose random r
2.,	$s = S_A(r)$	$\overset{s}{\rightarrow}$	Check $V_A(r, s)$

Where (S_A, V_A) is a signature scheme.

This authentication scheme is not secure as it exposes Alice's signature system to a chosen *plaintext attack*, eg. Bob can get Alice to sign any message.

Chosen plaintext attacks are very powerful and hard to defend against, everything that can be done against it is good. The main idea is that no party (neither Alice nor Bob) should have full control over the message being signed.

A more complex, but not more secure version of the protocol is the following:

	Alice		Bob
1.,	Choose random r_1	$\xrightarrow{r_1}$	
2.,		$\xleftarrow{r_2}$	Choose random r_2
3.,	$r = r_1 \oplus r_2$		$r = r_1 \oplus r_2$
4.,	$s = S_A(r)$	$\xrightarrow{s}$	Check $V_A(r, s)$

Where $\oplus$ is some kind of "addition" operation.

This protocol is not better than the previous as Bob still can have full control over the value of r: given a value of r_a, he just calculates $r_2 = r \oplus (-r_1)$.

If the order of the first two steps is changed, then Alice (or anyone pretending to be Alice) will have full control over the value of r for the same reasons. An active eavesdropper, after recording a successful authentication session, will be able to successfully authenticate to Bob.

To weaken the ability of Bob to initiate a chosen plaintext attack, and also to weaken the ability of an active eavesdropper (Malory) to succeed in an attack, let use $r = r_1 \cdot r_2$ concatenation instead of the addition. The attack can even be made harder if some cryptographic hash function is used.

The latest variant of the protocol:

	Alice		Bob
1.,	Choose random r_1	$\xrightarrow{r_1}$	
2.,		$\xleftarrow{r_2}$	Choose random r_2
3.,	$r = h(r_1 \cdot r_2)$		$r = h(r_1 \cdot r_2)$
4.,	$s = S_A(r)$	$\xrightarrow{s}$	Check $V_A(r, s)$

2.1.6 RSA

One assymetric cryptosystem which can be used to implement the authentication protocol introduced in the previous subsection is the RSA cryptosystem, published in

1977 by Rivest, Shamir and Adelman. This cryptosystem is widely used to encrypt, sign messages and authenticate users.

The algorithm

1. Choose p, q big prime numbers.
2. Let $n = pq$
3. Choose e where $e < n, gcn(e, \phi(n)) = 1$
   $gcn(n)$: greatest common divider
   $\phi(n)$ The *Euler phi-function*. Informally, it gives the number of the positive integers not greater than n that are relatively prime to n, where 1 is counted as being relatively prime to all numbers.
4. Choose d where $ed \equiv 1(mod\ \phi(n))$
5. The public key: (n, e)
   The private key: (d).
   The public operation: $RSAPUBLIC(m) = m^e\ mod\ n$ where $m < n$
   The private operation: $RSAPRIVATE(m) = m^d\ mod\ n$

By using basic number theory and group theory, the correctness of the RSA algorithm can be proved. This states that

$m = RSAPRIVATE(RSAPUBLIC(m))$
$m = RSAPUBLIC(RSAPRIVATE(m))$

To be brief, (1) a message encoded with the RSA public key can be decoded by the corresponding RSA private key. (2) A message signed by an RSA private key, can be verified by the corresponding RSA public key. By this property the RSA cryptosystem is considered a *commutative cryptosystem*, which allows to use it both in data encryption and digital signature schemes.

The RSA algorithm is based on the problem of factoring large numbers, and on the RSA problem. These problems are currently believed to be computationally *hard*. The hardness of factoring numbers is a classical example in *complexity theory*. See (Menzes et al. 1996) for more details. The second cornerstone of the RSA algorithm, the *RSA problem* states that decrypting an RSA-encrypted message while the private key is unknown, is *computationally hard*. Using mathematical notation:

Definition 8 RSA problem *For given c, e, n the following equation is hard to solve: $c = m^e\ mod\ n$ where $m =?$.*

To solve this equation, e should be factored, hence the complexity of this problem is reduced to the first one, which is known to be hard.

The RSA key generation needs deep knowledge of number theory. The prime numbers are usually generated using probabilistic prime tests. For the d coefficients, a standard value, one of 3, 17, or 65537 is used. This is beneficial as using one of these numbers makes the public key operation very efficient. The efficiency comes from the properties of the modular exponential calculating algorithm. The corresponding e can be calculated using the *Extended Euclidean algorithm*. The value of $\phi(n)$ can be easily calculated as $\phi(n) = (p-1)(q-1)$ if p and q are prime numbers.

As the private exponent contains much more '1' value bits than the public exponent d, calculating the modular exponentials takes much more time. Hence, private operations are much more time consuming. To make the private operation more efficient, a modified version of the RSA algorithm is published which uses the *Chinese remainder theorem*. The private operation of this implementation is four times more efficient than the original.

Detailed, mathematically correct definition of the mathematical background of the RSA can be found in (Menzes et al. 1996).

RSA as a digital signature scheme

When using RSA as a digital signature, Alice generates a keypair of (e, n) and (d) where the first is the public and the second is the private key.

Then can introduced:

$S_d(m) = RSAPRIVATE(m)$
$V_e(m, s) = true$ iff $m = RSAPUBLIC(s)$

2.1.7 Distribution of public keys

The protocol introduced earlier has the precondition that Bob has the verification function (eg. the public key) of Alice. As defined in (Menzes et al. 1996) This relationship, in which both entities share common data to facilitate cryptographic techniques is called *keying relationship*

Establishing keying relationships is not a trivial procedure. Besides other things, the initialization, generation, and distribution of keying material is the responsibility of the *key management*.

Definition 9 (Bishop 2002) Key management *is a set of techniques and procedures*

supporting the establishment and maintenance of keying relationships between authorized parties.

Key management encompasses techniques and procedures supporting:

1. *initialization of system users within a domain;*
2. *generation, distribution, and installation of keying material;*
3. *controlling the use of keying material;*
4. *update, revocation, and destruction of keying material; and*
5. *storage, backup/recovery, and archival of keying material.*

These tasks are described in great detail in (Bishop 2002) and (Menzes et al. 1996). In this section the key distribution aspect of public keys is the focus.

As active eavesdroppers can be active in the network, public keys cannot be sent in clear text from Alice to Bob, as an adversary (Mallory) can step between Alice and Bob and he can successfully accomplish a *man in the middle* attack. To Alice Mallory will act like being Bob, and to Bob, he will act like being Alice. All the communication will flow through him, and he can read and modify the data sent between Alice and Bob.

To defend against attacks like this, many different key distribution strategies were developed. (Menzes et al. 1996)

- *Point-to-point delivery over a trusted channel.* Alice gives Bob her public key in a channel on which there is no active eavesdropping. This can involve copying public keys from one computer to another using USB sticks, or sending the keys printed on paper as registered mail. This solution is uncomfortable especially if the public keys change frequently or the number of the associated entities is high.
- *Direct access to a public file.* A secure, trusted public file or database of the public keys is maintained by an authority. Users read this file to attain the public keys. They have to access the file using a secure channel, as using clear text, this architecture is sensitive for active eavesdropping.
- *Off-line delivery service using certificates* A *certificate* is a document containing a data part (a public key, the name of its owner and other supporting information) and a digital signature of the data part signed by a *Certificate Authority* (CA). Certificate authorities are entities performing public key management. By creating the certificate and signing it, the certificate authority affirms that the public key is owned by the entity named in the certificate. Users of the system have the authentic public key of the CA, hence they can verify the validity of any certificate published by this CA.

Certificates are usually provided by the CA's on-line directory service, although certificates can be stored in any, even untrusted direct repositories, and communicating parties can also exchange them freely over untrusted networks. An active eavesdropper cannot alter the signed certificate without breaking the digital signature. *certificates*

2.2 Smart cards

By using one of the key distribution mechanisms, Bob can easily get the public verification key needed to authenticate Alice. Alice also has to store her private signing key somewhere, in a safe place where it is always available when needed. Having the signing key, maybe password protected in the filesystem of her laptop, or on a USB stick, by all means not the most secure method. This is where the *smart cards* come into the picture.

Smart cards are small, usually 85 mm x 54 mm plastic cards with embedded integrated circuits which can process information. They are widely used in telephone, transportation, banking, health care systems, and as electronic purse.

Smart cards communicate with the outside world through a serial interface. This interface can be a contact interface specified by the ISO 7816 standard, or a contactless interface which uses radio-frequency to communicate with the outside world. (Specified in the ISO 14443A/B standards)

Using the serial interface, the cards can be connected to *Card acceptance devices* (CAD-s, card terminals). When connected the card behaves like a state machine. APDU commands can be sent to the card through the CAD, and after executing the requested command the card replies with a response containing a status word and optional data.

Depending on the functionality which can be achieved using APDU commands in a smart card, the cards can be divided into two groups:

Memory cards which are used for information storage only. A common example for memory cards is the telephone card used in public phones. These cards provide security logic for accessing data.

Intelligent smart cards which contain a microprocessor and accompanying memory. They are capable of reading writing data and having calculations on them. Applications developed by third parties can be uploaded to the cards. The most important aspect of intelligent smart cards, regarding the thesis, is that some smart cards contain cryptographic coprocessors for AES, DES, RSA and EC cryptosystems.

Table 1: CLA values defined in ISO 7816

0x00-0x1F:	Card instructions like read file, etc. The last digit is defined by the standard.
0x20-0x7F:	Reserved
0x80-0x9F:	Application specific, last digit is interpreted as the standard.
0xA0-0xAF:	Card vendor specific
0xB0-0xCF:	Application specific
0xD0-0xFE:	Card specific
0xFF:	Protocol type selection

Table 2: Status words defined in ISO 7816

0x61xx, 0x9000:	Normal processing
0x62xx, 0x63xx:	Warnings
0x64xx, 0x65xx:	Execution errors
0x67xx-0x6Fxx:	Checking errors

Smart cards are *tamper-resistant* devices. This means any tampering of the card results in the destruction of the data in it. By this property, cards are ideal devices for security-related use-cases like storing private keys. In the case of authentication, private signing keys can be generated on it which can never leave the card.

Communication with smart cards

The communication unit of the protocol is the APDU. An APDU can contain a command directed to the card, or a response from the card to the CAD.

The first byte of the command is a class byte (CLA). In card applications usually 80 is used as CLA. The second digit is the instruction byte which is followed by two, one-byte long parameters called P1 and P2. A command APDU can contain optional data, which should be preceded by the length of the data itself. The last byte of the command is the maximum length of the data the response can provide with the maximum value of 0xF9.

The CLA values are specified by the ISO 7816, and shown in table 1.

A *response APDU* contains two parts: the data returned by the execution of the command APDU, and a two-byte-long response code consisting of the values of SW1 and SW2. The response code domains can be seen in table 2

On intelligent smart cards the card applications are not referenced by names but by AID-s. An AID is a 5-16 bytes long number as defined in ISO 7816-5. All AID-s contain

two parts. The first part specifies the application vendor. The vendor AID-s are unique for every application vendor and assigned by the ISO. The remaining part is application specific and assigned by the vendor itself. In the Sun Java Card SDK, the vendor specifier is the 8 byte value of 0xa00000006203010c, which is followed by a one-byte package, then a one-byte applet identifier. As an example, the Hello World card application included in the SDK has the AID of 0xa00000006203010c0101 (package no. 1, application no. 1)

Smart card applications

Smart card applications are not standalone but distributed systems. A smart card application usually has three parts:

- *Card side*: the smart card running some proprietary operating system, and some on-card applications.
- *Reader side*: a PC, a terminal or a mobile phone equipped with a card acceptance device. Its role is to orchestrate the communication between the card side, the back-end and the end user. It can perform business logic depending on the application.
- *Back-end*: a server side service which communicates with the reader side. It can provide authentication, accounting functionality or something else depending on the business case.

The architecture is displayed in figure 1.

2.3 Contactless smart cards and NFC

Regarding the topic of the thesis, an important type of smart cards are the *contactless smart cards*. These cards connect and communicate using 13.56 MHz radio frequency waves with the card acceptance devices using the ISO 14443 *contactless proximity card* standard. The technology utilizes the electromagnetic induction. Every card contains a magnetic loop antenna, in which electricity is induced if it is placed near the magnetic field generated by the card acceptance device. This energy is enough to power the microprocessor in the card. The cards usually should be less than 10 cm from the CAD to communicate. Data transfer rates between the two devices are high, ranging from 106 kbit/s to 848 kbit/s.

A widely used, similar technology which uses the same, ISO 14443 standard for communication, is the *Radio Frequency Identification* (RFID). RFID uses inexpensive

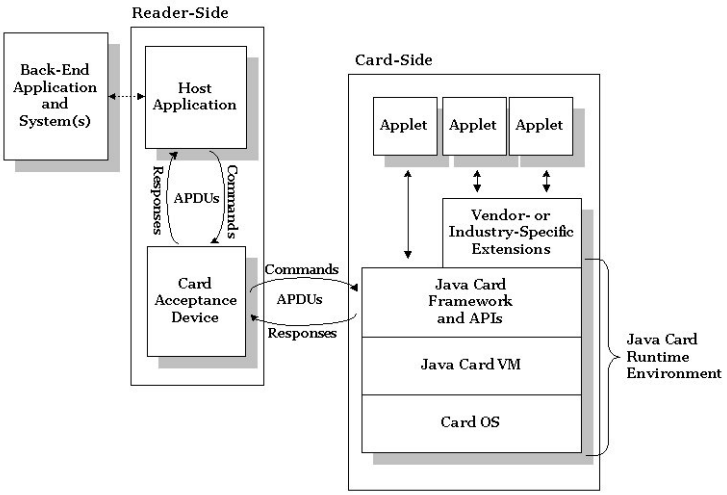

Figure 1: Architecture of a Java Card Application. The image is copied from (Oritz 2003a)

micro controllers equipped with antenna sticked to paper or embedded into small boxes to store identification information. In contrast with smart cards, the RFID micro controllers have no processing capability and have very limited amount of memory. (Typical memory capacities: 48 byte, 720 byte, 4 KByte) The same contactless card acceptance device can be used to read the values of the tags for communicating with contactless smart cards. RFID tags are widely used in warehouses, libraries and other logistics services, E-Passports, animal identification, and in various other areas. RFID is supposed to replace barcodes in many areas.

Near Field Communication (NFC), a simple extension of the ISO 14443 which can be used for short distance high frequency communication. NFC combines the smart card interface and the card reader into one device, targeting to be embedded into mobile phones. NFC devices are compatible with the existing ISO 14443 infrastructure, and can be used to read smart cards, RFID tags, or to communicate with other CADs, simulating a smart card. Two NFC devices can also communicate with each other. The NFC standard and related data format standards are maintained by the NFC Forum (`http://www.nfc-forum.org`), a non-profit organization with the members of NXP Semiconductors, Sony and Nokia. In the near future many mobile phones utilizing NFC technology are expected to come to the market. Currently the only one is the Nokia 6131 NFC.

As the ISO 14443 devices use public unprotected channels while communicating with each other, the technology is vulnerable to various attack types. The devices use radio frequency communication, the interactions between them are very vulnerable to *eavesdropping*. As the communication channel is public and unprotected, various *man in the middle* attacks can be also performed if the communicating parties do not use some kind of encryption and mutual authentication. These aspects should be considered when developing sensitive NFC applications.

2.4 The Nokia 6131 NFC phone

The only hardware which was mandatory to use in the project was the Nokia 6131 NFC mobile phone. The main information sources about the phone were the product home page (`http://europe.nokia.com/A4307094`), the Wiki and public forum of the Forum Nokia website (`http://www.forum.nokia.com/`), and the documentation found in the Nokia 6131 NFC SDK. (The SDK can be downloaded from the Forum Nokia website.)

Main features of the phone:

- Clamshell design, QVGA display.
- Implements the Series 40 platform, 3rd edition. Java ME, MIDP 2.0 CLDC 1.1 support.
- Built in support for NFC technology.
  - Integrated card reader which can communicate with (reading and writing) Mifare UltraLight, 1k, 4k RFID tags, and contactless smart cards.
  - The NFC device can emulate a Mifare 1k tag.

Figure 2: The Nokia 6131 NFC mobile phone

- Integrated JavaCard 2.2.1, GlobalPlatform 2.1.1 compatible secure element with 72 KByte EEPROM. With it the phone can be used as a contactless smart card.
- JSR-257 API for 3rd party Java-based NFC applications. Java applications can communicate with RFID tags and smart cards, and use peer-to-peer communication protocols to exchange data between Nokia 6131 NFC devices. Signed Java MIDlets can access the phone secure element as a smart card.

3 JAVA CARD TECHNOLOGY

After researching the the capabilities of the NFC phone, it was found that the phone can only communicate thrugh the secure element with the card acceptance device. Hence, the next topic to learn was the Java Card Technology and the usage of the Sun Java Card Software Development Kit. There was no concise book, the needed information had to be found bit by bit from different sources. Most of this chapter was written using the on-line sources from the Sun's webpage (Oritz 2003a; Oritz 2003b; Oritz 2003c; Ort 2001a; Ort 2001b), articles from JavaWorld (Chen, Giorgo 1998) and the documentation included with the Nokia 6131 NFC SDK, and the Java Card SDK.

In this chapter the process of developing java card applications is introduced, and the usage of the Java Card Software Development Kit v2.2.1. A demonstration application is developed, which implements a counter in a contactless smart card. When the card is touched on the card acceptance device, the counter is incremented, and displayed on the screen of the CAD.

The JavaCard technology (`http://java.sun.com/products/javacard/`) was developed by Sun Microsystems in 1997. By the definition of Sun, from the product homepage, "Java Card technology provides a secure environment for applications that run on smart cards and other devices with very limited memory and processing capabilities. Multiple applications can be deployed on a single card, and new ones can be added to it even after it has been issued to the end user. Applications written in the Java programming language can be executed securely on cards from different vendors."

All the big smart card manufacturers like Gemalto (Former Gemplus and Axalto), NXP etc. License Java Card Technology, and many smart cards (like mobile phone SIM cards, bank cards) already on the market supports this technology. The main goal of Java Card technology is to provide a standardized software development and deployment environment for various smart card manufacturers and for third party software developers. The Java Card technology is based on the well known Java technology, and can be treated as a precise subset of the bigger platform. From the developers' side, all smart card applications written using the open standard of Java Card Technology can be run on the Java Card Virtual Machine embedded into all the compliant smart cards no matter which manufacturer the card is from. As different cards come with different memory capacity, computation power, communication protocol and API support, smaller incompatibilities can happen.

3.1 Java Card Specification

The Java Card specification published by Sun (see the documents directory in the JCDK for details) has three parts:

- The Java Card Virtual Machine (JCVM) specification. Contains the Java Card language definition, the virtual machine specification, instruction set, and the various file and formats for data types and the Java Card bytecode.
- The Java Card Runtime Environment (JCRE) specification: describes further runtime behavior like memory management and security.
- The application programming interface. Provides access to the JCRE and other native card services from the on-card applications.

The Java Card Virtual Machine

All Java Card compatible smart cards run a very specific, simplified virtual machine on the card operating system. The Java Card virtual machine runs *applets* compiled into Java Card bytecode. The applets use different, optimized bytecode encoding compared to the Java SE, because of memory limitation considerations. In one JCVM multiple applets can be executed simultaneously from different vendors.

Because of the limitations of the Java Card platform, some Java language features are not supported in Java Cards. The Java Card language is a precise subset of the Java language. The applets are still object-oriented programs, utilizing classes, interfaces, etc. Compared to the standard Java language, the main differences are the following:

Not supported features .

- Dynamic class loading
- Security manager
- Threads
- Object cloning
- Certain aspects of package-level visibility

Not supported keywords : `native`, `synchronized`, `transient`, `volatile`

Not supported data types : char, double, float, long. The support of the int type is optional.

There are limitations on the number of implementable interfaces, number of methods in a class, number of classes in a package. A more detailed list can be found in (Oritz 2003a).

The Java Card virtual machine is masked into the read-only memory of the card during manufacturing, and started only once. Its lifetime is equivalent with the lifetime of the card. The virtual machine is never stopped. The objects instantiated by the JCVM are stored in the non-volatile memory, so when the card is unpowered, all the objects will stay alive in the card. The JCVM can use garbage collector to clean out abandoned instances.

Java Card Runtime Environment

The Java Card Runtime Environment specification describes the additional services the JCVM should provide to ensure security and to optimize memory consumption.

The Java Card Runtime Environment ensures that multiple on-card applications can run simultaneously in a secure environment. Every applet has access only to its own data, and applets are separated using firewalls. Objects can be shared between applets if this is requested explicitly.

Exception handling is also handled through JCRE-owned exception instances to lower the memory consumption of the applets.

Application programming interface

The Java Card specification provides very limited runtime library API for accessing the JCRE and other services. The Java Core API is not supported. Most methods of the Object and Throwable classes are not available.

The main packages specified in the runtime library:

javacard.framework Provides the applet superclass, APDU I/O, exception handling, PIN support classes.

javacard.security Security related classes and interfaces for cryptographically secure random number generation, key generation, message digest, and signatures.

javacardx.crypto Cryptographic encryption and decryption support.

As of the Java Card 2.2.1, the cryptographic library, the supported cryptographic functions:

- Symmetric cryptosystems: AES with 16, 24, 32 bit keys; DES, 2 and 3 key TDES
- asymmetric cryptosystems: DSA with 512, 768, 1024 bits, Elliptic Curve cryptography, and RSA with 512-2048 bits.
- Diffie-Hellman key exchange
- Message digents: MD5, RIPEMD160, SHA-1

Java Card specification implementation

The Java Card specification is implemented in two separate software packages. The on-card virtual machine, and the Java Card Converter tool.

The first part is the *on-card java virtual machine* and runtime environment, usually embedded into a smart card. This interprets the byte code downloaded to the card. It instantiates objects, and executes the methods of the objects according to the Java Card specification.

The second part of the specification is implemented in the Java Card Converter tool, which is part of the Java Card Development Kit. This loads the .class files generated by a standard Java compiler, verifies that they conform the Java Card language restrictions, then generates a CAP archive of Java Card bytecode compatible class files, ready to install and execute on a JCVM. The converter tool is necessary as the cards have no processing power to ensure the restrictions mentioned above, and they have to rely on an external verifier. (This conversion is similar to the preprocessing step of Java ME applications running on mobile phones.)

3.2 Java Card Applets

Using the Java Card terminology, an on-card Java Card application is called an *applet*. Java Card applets run in the Java Card Virtual Machine of the card, under the supervision of the Java Card Runtime Environment.

The life-cycle of a Java Card Applet starts with its installation. Applets can be embedded into the card ROM while manufacturing, or at a later time. The installation of applets is concerned later in this section. The first step of the installation process establishes a secure connection between the reader side and the card. After the connection is established, the on-card installer application is selected on the card. At this point the card is ready to accept command APDUs containing the data of the CAP archive. Then the installer calls the install method of the applet. The applet should instantiate and register itself to the JCRE. In this point the applet is in deselected and in inactive state.

When a host application sends a SELECT APDU to the JCRE requesting a specified applet, the JCRE selects the applet, and the applet should take itself into active state. Subsequent command APDU-s will be dispatched to this active applet. The selected applet will remain selected until another applet is requested to be selected by the host application. When another SELECT APDU is received, the actually selected applet will be requested to do the necessary cleanup and deactivate itself then it will be deselected by the JCRE.

3.2.1 Implementing an applet

Creating an applet involves extending the `javacard.framework.Applet` class, overriding and implementing the life-cycle methods. A skeleton of a Java Card applet, from
(Oritz 2003b), extended with the formal parameter list:

```
import javacard.framework.*
...
public class MyApplet extends Applet {
    // Definitions of APDU-related instruction codes
    ...
    private MyApplet() {...} // Constructor

    // Life-cycle methods

    public static void install(byte[] data, short offs, byte len) {...}

    select() {...}

    deselect() {...}

    process(APDU apdu) {...}

    // Private methods
    ...
}
```

Life-cycle methods

The `MyApplet()` is called once, when the applet is instantiated in the `install()` method. The constructor is responsible for initializing the state of the applet while taking care of the memory limitations. It is advised that everything should be instantiated in the constructor, and these instances should be reused later during the lifetime of the applet. Usually the applet registration also takes place here by calling the `register()` method of the current applet instance.

The static `install()` method is called by the Java Card Runtime Environment during the installation process of the applet. The method should be overwritten in every applet, and it should instantiate the applet and register it, directly or through the

constructor. If installation properties are supported, they can be accessed using the method arguments.

The `select()` method is called by the JCRE to inform the applet that it has been selected for APDU processing. The method should return true if it is ready for data processing, or false if not. Applets which require PINs return false in this method when the internal counter PIN error counter exceeds a limit, hence disabling the card. The default behavior inherited from the `Applet` class returns the value of *true*.

The `deselect()` method, is called to inform the currently selected applet that another applet will be selected. The applet should perform cleanup and other bookkeeping tasks and should return to an inactive unselected state. The method has a default, empty behavior defined in the Applet class.

The `process()` method is called every time when there is an APDU for the applet to process. The method usually performs the following operations: (Oritz 2003b)

- Extracts the CLA and INS parameters from the APDU. (See the smart card overview)
- Retrieves the P1, P2, and data fields if they are present.
- Processes the request
- Generates and sends response
- Returns gracefully or throws the appropriate ISOException.

The SELECT APDU (CLA=0x00, INS=0xA4) is also passed to the `process` method, when the applet is selected. If the method returns without throwing any exception, it means the processing of the requested instruction finished successfully.

A typical implementation of the `process` method is like the following. (Extended version of the example in (Oritz 2003b))

```
public void process(APDU apdu) throws ISOException {

    byte[] buffer = apdu.getBuffer();

    buffer[ISO7816.OFFSET_CLA] =
    (byte)(buffer[ISO7816.OFFSET_CLA] & (byte)0xFC);

    if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
        (buffer[ISO7816.OFFSET_INS] == (byte)(0xA4)) )
                /*
                 * Handle the select APDU
                 */

    //  If unrecognized class, return "unsupported class."
    if (buffer[ISO7816.OFFSET_CLA] != MyAPPLET_CLA)
       ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
```

```
    //  Process (application-specific) APDU commands aimed at
    //  MyApplet.
    switch (buffer[ISO7816.OFFSET_INS]) {

        case CUSTOM_1_INS:
            handleCustomInstruction1(apdu);
            break;

        case CUSTOM_2_INS:
            handleCustomInstruction2(apdu);
            break;

        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
            break;
    }
}
```

Processing APDUs

APDUs are processed using the `APDU` instances, which are passed by the runtime environment to the `process` method. The `APDU` instance owns a buffer, obtained using the `getBuffer()` method. This is used both as an input buffer for reading the data part of the command APDU, and for passing data in the response APDU generated by the command. According to the Java Card specification, the Applet cannot store any objects in instance variables which were passed to the applet by the JCRE. This involves APDU instances, buffers, etc.

First the command APDU data should be extracted. The length of the data in the command APDU can be queried using the following construct:

```
short dataLength = (short)buffer[ISO7816.OFFSET_LC];
```

The data has to be received using the `apdu.setIncomingAndReceive()` method. This will fill as much data as the APDU buffer allows without overflow. If possible, it will read all the data. The return value of the method indicates how many bytes have been read into the buffer.

If this number is smaller than the size in the LC field, the —apdu.receiveBytes()— method should be called, and the whole data parameter should be assembled from small chunks of data. The `receiveBytes()` method fills the APDU buffer, reading as many bytes as possible. The operation can be performed like the following:

```
...
byte[] buffer = apdu.getBuffer();
short bytes_left = (short) buffer[ISO.OFFSET_LC];
short readCount = apdu.setIncomingAndReceive();
byte[] tbuf = new byte[bytes_left];
short tbuf_offset = 0;

while (bytes_left > 0) {

    // Process received data in buffer; copy chunk to temp buf.
```

```
    Util.arrayCopy(buffer, ISO.OFFSET_CDATA, tbuf, tbuf_offset, readCount);
    bytes_left -= readCount;
    tbuf_offset += readCount;
    // Get more data
    readCount = apdu.receiveBytes(ISO.OFFSET_CDDATA);
}
...
```

(The source code is from (Oritz 2003b), but it had to be modified as in the original code, the chunks were copied to the offset 0 of the `tbuf`, overwriting the previous chunk. The `tbuf_offset` variable was introduced to maintain the position of the next chunk.

After processing the input data, the APDU instance has to be set to *outgoing* mode. the `apdu.setOutgoing()` will do this, returning the LE (length expected) field from the command APDU. If the value is not desired, a *wrong response size* exception can be thrown. After filling the APDU buffer from the offset zero, it has to be sent with the `apdu.sendBytes(offset, length)`, and our method can return.

3.2.2 The Counter applet

The counter applet, the demo used to demonstrate the Java Card development, uses three instructions:

Name	Code	Description
`INS_GET_VALUE`	0x10	Queries the value of the counter Returns a byte value of the counter
`INS_INCREMENT`	0x20	Increments the counter
`INS_RESET`	0x30	Resets the counter to zero

None of the instructions specify any P1, P2 parameters or data in the command APDU-s. The only instruction which returns data, is the incrementer. This reports the one byte value of the counter.

The source code of the applet is the following:

```
package hu.juhasz.sandor.javacard.counter;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;

public class Counter extends Applet {

    private static final byte INS_GET_VALUE = (byte)0x10;
    private static final byte INS_INCREMENT = (byte)0x20;
    private static final byte INS_RESET     = (byte)0x30;

    private static final byte CLA_COUNTER   = (byte) 0x80;
```

```
    private byte ctr;

    private Counter() {
        ctr = 0;
        register();
    }

    public static void install(byte[] args, short offset, byte len) {
        Counter counter = new Counter();
    }

    public void process(APDU apdu) {
        byte[] buffer = apdu.getBuffer();

        if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
            (buffer[ISO7816.OFFSET_INS] == (byte)(0xA4)))
            return;

              if (buffer[ISO7816.OFFSET_CLA] != CLA_COUNTER)
                  ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

       switch (buffer[ISO7816.OFFSET_INS]) {
                  case INS_GET_VALUE:
                      getValue(apdu);
                      break;
                  case INS_INCREMENT:
                      increment(apdu);
                      break;
            case INS_RESET:
                 reset(apdu);
                 break;
                  default:
                      ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
             }
    }

    private void increment(APDU apdu) {
              ctr++;
    }

    private void reset(APDU apdu) {
        ctr = 0;
    }

    private void getValue(APDU apdu) {
              byte[] buffer = apdu.getBuffer();
              short le = apdu.setOutgoing();

              if ( le != 1 )
                  ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

        apdu.setOutgoingLength((byte)2);
              buffer[0] = ctr;
              apdu.sendBytes((short)0, (short)1);
    }

}
```

The install method instantiates the counter application. The constructor resets the counter and registers the applet.

As the increment and reset instructions do not perform any I/O, and there is no exceptional state, the code is trivial to understand. The `getValue` sends the value of the of the counter in the response APDU. As the instruction has no input, after acquiring the buffer there is immediate change to outgoing mode. After checking the LE field of

the command APDU, just the value to the buffer is written and send the result.

In the example applet, it is clearly visible that the Java Card API was designed for small devices.

- Most data communication is performed through byte arrays, which are reused as many times as possible. The addressing of the various APDU fields resembles an assembly language, rather than an object-oriented language.
- Card protocol specific ISO Exceptions are not instantiated and thrown by using the `throw` statement, but thrown using the static method of the ISOException class. This allows the use of Java Card runtime-owned instance. See the API documentation of the `ISOException` class for more details.

3.3 Java Card Development Kit

The Java Card Development Kit (JCDK) is a standalone development environment in which Java Card applets can be developed and tested. The package provides programming tools, emulation and simulation components, samples, documentation, and a reference implementation of the Java Card Runtime Environment.

As the Java Card 2.2.1 compatible hardware was used in the project, the same JCDK from the Sun's webpage was used too. There is a newer, 2.2.2 version available, but it is not compatible with 2.2.1 hardware.

3.3.1 Installation

The JCDK can be installed in a few easy steps.

1. Download and install the Sun JDK 1.4.1 SDK. The JCDK is not compatible with the more recent versions of the Java SDK.
2. Download the JCDK 2.2.1 from the `http://java.sun.com/javacard` page.
3. Unzip the .zip archive to a directory which does not contain any space in its name.
4. Set up the following environment variables:
   - `JAVA_HOME` to he JRE contained in the JDK 1.4.1 installation.
   - `JC_HOME` to the JCDK root directory.
   - `PAHT` to contain the `bin` directory of the JCDK.

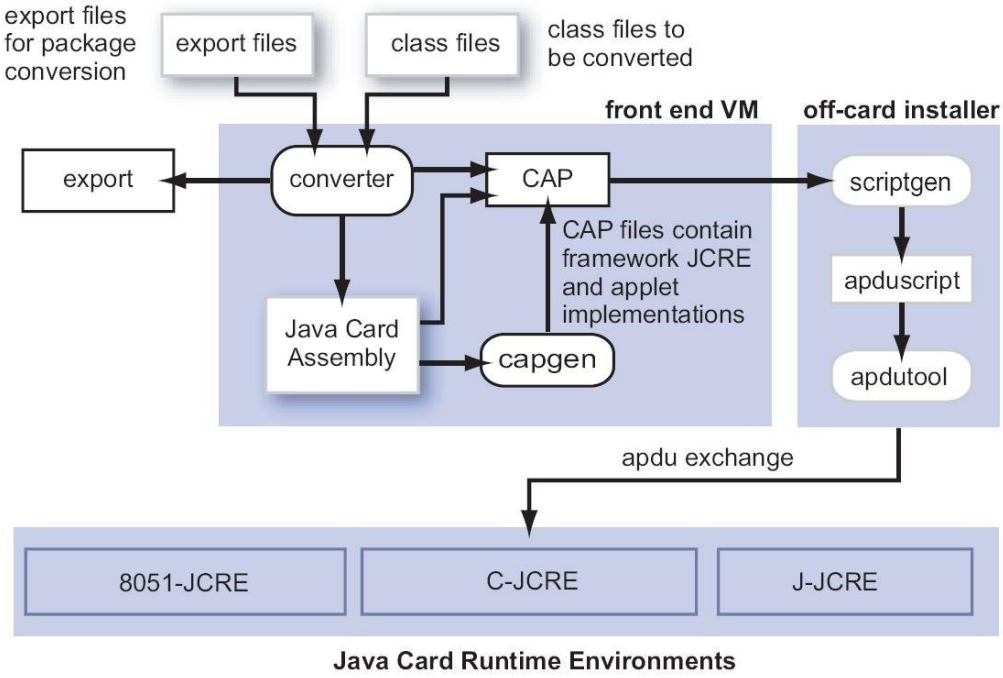

Figure 3: The Java Card Development Kit toolchain.
The image is copied from the Java Card Development Kit 2.2.2 Users Guide.

5. If the JDK is installed in the `Program Files` or any other directory containing whitespace in its name, the batch files in the JCDK should be edited. The call of the java interpreter should be enclosed in quotation marks.

   As an example, the line which calls the Java interpreter in the `apdutool.bat` file should look like the following:

```
"%JAVA_HOME%\bin\java" -classpath %_CLASSES%
   com.sun.javacard.converter.Converter
```

3.3.2 The JCDK toolchain

As described in the *Development Kit User's Guide* (Developer Kit User's Guide 2003), the JCDK uses many utilities to process various input files and to generate others. Although the number of file types and utilities makes the JCDK chaotic for the first time, these utilities form a well defined toolchain, illustrated on figure 3.

File types

This list of file types is syndicated from various sources like (Developer Kit User's Guide 2003), (Oritz 2003a) and (Oritz 2003b).

Source files Standard Java source file with `java` extension.

Export files (`.exp` extension) Contain the public linking information of classes in a package. The export files are not used directly by the JCRE, but the information they contain is critical for the working of the JCVM. Export files are used for linking of packages during conversion. When an application being converted uses another package, the export files of these other packages should be available to the converter utility. The export files of the JCRE are included in the `api_export_files` directory of the JCDE. Export files can be generated with the converter utility.

CAP (`.cap`) Acronym of *converted applet package*. Binary representation of the converted package. CAP files can be generated using the converter utility, and can be installed on JCRE-s using off-card installers. (Eg. GPShell, or scriptgen + apdutool) CAP files are standard JAR archives also containing a manifest file, so they are self-describing files.

Java Card Assembly (`.jca` extension) A text representation of a CAP file. Used for masking applets into the read-only memory of a card during manufacturing. Can be useful for debugging and tracing the conversion process. Further details can be found in (Ort 2001a). Detailed description of the file format can be found in (Developer Kit User's Guide 2003).

APDU scripts Contain series of APDU commands. Can be used for testing applications loaded into JCRE-s, or for installing CAP files converted into APDU commands. The apdutool can be used to send the commands listed in the file to the actual JCRE.

Toolchain programs

Java compiler The standard JDK 1.4.1 java compiler.

Converter The JCDK utility which takes export files and .class files compiled using the java compiler and generates export, JCA and CAP files from them.

capgen Generates a CAP file from a JCA file.

scriptgen Generates installations script from CAP files for installation to the JCRE reference implementation.

apdutool can parse APDU scripts and send command APDUs to the destination JCRE. The target device can be the emulator, the reference implementation or a real smart card.

3.3.3 Compiling source files

Source files are compiled using the standard Java compiler. The `-g` option should be specified when compiling applets, as the converter utility uses the debugging information during the conversion. The proper classpath settings should be set to successfully compile the files. Optimization should not be performed during compilation as the java optimizer does not optimize for the limitations of smart cards but for the capabilities of desktop computers.

Sample compile command from the (Developer Kit User's Guide 2003)

```
javac -g -classpath .\classes;..\lib\api.jar;..\lib\installer.jar
src\com\sun\javacard\samples\HelloWorld\*.java
```

3.3.4 Testing with the JCWDE

The Java Card Workstation Development Environment (JCWDE) allows the simulated running of a Java Card applet as if it was masked in ROM. It partly emulates the Java Card runtime environment, but it is not an implementation of the JCVM. It is only useful for initial applet testing. To use the emulator, there is no need to convert, mark and install the applets being tested.

The applets being tested should be in the classpath, and should be listed in the JCWDE configuration file. (`app` extension.) The configuration file contains an applet reference in every line. Every line has two components separated with whitespace. The first is the fully qualified applet name including package name, the second is the AID of the applet in hexadecimal format. The first applet should be always the installer applet. See the following example. The lines have been broken for formatting reasons.

```
com.sun.javacard.installer.InstallerApplet  0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0x8:0x1
hu.juhasz.sandor.javacard.counter.Counter   0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0x0c:0x11:0x2
```

The JCWDE should be executed by specifying the configuration file as a runtime argument. For detailed description of the command line arguments, see the (Developer Kit User's Guide 2003).

```
D:\StrongAuthentication\Work>jcwde Counter.app
Java Card 2.2.1 Workstation Development Environment, Version 1.3
Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to lic
ense terms.
jcwde is listening for T=0 Apdu's on TCP/IP port 9á025.
```

The JCWDE opens a server socket connection using port 9025 by default. The apdutool can connect to this port, and command APDU-s can be sent to the emulator like to any

other JCRE. For the proper syntax of the APDU scripts, see the description of the APDUTool utility.

3.3.5 APDUtool

The APDUTool is used to send command APDU-s to the JCWDE and the JCRE-s. Its primary input is a script file (*.scr) which contains for the APDUTool and command APDU-s for the JCRE. The command-line arguments of the `apdutool` command are well documented in the (Developer Kit User's Guide 2003). In this project the APDUTool was only used to send test commands to the JCWDE. In this case only the script file name is needed, nothing else. An example output of the tool:

```
D:\StrongAuthentication\Work>apdutool counterTest.scr
Java Card 2.2.1 APDU Tool, Version 1.3
Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to lic
ense terms.
Opening connection to localhost on port 9025.
Connected.
Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x00
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 09, a0, 00, 00, 00, 62, 03, 01, 08, 01, Le: 00,
 SW1: 90, SW2: 00
CLA: 80, INS: b8, P1: 00, P2: 00, Lc: 0c, 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06, 01,
 00, Le: 00, SW1: 64, SW2: 43
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06, 01, Le:
 00, SW1: 6d, SW2: 00
CLA: 80, INS: 20, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 6d, SW2: 00
CLA: 80, INS: 10, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 64, SW2: 21
```

The APDU script file can contain comments, command APDU-s and script commands. Comments are the well-known Java comments, eg. `//`, `/* .. */` and `/** ... */`.

A command APDU can be put in one line. The bytes of the command are separated with space, and the command is terminated with a semicolon. The bytes of the command can be specified as decimal or hexadecimal numbers. In the later case the standard `0x..` prefix is needed to indicate hexadecimal notation. A command consists of the standard APDU parts:

```
CLA INS P1 P2 LC [byte0, byte2, ... ,byte(LC-1)] LE ;
```

The script file can contain commands for the APDUTool itself. There are commands for power up and power down, specifying contact or contactless protocols, outputting strings, switching on or off extended APDU support, etc. See (Developer Kit User's Guide 2003) for details. The only important commands in our case are the `PowerUp;` and `PowerDown` commands. The power up command should be the first command in a script file, and the power down the last.

An example APDUTool script file testing the Counter applet:

```
PowerUp;
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;
0x80 0xB8 0x00 0x00 0x0c 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0x0c 0x06 0x1 0x00 0x7F;
0x00 0xA4 0x04 0x00 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0x0c 0x06 0x1 0x00;
0x80 0x20 0x00 0x00 0x00 0x00;
0x80 0x10 0x00 0x00 0x00 0x01;
PowerDown;
```

3.3.6 Converter

The Converter utility takes export files (version 2.2.x and 2.1.x) and .class files compiled using the java compiler and generates export, JCA and CAP files from them. During the first phase of conversion, the utility checks if the classes comply to the limited language elements defined in the Java Card specification, and checks the export files for correctness. In the second phase it preprocesses the classes: initializes static fields, resolves symbolic references and optimizes the classes for the limited environment of the Java Card platform.

The command line syntax for the converter utility:

```
converter [options] <package name> <package aid> <major>.<minor>
```

The most common command line options are the following. For complete reference see the reference manual.

`-applet <aid> <classname>`	Assigns the AID to an applet. If more applets can be found in the package, this option is needed for all of them.
`-classdir <root>`	Root directory of the classes being converted.
`-exportpath "dir"`	Specifies the directory for the export files.
`-out [CAP] [EXP] [JCA]`	Specifies the output targets.

The converter utility can be parametrized using a configuration file. (Usually having .opt extension) The configuration file contains the command line arguments specified to the converter, one at a line. The order and the syntax of the parameters are the same as the command line usage.

The example configuration file for the converter applet:

```
-out EXP JCA CAP
-exportpath "d:\java_card_kit-2_2_1\api_export_files"
-applet  0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0x0c:0x11:0x2
   hu.juhasz.sandor.javacard.counter.Counter
hu.juhasz.sandor.javacard.counter
0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0x0c:0x11 1.0
```

Calling the converter utility using a configuration file:

```
D:\StrongAuthentication\Work>converter -config Counter.opt

Java Card 2.2.1 Class File Converter, Version 1.3
Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to lic
ense terms.


conversion completed with 0 errors and 0 warnings.
```

3.4 Supporting tools

As it can be seen from the previous sections, developing, testing and building Java Card applets involves a lot of work with hexadecimal numbers. To help making the application development faster, some utilities were developed using Windows batch files and the Ruby programming language.

All the source code, utility scripts and work directories are located in a directory, called `d:\StrongAuth` in the test system. This contains the `Codebase`, `Dist`, `Work` directories. The source code of different projects developed during the pilot is located in the `Codebase` directory. The final, installable and executable versions of the projects can be found in the `Dist` folder. The `Work` directory is used by the build scripts to store generated and compiled files. The build utility scripts `generateAPDUScript.rb`, `generateAppFile.rb` and `generateConverterOptFile.rb` are located in the root of the `d:\StringAuth` directory.

3.4.1 Java Card projects

Java Card projects (`Counter` and `AuthenticationCardlet`) have common directory layout. Every project directory has three subdirectories for storing documentation (`/Docs`), source code (`/Source`) and test cases (`/Test`). The project configuration file called `project.ini` can be found in the root directory of the project.

The `/project.ini` file is used to associate applets with an AID, and an alias. This information is used later in the build process. The file can contain applet descriptions, separated with empty lines. All applet description contains four lines, as shown in the following example.

```
applet
alias: AuthApplet
class: hu.juhasz.sandor.javacard.auth.AuthApplet
AID: 0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0x0c:0x10:0x1
```

The first line indicates that this block is an applet descriptor. The second line contains the `alias:` keyword and the applet alias, separated with a TAB character. The third line associates the Java class to the applet, the last line assigns the AID to the applet.

The test files found in the `/Test` directory are APDU script files (described in 3.3.5) which use an extended syntax developed by the author. The `PowerUp` and `PowerDown` APDUTool commands work as specified in the JCDK. Command APDUs are specified using one of the following three commands: `select <appletAlias>`, `install <appletAlias>`, and `apdu CLA INS P1 P2 LC DATA LE`. As an applet alias, the `installer` string or an alias defined in the project file can be used. CLA and instruction constants defined in the source file can be also used in command APDU-s. APDUs can be also specified in the original format of the APDUTool. The test file preprocessor will not alter those lines.

An example test file:

```
PowerUp;
select installer ;
install AuthApplet ;
select AuthApplet ;

apdu CLA_AUTH INS_SEND_PIN 0x00 0x00 0x04 0x39 0x39 0x39 0x39 0x00;

apdu CLA_AUTH INS_GET_PUBLIC_KEY 0x00 0x00 0x00 0xF9;

apdu CLA_AUTH INS_AUTH 0x00 0x00 64 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
      7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
      8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 192 ;

PowerDown;
```

3.4.2 The generateAPDUScript utility

The GenerateAPDUScript utility is a script file written in the Ruby programming language. From the `project.ini` and a test file as input it generates an APDU script which can be later executed. The utility expects three command line arguments: test input file, test output file and the project file.

The utility scans all the Java source files in the project associated with the specified project file. All the public static final fields and their values are extracted from the source files and stored in a map. The `project.ini` is also parsed for obtaining the alias and AID values. As a final step the test file is read line by line, and the extended commands like `apdu`, `select` and `install` are replaced with standard APDU byte values.

3.4.3 The generateAppFile utility

Ruby script which generates a JCWDE `.app` file from the `project.ini` file. The first command line argument is the project file, the second parameter is the output `.app` filename.

The script first inserts the installer applet class and its AID in the first line, then adds a line with similar contents for every applet defined in the project file.

3.4.4 The generateOptFile utility

Ruby script to generate an `.opt` file from the `project.ini` file. This `.opt` file can be used by the `converter` utility. The first command line argument is the project file, the second parameter is the output `.opt` filename.

3.4.5 The build script

Every project has a project specific build script in the root of the `d:\StrongAuth` directory. This is a Windows batch file for building the project. As batch files can provide very limited functionality, the build files are not generalized. As a further improvement the build system could be rewritten in Ruby to support easier configuration.

An example build file used for the Counter applet is the following:

```
@echo off
set JAVA_HOME=c:\Program Files\java\jdk1.4.1\jre
set JC_HOME=d:\java_card_kit-2_2_1
set PATH=.;c:\Program Files\java\jdk1.4.1\bin;
         d:\java_card_kit-2_2_1\bin;%PATH%
set CLASSPATH=%CLASSPATH%;%JC_HOME%\lib\api.jar

cd Codebase\Counter\Source
javac -g -d ..\..\..\work hu\juhasz\sandor\javacard\counter\*.java
if errorlevel 1 goto JAVAERROR

cd ..\..\..

generateAppFile    Codebase\Counter\project.ini work\Counter.app
generateAPDUScript Codebase\Counter\Test\test.scr work\counterTest.scr
    Codebase\Counter\project.ini
generateOptFile    Codebase\Counter\project.ini work\Counter.opt
cd  work
converter -config Counter.opt
mkdir ..\Dist\Counter
copy hu\juhasz\sandor\javacard\counter\javacard\counter.cap ..\Dist\Counter


goto END

:JAVAERROR
cd ..\..\..
goto END

:END
```

3.5 Installing card applications

As Java Cards are security-related devices containing highly sensitive data, the applet management tasks like downloading and installing applets to cards, deleting them, should be allowed only for authorized personnel, utilizing a secure channel. The JavaCard specification includes APDU specifications on downloading the applets to cards, but does not contain any specific information how the secure connection should be established between the card reader and the card. This is the point where the *Global Platform* standard comes into the picture.

During development the Nokia 6131 NFC, and NXP JCOP41 contactless smart cards were used. Both cards use Global Platform 2.1.1 for opening secure connections to the cards. The open source *GPShell* application can be used to install and administer applets on cards. The application is available at `http://sourceforge.net/projects/globalplatform`. As its readme file states, "GPShell is a script interpreter which talks to a smart card. It is written on top of the GlobalPlatform library, which was developed by Karsten Ohme. It uses smart card communication protocols ISO-7816-4 and OpenPlatform 2.0.1 and GlobalPlatform 2.1.1. It can establish a secure channel with a smart card, load, instantiate, delete, list applets on a smart card."

The application trying to establish a secure connection to a Global Platform card device should provide authentication keys to connect. According to the *Nokia 6131 NFC SDK Users Guide* from the Nokia 6131 NFC SDK documentation, every 6131 NFC secure element contains chip-specific, secret device-specific authentication keys. Anyone who wants to install applets on a card or secure element, should provide these keys.

The secret keys are not available for third party developers, but the Nokia 6131 NFC Secure Element can be *unlocked* to allow secure element development. In this case the keys are changed to a default value by a company authorized for this operation. After unlocking the secure element, developers can download and test applications, but the secure element will not be *trusted* by third party applications which need secure and trusted connection between the service provider and the secure element. This means that no real application (like a real credit card application) can be installed on the card any more. The unlocking is permanent and irreversible, and can be performed only on factory-default clean secure elements not containing any applets.

For opening a secure connection, MAC+ENC security model should be used with *key set version* 42. The default key values for an unlocked 6131 secure element use the same value:

Key name	value
MAC key	404142434445464748494A4B4C4D4E4F
ENC key	404142434445464748494A4B4C4D4E4F
KEK key	404142434445464748494A4B4C4D4E4F

After performing *ten* unsuccessful authentications, the secure element becomes permanently locked. Applets installed on the secure element will be kept available but no applets can be removed or installed.

The factory default NXP JCOP41 cards use only MAC authentication with *key index* and *key version* 0, using the same values for keys as the unlocked Nokia 6131 NFC.

The GPShell application executes script files specified as the first and only command line argument. The file format is very straightforward, and there are ready made examples in the distribution of the software for installing, deleting and listing applets.

Example command line:

```
c:\GPShell> gpshell helloInstallNokia6131.txt
```

As an example here is the demo install scripts provided with the GPShell. The scripts install the HelloWorld Java Card SDK demo application.

The Nokia 6131 NFC specific install script:

```
mode_211
enable_trace
establish_context
card_connect -reader "OMNIKEY CardMan 5x21-CL 0"
open_sc -security 3 -keyver 42
    -mac_key 404142434445464748494A4B4C4D4E4F
    -enc_key 404142434445464748494A4B4C4D4E4F
    -kek_key 404142434445464748494A4B4C4D4E4F
delete -AID a00000006203010c0101
delete -AID a00000006203010c01
delete -AID a00000006203010c0101
install -file HelloWorld.cap -priv 2
card_disconnect
release_context
```

The generic GlobalPlatform 2.1.1 script for the NXP cards:

```
mode_211
enable_trace
establish_context
card_connect -reader "OMNIKEY CardMan 5x21-CL 0"
select -AID a000000003000000
open_sc -security 1 -keyind 0 -keyver 0
    -mac_key 404142434445464748494a4b4c4d4e4f
    -enc_key 404142434445464748494a4b4c4d4e4f // Open secure channel
delete -AID a00000006203010c0101
delete -AID a00000006203010c01
install -file HelloWorld.cap -sdAID a000000003000000
    -nvDataLimit 2000 -instParam 00 -priv 2
card_disconnect
release_context
```

All the commands should be in one line. In the `card_connect` command the name of the card reader (specified during the installation of the card reader) should be specified.

First the application and its package - if installed - should be deleted with the `delete` commands.

3.6 Reader-side Java applications

Reader-side applications are traditionally implemented in the C programming language using proprietary card reader access libraries. Thanks to the PC/SC SmartCard integration specification, modern smart card readers can be accessed in a centralized interface on various platforms. PC/SC is supported by default in Microsoft Windows 200x/XP operating systems, and for Linux and MacOSX, a free implementation called PC/SC Lite is available. (See `http://pcsclite.alioth.debian.org` for details)

As the author is familiar with the Java programming language, the most obvious choice was to use the Java platform to develop the reader-side applications. Regarding the Java SE platform, application developers have many options to connect to card readers. On Java ME platform many handheld devices support the Security and Trust Services API (SATSA) which allows APDU-based communication between Java MIDlets and smart cards. (Usually the phone SIM card) Before the release of Java 6, on the Java SE platform the Open Card Framework (OCF) (`www.opencard.org`) was the only possibility to to access smart card readers. The OCF is a pure Java framework which supports plugins (Resource Managers) for different Smart Card hardware. It has PC/SC connectivity support.

From the Java SE 6, the Smart Card I/O package is included in the Java Runtime Environment. Using this package applications can connect to smart card readers supporting PC/SC without third party libraries. In this section this package is introduced. A small reader-side application is developed as a front-end for the Counter applet. The application will display the counter value and increments it by communicating with the smart card.

3.6.1 The Smart card I/O API

The Smart Card I/O library is located in the `javax.smartcardio` package, and only supported on Java SE 6 and onwards. Official article about the library besides the Java Reference API has been yet found. See (Java Smart Card I/O API reference. 2006) for details. The dependencies of the classes can be seen in the figure 4.

The entry point to the library is the `TerminalFactory`, which is a factory for CardTerminal objects. A `TerminalFactory` instance can be instantiated using the `getDefault` and `getInstance` methods. When using the `getInstance` method, a terminal factory type string is needed to be specified. As of the Java SE 6 implementation, the only supported terminal factory type is the "`PC/SC`". In the

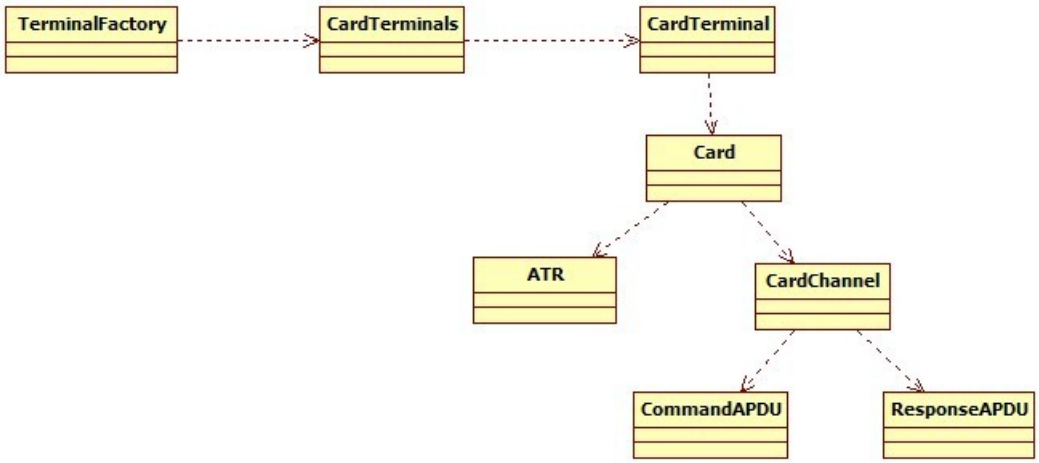

Figure 4: Dependencies between the classes of the Smart card I/O API.

development environment the Omnikey 5321 dual interface card reader was used which supports PC/SC. As no other types of card readers were present, the `getDefault` method returned the needed terminal factory.

The connected card readers can be queried using the `terminals()` method of the TerminalFactory instance, which returns a `Terminals` instance. The `Terminals` instance maintains a list of `CardTerminal` instances (listed using the `list()` method), and it can be also used to wait for changes in the availability of card terminals. Card readers can be also queried using their names, which is assigned to them during the hardware installation.

A `CardTerminal` instance represents a card reader attached to the computer. Its name can be queried, and it has methods for checking the card availability, for waiting for insertion or removal of cards, and for connecting to cards.

The `connect(String protocol)` method of the `CardTerminal` is used to connect to the smart card in the reader. The possible values of the protocol argument are "`T=0`", "`T=1`", "`T=CL`" and "`*`". If the last value is present, it tries to connect using any available protocol. The `connect` method returns a `Card` instance. A card instance represents the physical smart card. Methods for requesting exclusive access, for getting the ATR value of the card, and for opening communication channels exist in this class. In the applications the `getBasicChannel()` method is used to open a basic logical channel to the card.

After the `CardChannel` instance is returned by the Card instance, the devices are ready to communicate. The `transmit` methods can be used to send APDU commands and receive responses. Two implementations are present in the class. One uses `ByteBuffer`-s for storing the command and response APDU-s, the other uses the `CommandAPDU` and `ResponseAPDU` instances to further abstract the communication. The `CommandAPDU` and

`ResponseAPDU` classes provide a very convenient interface for creating APDU-s and accessing APDU fields.

3.6.2 The CounterTester application

The CounterTester application first asks the user to select a connected card reader. After the selection, the application starts a background thread which handles the card events, and displays the counter form. If a card is present in the reader, the background thread notices this. It increments the counter and then reads the new value. Finally it reports the new value back to the application, and waits for a new card to be inserted.

The available terminal names are loaded into the `CardTerminalSelector` form when the application is started, using the following code.

```
CardTerminals terminals;
if (TerminalFactory.getDefaultType().equals("None")) {
    return;
    // No PC/SC card reader driver installed.
}

terminals = TerminalFactory.getDefault().terminals();
try {
    Iterator<CardTerminal> it = terminals.list().iterator();
    while (it.hasNext()) {
        CardTerminal terminal = it.next();
        cardTerminalSelector.addTerminal(terminal.getName());
    }
} catch (CardException e) {
    return;
}
```

The `CounterTestThread`, which is started when the card reader is selected and runs till the application exits, contains a simple loop for handling cards.

```
public void run() {
    CardTerminals terminals = TerminalFactory.getDefault().terminals();
    CardTerminal terminal = terminals.getTerminal(terminalName);
    String terminalName = terminal.getName();

    while (running) {
        try {
            if (terminal.waitForCardPresent(3000)) {
                Card card = terminal.connect("*");
                processCard(card);

                card.disconnect(true);
                terminal.waitForCardAbsent(10000);
            }
        } catch (CardException e) {
            System.out.println("Card Exception happaned: "+e.getMessage());
        }
    }
}
```

The card reader name is assigned to the `terminalName` field before starting the thread. The `processCard()` method opens a communication channel, selects the applet on the

card, and communicates with the applet using its APDU interface. The method is quite self-describing.

```
private void processCard(Card card) throws CardException {
    System.out.print("   Requesting basic channel ... ");
    CardChannel channel = card.getBasicChannel();
    System.out.println("OK.");

    System.out.print("   Selecting the Counter applet ...");
    CommandAPDU selectCounter = new CommandAPDU(
            0,
            0xA4,
            0x04, 0x00,
            new byte[] { (byte)0xA0, (byte)0x00, (byte)0x00, (byte)0x00,
                         (byte)0x62, (byte)0x03, (byte)0x01, (byte)0x0C,
                         (byte)0x11, (byte)0x2 },
            0x7F
            );
    ResponseAPDU response = channel.transmit(selectCounter);
    if (response.getSW() == 0x9000)
        System.out.println(" Ok.");
    else {
        System.out.println("Error: ");
        System.out.println(response.getSW());
        return;
    }

    CommandAPDU incCommand = new CommandAPDU(0x80, 0x20, 0x00,0x00, 0x00);

    System.out.println("   Sending the increment command ... ");
    response = channel.transmit(incCommand);
    if (response.getSW() == 0x9000) {
        System.out.println(" Ok.");
        System.out.print("   Data: ");
        byte[] data = response.getData();
        for (int i=0; i<data.length; i++) {
            System.out.print(data[i]);
            System.out.print(" ");
        }
        System.out.println();
    } else {
        System.out.println("Error: ");
        System.out.println(response.getSW());
        return;
    }

    CommandAPDU getCounterCommand =
        new CommandAPDU(0x80, 0x10, 0x00,0x00, 0x01);

    System.out.println("   Sending the get command ... ");
    response = channel.transmit(getCounterCommand);
    if (response.getSW() == 0x9000) {
        System.out.println(" Ok.");
        System.out.print("   Data: ");
        byte[] data = response.getData();
        System.out.println(data[0]);
        parent.newCounterValue(data[0]);
    } else {
        System.out.println("Error: ");
        System.out.println(response.getSW());
        return;
    }
}
```

4 IMPLEMENTATION OF THE PROJECT

4.1 The StrongAuth pilot

After the theoretical research and the study of the Java Card Technology, the usage concept of the authentication application was needed to be designed.

From the user's point of view, the application should look like the following. There is a computer equipped with a card acceptance device. The user wants to use an application installed on this computer, which needs strong multi factor authentication of the user. The application is launched, and a login dialog box appears on the screen. The user enters the user name, and a PIN, then presses the login button. At this point the application asks the user to touch the NFC phone to the attached card reader. If the PIN is correct, and the phone being touched to the reader is associated to the user, access is granted to the system.

During the design of the usage scenario questions arose whether the PIN should be entered through the phone keyboard, or through the PC. On the Nokia 6131 NFC phone, keyboard events can be recorded only by writing Java Micro Edition based MIDlet applications. As the phone can only communicate with the card reader through the secure element, so the MIDlet has to have access to the secure element. As it is emphasized in the Nokia 6131 NFC SDK Programmer's Guide, MIDlets accessing the secure element should be signed by Certificate Authorities like Verisign or Thawte. As there were no appropriate certificates available in the project, and there were no resources to obtain one, the idea of using the phone keyboard as the PIN input device was dropped.

As there was no means to write MIDlets, the protocol had to be designed so that it uses the phone secure element as a traditional smart card. This implicated that the phone will be only a passive device, answering to the questions of the card reader.

To implement the whole StringAuth pilot application, the protocol between the CAD and the secure element had to be designed, the Java Card Applet had to be

implemented, and the host-side application had to be written. All the resulting artifacts are written to a CD. This CD also includes the software needed to compile and deploy the pilot application.

4.2 The Java Card Applet

The development of the applet involved three steps: designing the protocol between the reader side application, specifying the card instructions, and implementing the card application using the previous specifications.

4.2.1 Designing the protocol

As introduced in the theoretical part, the strong authentication protocol chosen for the project is a challenge-response protocol involves random numbers, asymmetric cryptography, and cryptographic hash functions. See page 17. for the protocol. Algorithms had to be chosen for every one of them, and their parameters. For the easier understanding, the scheme of the protocol is included once more.

	Alice		Bob
1.,	Choose random r_1	$\xrightarrow{r_1}$	
2.,		$\xleftarrow{r_2}$	Choose random r_2
3.,	$r = h(r_1 \cdot r_2)$		$r = h(r_1 \cdot r_2)$
4.,	$s = S_A(r)$	$\xrightarrow{s}$	Check $V_A(r, s)$

One of the main design considerations was that the authentication process should be performed using one APDU command, so all the data communicated between the card reader and the card applet should be able to fit into the APDU data part, which has the maximum size of 249 bytes. (The upcoming JavaCard 2.2.2 specification allows the *extended APDU syntax*, which allows 2^{16} bytes for data fields. The Nokia 6131 NFC does not support this standard.)

In the protocol being implemented, Alice generates the first random sequence called r_1, then Bob replies with r_2. The security of the protocol is not broken if the first two steps are switched. By switching these steps, one APDU command will be enough for executing the protocol. As Alice has the passive smart card device, Bob has to initiate the protocol. He can generate r_2 immediately and pass it to Alice during the authentication request.

To simplify the communication between the two parties further, Alice can send back the r_1 value at the same time when she sends the s value. The security of the protocol is not weakened if Bob calculates the r value and verifies the digital signature at the same time.

To implement the asymmetric protocol, the RSA algorithm was chosen, described in 2.1.6. The size of the encrypted text using RSA is equals size of the key. Considering the powers of two, in the response APDU 128 bytes of digital signature can be placed. This means 1024 bits of RSA key. As found in (Kaliski, 2003), the 1024 bit RSA provides about the same cryptographic strength as a 80 bit symmetric cryptosystem. The time when the 1024 bit RSA key can be broken is estimated around 2010. If this breakthrough is reached, the asymmetric protocol can be replaced for example with *Elliptic Curve cryptography* which uses smaller key sizes, and reasonably high degree of security. As there were more resources available for the RSA algorithm, this was chosen over the EC. For the message padding RSA PKCS v1.5 digital signature padding was used.

For the cryptographic has function h, the SHA-1 hash was chosen, over the MD5. (Both are supported by the Java Card 2.2.1 specification) The SHA-1 is considered more secure, providing 80 bit of security against collisions. The SHA-1 algorithm provides 512 bit (64 byte) length hash values. The length of the random r_1 and r_2 have been chosen to be 512 bits both.

Utilizing these design considerations, the actual authentication protocol between Alice and Bob is the following:

	Alice		Bob
1.,		$\overset{r_2}{\leftarrow}$	Choose random r_2, $\|r_2\| = 512bits$
2.,	Choose random r_1, $\|r_1\| = 512bits$		
3.,	$r = SHA1(r_1 \cdot r_2)$		
4.,	$s = RSAPRIVATE_A(r)$	$\overset{(s.r_1)}{\rightarrow}$	
5.,			$r = SHA1(r_1 \cdot r_2)$
6.,			Check $RSAPUBLIC_A(r, s)$;

Although the "Strong" part of the strong multi factor authentication is solved using this challenge-response authentication scheme, it is not a multi-factor authentication yet. A PIN was introduced which is needed to access the authentication applet functionality. It is a four-digit number, set to the default '9999' while installation, but can be changed later. The PIN can be specified to the card with All the card instructions need a successful PIN authentication performed using the `INS_SEND_PIN` command. With this the authentication is extended with an information the user "knows", besides the card the user "has".

4.2.2 APDU commands

To implement the desired functionality, the APDU command interface had to be designed between the reader side application and the applet on the smart card.

1. `INS_GET_PUBLIC_KEY` (0x10)

   Queries the 1024 bit RSA public key from the card. Successful PIN authentication is needed before using the command.

   - **Input data:** None.
   - **Output data:**

expSize	exp	modSize	modulus

     The expSize and modSize fields are 1 byte long. The public exponent used by the card by default is 0x010001, so the expSize is 3 in the majority of cards. The size of the modulus is 128 bytes using 1024 bit RSA.

2. `INS_AUTH` (0x20)

   Performs an authentication session. The reader side application asks for authentication by sending the r_1 value to the applet. The applet performs its part of the authentication protocol and responds with the (r_2, s) data. Successful PIN authentication is needed before invoking this command.

   - **Input data:**

r_1

     The r_1 value should be a 64 bytes long, cryptographically random string.
   - **Output data:**

r_2	l_s	s

     The r_2 is a cryptographically random, 64 bytes long string. l_s indicates the length of the s digital signature. s uses RSA PKCS v1.5 digital signature padding.

3. `INS_SEND_PIN` (0x30)

   Sends the PIN to authenticate to the card. After 3 unsuccessful PIN authentications the application is locked, and should be reinstalled.

   - **Input data:**

pin

     The PIN should be a 4-byte string consisting of ASCII codes of numbers. These bytes should be in the range of (0x30-0x39).
   - **Returned status word:**

     `SW_PINVERIFY_FAILED` (0x6900) if the authentication was not successful.
     `SW_NO_ERROR` (0x9000) if the authentication succeeded.

4. `INS_SET_PIN` (0x40)

   Sets the PIN stored in the applet to the one specified in the command APDU data field. Successful PIN authentication is needed before invoking this command. Specifying the previous PIN is not needed.

- **Input data:**
  newPIN
  The PIN should comform to the PIN format described previously.
- **Returned status word:** The instruction returns with the `SW_NO_ERROR` status word if the user has been authenticated successfully before.

4.2.3 Security specific Java Card API

While coding the applet, the cryptographic and security features of the Java Card API had to be used. These involve the PIN checking, the cryptographic random number generation, the RSA digital signatures, and the SHA-1 cryptographic hash.

PIN support

The Java Card API supports PIN verification and management with the `OwnerPIN` class. When instantiated a try limit and a maximum PIN size can be specified. The OwnerPIN instance has a *try counter* which counts the unsuccessful verifications. The object maintains a *validated flag*, which is true if a successful PIN verification was performed after the last reset. The *blocked flag* indicates if the card application is blocked or not.

While authenticating, the `check` method should be called, which accepts the PIN as part of a byte array. If the PIN matches with the stored one, it sets the validated flag to true, and returns true. If the verification was unsuccessful, the object increases the try counter, sets the validated flag to false, sets the blocked flag to true if the try counter exceeds the specified limit, and returns false.

The number of remaining tries before the OwnerPIN is blocked can be queried using the `getTriesRemaining()` method. The validated flag can be read using the `isValidated()` method. The `update` method is used to update the PIN to a new value.

When the application is deselected, it should reset the OwnerPIN instance using the `reset()` method. This means that if the validation flag is true, the flag is changed to false and the try counter is set to zero. If the validation flag is false, the try counter value is kept. To unblock a blocked OwnerPIN instance, the `resetAndUnblock()` method should be called.

The OwnerPIN instance should be instantiated in the constructor of the applet, and updated with the default PIN. In the applet `select` method the applet should verify if any tries are remaining. If not, it should return false meaning the card is blocked, and cannot be selected. When processing APDUs needing PIN verification, the validated flag of the OwnerPIN instance should be checked as a first task. If the validated flag is false, the applet should throw an ISOException, indicating the needed authentication.

Random number generation

Generating pseudo-random or cryptographically secure random numbers is straightforward using the Java Card API. The `RandomData` class used for this task can be found in the `javacard.security` package.

A random number generator should be obtained using the static `getInstance` method, which requires the type of the random number generator as an argument. The two type constants supported: `ALG_PSEUDO_RANDOM` and `ALG_SECURE_RANDOM`. The RandomData instance is advised to be stored as a field, and should be constructed in the constructor.

Random data is generated by calling the `generateData` method. This fills the specified region of the byte buffer passed as an argument.

Cryptographic hash functions

The Java Card API supports the MD5, RIPE MD-160 and SHA-1 algorithms through the `MessageDigest` class located in the `javacard.security` package. The object instance is created using the static `getInstance` method, the algorithm should be specified as a parameter. Possible values are `ALG_MD5`, `ALG_RIPEMD160`, and `ALG_SHA`.

Data can be passed to the MessageDigest object using the `update` and `doFinal` methods. The `update` is used to pass data chunks which are not the last chunks. The last data chunk should be passed using the `doFinal` method. This method has the output buffer as a parameter, the hash value will be copied into this output buffer.

The MessageDigest instance can be reset to its initial state for further use by calling the `reset` method. A card reset or a tear event will also reset the MessageDigest. If the second argument of the `getInstance` method is true, the instance is to be shared between applets, hence it will not be reset when the applet is deselected.

RSA under Java Cards

Key pairs for every asymmetric cryptosystems are stored in `KeyPair` instances. The properties of a keypair are passed to the constructor. The KeyPair class contains many algorithm constants and key lengths. In our case the `ALG_RSA` and `ALG_RSA_CRT` can be used with the length constant `LENGTH_RSA_1024`. As mentioned in the section 2.1.6, the private key operations of the RSA are less efficient than the public key operation. The Java Card implementation in the Nokia 6131 NFC and the NXP JCOP card does not support the pure RSA algorithm for 1024 bit keys. The modified version using the *Chinese remainder theorem* had to be used, which can be obtained with the

`ALG_RSA_CRT` constant. The key pair instance can be used to generate keys. The `genKeyPair()` method should be called to initialize the key with a new key pair.

The cipher performing the private and public operations is represented by the `Cipher` class, found in `javacardx.crypto` package. It can be instantiated using the `getInstance` static factory method, which has the same syntax as the one found in the MessageDigest class: the first parameter is the type of the cipher, the second is a Boolean value indicating if the instance will be shared between applets.

The ciphers supported by the Java Card 2.2.1 API are the AES, DES and RSA with different padding modes. In this case the `ALG_RSA_PKCS1` should be used.

After instantiating the cipher, it should be initialized by passing the key, and the mode flag for encryption or decryption. As in this case the private key operation is used to encrypt data, the initialization code looks like the following:

```
rsa.init(authKeyPair.getPrivate(), Cipher.MODE_ENCRYPT);
```

The data is fed into the Cipher class in the same was as into a MessageDigest instance. As the Cipher can produce ciphertexts with length depending on the length of the plain text (AES or DES for instance), the output buffer should be passed to the update method as well as to the `doFinal` method.

4.2.4 The AuthApplet

The AuthApplet implements the protocols and uses the algorithms described in the previous pages. The applet is contained in the `AuthApplet.java` source file. The full source code can be found in Appendix 1. The applet uses the AID of A0:00:00:00:62:03:01:0C:10:01. This is the Sun company ID A0:00:00:00:62:03:01:0C, with package 0x10 and Applet ID 0x01. This package ID is not used in the development kit examples.

The Applet defines private static final fields to store the instruction and error codes. These can be found as the first lines in the class definition.

The key fields of the application are the `authKeyPair` containing the RSA keypair, the `ownerPin` to store and manage the PIN, and the different instances using the cryptography and security API: `rand`, `rsa`, and `sha1`. These objects are instantiated and initialized in the constructor of the applet.

The `process` method verifies if the incoming command APDU has a valid instruction code, and delegates it to a processing method. The processing methods are:

`getPublicKey`, `auth`, `sendPin` and `setPin`. Their implementation is straightforward, and needs no explanation.

Unfortunately as the Java Card Development Kit does not support secure random number generation and RSA with longer keys than 512 bits, the applet cannot be tested directly on the JCDK.

4.2.5 Compiling and deploying the AuthApplet

The AuthApplet uses the customized project structure described in the previous chapter. The applet can be built by executing the `buildAuthApplet.bat` file in the root of the `d:\StrongAuth` directory. This will compile and package the applet into the `\Dist\AuthApplet` directory. The applet is installed using the `installAuthApplet.bat` and `installAuthAppletNokia.bat` files. The first one installs the applet to the NXP cards, the second one is used to install the applet to the Nokia 6131 NFC phones.

4.3 The StrongAuthLib

The card-side AuthApplet is useless without a reader-side application which can communicate with it. One of the main project requirements were that that the authentication service should be generic, and it should be able to integrated into any application easily. I decided to make a modular component containing the authentication classes. As I was using the Java platform to develop the reader-side application, and the project requirement did not involve any centralized authentication service just authentication to the local machine, the most obvious choice was to use the Java Swing graphical user interface library, and the filesystem to store authentication information.

The source files of the library are grouped in the `hu.juhasz.sandor.strongauth` package. This contains two sub-packages, named `service` and `ui`. The service package contains the authentication logic, the abstractions of the card application, and the key store access classes. The UI package provides an end-to-end solution for integrating the Swing-based authentication module into any Java application. The dependencies between the classes can be found in figure 5

4.3.1 The service package

The service package contains five classes.

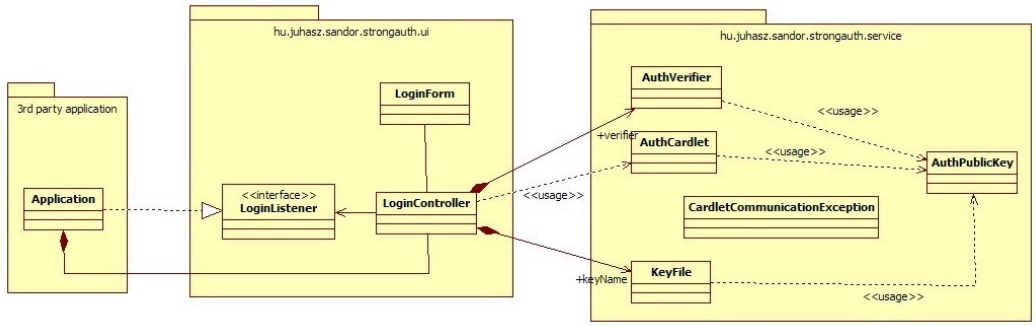

Figure 5: Dependencies of classes in the StrongAuthLib

AuthPublicKey

A simple class which can contain the parts of a public key. This includes the username as string, the key length, and the RSA specific public exponent and the modulus in byte arrays. The instance can be constructed using a string representing hexadecimal numbers, or byte arrays.

AuthCardlet

The `AuthCardlet` is an abstract interface to the AuthApplet card-side application. It provides an easy to use interface for the various commands supported by the AuthApplet. The `AuthCardlet` is instantiated by passing a CardChannel instance to its constructor. This means the communication channel between the reader and the card should be established before the AuthCardlet is created. The constructor selects the AuthApplet by sending a SELECT APDU.

During construction, exceptions can occur. The Smart Card I/O package can only report generic `CardException`, which is "exception for errors that occur during communication with the Smart Card stack or the card itself". Exceptions need to be reported if the returned status word from the card indicates an exceptional state. To make it more manageable, The `CardException` was extended in the `CardletCommunicationException` class. This class is responsible for reporting exceptional results while the communication regarding the APDU syntax is correct. In the constructor, if the applet cannot be selected (eg. not installed, blocked, etc) this exception is thrown.

All the methods of the `AuthCardlet` correspond to a command APDU defined in the card protocol. The methods throw `CardException`-s to notify the environment about errors.

The `sendPin` method can be used to send a PIN passed as a parameter to the card. The method returns true if the authentication succeeded, false otherwise.

The `getPublicKey` returns the `AuthPublicKey` instance containing the public key parts. To create the `AuthPublicKey` instance, the name of the user the instance belongs to should be specified.

The `authenticate` method sends the r_1 random string to the card, and returns a byte array containing the r_1 and s values. The r_1 value is passed to it as an argument, and the returned array has the structure of the response APDU data.

The PIN code stored in the AuthApplet can be changed using the `setPin` command. The new pin is passed as an argument, and the return value indicates if the pin change was successful or not.

AuthVerifier

This is the most complex class of the service model of the StrongAuthLib, the class which does all the cryptographic activities. It has a secure random number generator, an SHA-1 message digest, and an RSA cipher object as instance variables. These instances are creaded in the constructor.

This class uses the Java security architecture. All the interface classes for the various cryptographic primitives are located in the Java SE runtime library, but the actual implementations should be provided by third parties. While developing the library, the Bouncy Castle security provider (`http://www.bouncycastle.org/`) was used. This provider contains all the cryptographic functionality needed by the authentication application. Further information on the security architecture can be found in David Hook's excellent book on Java cryptography.(Hook 2005)

The `SecureRandom` secure random number generator is part of the Java SE runtime library, and it is located in the `java.security` package. The class can be instantiated using the default constructor. To fill byte arrays with random data, the `nextBytes` method should be called with the array as a parameter.

The `MessageDigest` class from the `java.security` package uses almost the same interface as the corresponding class from the Java Card library. The message digest instance is obtained using the `getInstance` static method, but instead of a constant, an algorithm string should be specified. In the case of the SHA-1, the string is "`SHA-1`". Data can be fed into the digester using the `update` method. Instead of the `doFinal`, the `digest` method can be used to pass the last chunk of data to the digester. If it is called without parameters, it returns the digest calculated by the chunks fed so far.

The `javax.crypto.Cipher` instance is also created using the `getInstance` factory method. The first argument of the method - a String - identifies the algorithm to be used. In our case it is "`RSA/None/PKCS1Padding`". The second, optional parameter specifies the security provider to be used, in case if more security providers support the requested algorithm. In our case this value is `BC` for the Bouncy Castle provider.

The API of the `Crypto` class is identical to the Java Card counterpart. The instance is initialized by selecting a mode (encrypt or decrypt) and specifying a key to be used. Data is fed into the cipher with the `update` method, and the cyphertext is obtained using the `doFinal`.

For initiating the authentication process, the `AuthVerifier`'s `generateR1` method is used. This generates 64 bytes of cryptographically secure random data, and returns it in a byte array.

When the response data containing the r_2 and s values arrive, it can be passed to the `verify` method. Three parameters are required to be specified for this method. The authentication key, with which the verification is performed; and the r_1 and the concatenated $r_2.s$ values, both in byte arrays.

The method feeds the r_1 and r_2 values into the message digester, and generates the SHA-1 value. In the second step, the RSA public function is performed on the s signature returned by the card. The returned value is compared to the calculated SHA-1 value. If they match, the authentication is successful.

Meanwhile, the method prints tracing information to the standard output to aid the demonstration of the authentication process.

KeyFile

In the pilot implementation developed for the LASSO project, the public keys imported from cards are stored in a key file in the local file system of the PC the authenticating takes place. Every line of the file contains a public key entry. The entry has three fields separated by TAB characters. The first is the username, the second is the public exponent, the third is the modulus. The numbers are in hexadecimal notation.

The `KeyFile` class can read a file like this, and store its contents in memory as a hash. The hash uses the username as key, and an `AuthPublicKey` instance as value. Authentication keys can be added, removed from the file, and queried using a username. The number of keys present in the file and the usernames can be queried, too.

4.3.2 The GUI interface

The main class of the authentication interface is the `LoginController`. When instantiated, it creates the `LoginFrame`, the `KeyFile` and the `AuthVerifier` instances which are used during the authentication process.

The `LoginController` informs the underlying application about the successful authentication or about an exit request through the `LoginListner` interface.

Figure 6: The login screen waiting for the presence of a card

After construction the `init` method should be called, which initialized the `LoginFrame` with the names of the available card readers. If the initialization was successful, the method returns true, false otherwise. It does not throw any exception, but pops up message boxes describing the errors.

The `LoginController` is activated by calling the `authenticate` method. This displays the frame, and waits for user interaction. If the user fills the login form correctly and requests an authentication by clicking the Login button, the `authenticationRequested` method is called in the `LoginController`. This method starts a thread, which tries to open a communication channel to the selected card reader, and performs an authentication session. If the authentication was successful, the underlying application is notified. If the authentication failed, this fact is shown to the user for a retry.

4.4 AuthDemo

The demo application for testing the authentication library is the simplest which can be imagined. On successful authentication, a "Hello world!" form is displayed on the screen. This is not a problem as the user authentication has nothing to do with the functionality of the application.

```
public class AuthDemo implements LoginListener {

    private LoginController loginController;

    public AuthDemo() {
        try {
            loginController = new LoginController(this,
                                                  "c:\\temp\\keyring.txt");
        } catch (IOException e) {
```

```
            JOptionPane.showMessageDialog(null, "I/O Error", "Error",
                    JOptionPane.ERROR_MESSAGE);
            System.exit(1);
        }
    }

    public void runApplication() {
        if (loginController.init()) {
            loginController.authenticate();
        }
    }


    public void authenticationSucceeded(String username) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JFrame app=new JFrame("My Application");
                app.getContentPane().add(new JLabel("Hello, World!"));
                app.pack();
                app.setVisible(true);
                app.addWindowListener(new WindowAdapter() {
                    public void windowClosing(WindowEvent e) {
                        exitRequested();
                    }
                });
            }
        });
    }

    public void exitRequested() {
        System.exit(0);
    }

    public static void main(String[] args) {
        Security.addProvider(new BouncyCastleProvider());

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                AuthDemo authDemo = new AuthDemo();
                authDemo.runApplication();
            }
        });
    }
}
```

The application is quite straightforward, and successfully demonstrates that the whole authentication system can be very easily integrated into a Java application.

4.5 KeyringManager

The Keyring manager application is used to manage the key file. The implementation uses the file `c:\temp\keyring.txt`.

The main functionalities provided by this application are the following:

- Importing public keys from cards.
- Deleting public keys from the key file.

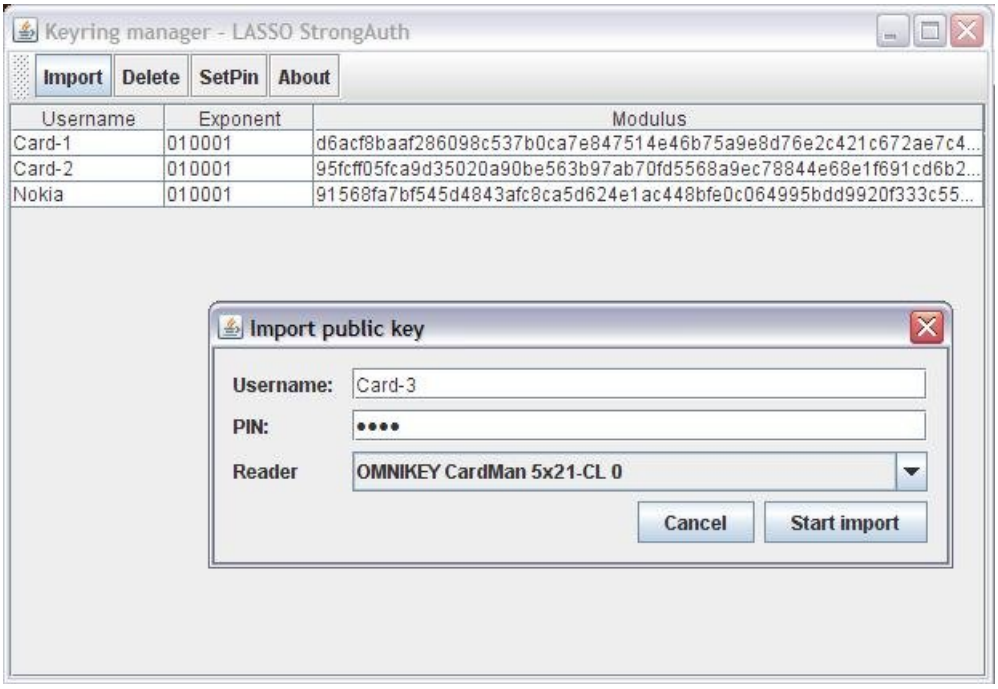

Figure 7: The Keyring Manager application while importing a public key

- Changing the PIN of an authentication card.

The main class of the application is the `KeyringManager`. It instantiates the various forms, and manages the workflow of the application. The three forms communicate using interfaces with the `KeyringManager` instance. The communication with the card reader is handled in background threads (`PinChanger` and `PublicKeyImporter`), and the `KeyringManager` is notified of their progress through interfaces.

4.6 Deploying and running the Strong authentication demo

As a testing and demonstration environment for the porject, the following hardware and software elements were used.

- Omnikey 5421 RFID contactless card reader
- PC running Microsoft Windows XP operating system.
- NXP JCOP41 or compatible smart cards
- Nokia 6131 NFC mobile phone with unlocked secure element

The installation process needed to execute the application is described in this section. All the installation steps should be performed by an user who has administrative rights on the local computer.

4.6.1 Installing the Omnikey CardMan 5321 RFID card reader

To install the device, the latest drivers can be downloaded from the Omnikey homepage (`http://omnikey.aaitg.com`. Three driver packages should be downloaded from the page: The *CardMan 5x2x PC/SC drivers*, the *CT-API for Windows* package and the *CardMan Synchronous API for Windows* archive. These packages can be also found in the project CD, in the `\Install\Omnikey` directory.

First the *CardMan 5x2x PC/SC driver* should be installed. Detailed instructions can be found about the installation process in the `readme_cm5x21_5x25.txt` file included in the driver package. After the card reader is installed properly, the *CT_API* and *Synchronous API* should be installed by running the `setup.exe` files. Although the installation process is straigthforward, further information can be found in the PDF files located in the driver directories.

As the Omnikey CardMan 5321 RFID card reader includes both contact and contactless interfaces, the device will be recognized as two card readers. All the card readers installed in the system should have an unique identifier. In the project configuration files these identifiers are assumed to be the following:

`OMNIKEY CardMan 5x21 0`	CardMan 5421 Contact interface
`OMNIKEY CardMan 5x21-CL 0`	CardMan 5421 Contactless interface

After a successful installation a new application can be found in the Windows Control Panel. Using this application the card reader can be tested.

4.6.2 Installing the Java SE Runtime Environment

To successfully run the demo, the SUN Java Platform, Standard Edition 6 should be installed. The version of the Java platform installed can be checked by performing the following steps. The Java Control Panel should be opened by clicking the *Java* icon in the Windows Control Panel. The version number can be queried by clicking the *About...* button in the *General* tab.

If the Java platform is not installed yet, it should be downloaded from the `http://java.sun.com` homepage. The installation is straightforward, only the directions displayed on the screen should be followed.

The Java SE Development Kit 6 Update 3 can be also found in the `\Install` directory of the project CD.

4.6.3 Installing the GPShell utility

The latest GPShell release should be downloaded from the project homepage (`http://sourceforge.net/projects/globalplatform/`) The utility comes as a ZIP file, which should be extracted to a desired location. The path to the `GPShell.exe` should be part of the PATH environment variable. The variable can be set by selecting the *Properties* by right clicking the *My Computer* icon, then by clicking the *Environment Variables* button on the *Advanced* tab.

4.6.4 Installing the AuthApplet to the smart cards

The AuthApplet can be installed to the NXP cards by executing the `installAuthApplet.bat` file. For installing the applet to the Nokia 6131 NFC phone, the `installAuthAppletNokia.bat` file should be used.

The target card should be in the card reader when the installer is executed, as it will not wait for the presence of the card, but will terminate with an error message.

Warning! As the cards become locked after ten unsuccessful card authentication. If the installation process terminates with card authentication error, it is advised to install a known to be working card applet first by which the lock counter can be reset. The Hello applet can be used for this purposes which is included in the GPShell package.

4.6.5 Running the demo

The demo applications read and write the `c:\temp\keyring.txt` file. The path is hard coded in the applications. To execute the demos correctly, the `c:\temp` directory should exist and the actual user has to have read and write permissions to the directory mentioned above.

The KeyringManager can be executed by executing the `java -jar KeyringManager.jar` command from the `\Dist\KeyringManager` directory on the CD.

The AuthDemo can be run using the `java -jar AuthDemo.jar` command from the `\Dist\AuthDemo` directory. The card reader should be connected to the PC when the applications are executed.

4.7 Testing the application

The strong authentication pilot application was tested in the LASSO laboratory of the Jyväskylä University of Applied Sciences. As a host computer, a Fujitsu Siemens Pa1538 laptop was used with attached Omnikey CardMan 5321 RFID card reader. One Nokia 6131 NFC phone was used with unlocked secure element. The unlocking process was performed by ToP Tunniste Oy (http://www.toptunniste.fi). Three additional NXP JCOP41 Java Cards were used for testing and development purposes.

After installing the card reader and the Java runtime environment, the AuthApplet was installed to the phone and two of the smart cards. One smart card was left for testing the error reporting functionality of the AuthDemo application.

The following test cases were performed successfully:

- Installing the AuthApplet using the bash scripts to the phone secure element and to the smart cards.
- Importing the public keys from the cards to the keyring file using the KeyringManager software.
- Changing the default PINs of the applets using the KeyringManager software.
- Successfully authenticating using the phone and using the smart cards.
- Failed authentication using improper PIN.
- Failed authentication using improper card. (When the NFC phone was to be used, another smart card was used instead.)
- Failed authentication using empty card.
- Impatient user: successful authentication takes around 1-2 seconds to be performed. If the user does not hold the card or the phone near the reader for this amount time, the authentication fails.

5 RESULTS

After finishing the project it can be clearly stated that the project was successful, and the main goals of the project were met. The results of this six-month research is summarized in this chapter.

The features of the Nokia 6131 NFC mobile phone were investigated and the possible ways for communication were described.

One purpose of the research and writing the thesis was to describe the full development process of an NFC application which involves communication between the Nokia 6131 NFC phone secure element, and a smart card reader connected to a PC. This requirement was fulfilled in the chapter 3. The Java Card Technology was introduced, and the whole process of the smart card application development was described from designing and developing the Java Card Applet, to the development of the reader-side application. The thesis can be used as a starting point by everyone planning to develop Java Card applets for the Nokia 6131 NFC Secure Element or any other Java Card 2.2.1 compatible smart card.

Before this project, there was no research done on the topic of strong multi factor authentication for the LASSO project before. The thesis provided valuable information on the topic, emphasizing the strengths compared to the password authentication. Different types of strong authentication was presented, and these protocols were evaluated for various security aspects and vulnerabilities.

The main result of the thesis was to provide proof that the Nokia 6131 NFC phone can be used in strong authentication systems. One possible strong authentication system was developed using Java technology to authenticate users to a stand-alone PC running a secured application. Lack of time and resources did not allow to extend the concept of the authentication from stand-alone PC to a networked environment. With minor modifications of the protocol, and by introducing *Off-line delivery service fort certificates* (described in 2.1.7), the protocol is capable for usage in a network.

6 DISCUSSION

The aim of the project was to merely pilot the possibility to use the Nokia 6131 NFC in a strong authentication system. As the requirements were very brief, extensive research had to be performed, especially in the area of computer security. As lots of information had to be evaluated during the process, it was a perfect opportunity to deepen my knowledge in many different areas of science and technology. Describing all the knowledge gained would fill many hundreds of pages, so much of it is missing from this thesis. Hence, this thesis is mostly just the summary of the straight path from the requirements to the final product, omiting most of the dead ends.

As the Nokia 6131 NFC phone was released in the beginning of 2007, it can be considered a new product, hence there is not any detailed description of developing applications for this device. This thesis can be a starting point for beginner Java Card developers considering this platform. By using the chapter 3, they can find all the necessary tools and references needed to start developing Java Card applications.

As much of the project time was spent on research, only the initial project goal was met, which involved the authentication to a stand-alone system, using only one authentication protocol. Future research can be done in various fields involving the following areas, to develop a more advanced strong multifactor authentication solution. If the requirements had been more detailed, and the development platform had not been so new, more detailed research would have been done in some of the following fields.

Future research areas:

- The possible use of the phone Java ME features. Can the phone keyboard be used to enter the PIN, or can the phone be used to configure a more advanced applet stored in the phone secure element. (Access control, displaying the name of the application the user is about to authenticate, etc.)
- Extending the authentication protocol for usage in a networked environment.
- Researching the possibilities of integrating the Nokia 6131 NFC phone with security solutions which already exist on the market.
- Investigating the possibilities of other, more secure security protocols like Elliptic

Curve cryptosystems, or Zero Knowledge proofs. Key question in this area is the efficiency of implementing these algorithms on handheld devices.

It is obvious that the field of Near Field Technology is a topic which worth research, and one thesis cannot cover all of the areas. This thesis can be treated as a foundation for further research, as it introduces a possible hardware and software platform which can serve as the basis of a future research.

Appendix 1. AuthApplet

```
/*
 * LASSO Strong Authentication project
 * Copyright 2008 Sandor Juhasz
 *
 * AuthApplet.java
 */

package hu.juhasz.sandor.javacard.auth;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.OwnerPIN;
import javacard.framework.Util;
import javacard.security.KeyPair;
import javacard.security.KeyBuilder;
import javacard.security.MessageDigest;
import javacard.security.RandomData;
import javacard.security.RSAPublicKey;
import javacardx.crypto.Cipher;

/**
 * Java card applet which can be used in the stron authentication process
 * defined by the LASSO Strong authentication project. See the thesis for
 * further design considerations and specifications.
 */
public class AuthApplet extends Applet {

    /*
     * The instructions supported by the cardlet.
     */
    private static final byte INS_GET_PUBLIC_KEY = (byte)0x10;
    private static final byte INS_AUTH           = (byte)0x20;
    private static final byte INS_SEND_PIN       = (byte)0x30;
    private static final byte INS_SET_PIN        = (byte)0x50;

    /*
     * Using the default CLA, 0x80.
     */
    private static final byte CLA_AUTH   = (byte) 0x80;

    /**
     * The size of the PIN
     */
    private static final byte PIN_SZ           = (byte)0x04;

    /**
     * The R1 random string size in bytes.
     */
    private static final byte AUTH_R1_SZ       = (byte)64;

    /**
     * The size of the authentication response. This is 192 bytes
```

```
     * if 1024 bit authentication is used. (64 byte for the R1, and
     * 128 byte for the signature.
     */
    private static final byte AUTH_RESPONSE_SZ = (byte)192;

    /**
     * Error code for PIN verification failure.
     */
    private static final short SW_PINVERIFY_FAILED = (short)0x6900;

    /**
     * Error returned when an PIN authenticated operation is called
     * without previous PIN verification.
     */
    private static final short SW_AUTH_REQUIRED    = (short)0x6901;

    private KeyPair authKeyPair;
    private RandomData rand;
    private Cipher rsa;
    private MessageDigest sha1;
    private OwnerPIN ownerPin;

    /**
     * Creates an applet instance.
     * Generates a new keypair, sets up the PIN with the default value of
     * '9999', creates the cipher, the message digest, and the randum number
     * generator.
     */
    private AuthApplet() {
        authKeyPair = new KeyPair(KeyPair.ALG_RSA_CRT,
                                  KeyBuilder.LENGTH_RSA_1024);
        authKeyPair.genKeyPair();

        ownerPin = new OwnerPIN((byte)3, (byte)4);
        ownerPin.resetAndUnblock();
        ownerPin.update(new byte[] { 0x39, 0x39, 0x39, 0x39 },
                        (short)0, (byte)4);
        rsa = Cipher.getInstance(Cipher.ALG_RSA_PKCS1, false);
        sha1 = MessageDigest.getInstance(MessageDigest.ALG_SHA, false);

        rand = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);


        register();
    }

    public static void install(byte[] args, short offset, byte len) {
        new AuthApplet();
    }

    /**
     * Called when the cardlet is selected. If the PIN counter is more than
     * the allowed, it refueses to select the applet.
     */
    public boolean select() {
        if (ownerPin.getTriesRemaining() == 0)
            return false;
        return true;
    }

    public void deselect() {
        ownerPin.reset();
    }

    public void process(APDU apdu) {
        byte[] buffer = apdu.getBuffer();

        if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
            (buffer[ISO7816.OFFSET_INS] == (byte)(0xA4))) {
            return;
```

```
        }

        if (buffer[ISO7816.OFFSET_CLA] != CLA_AUTH)
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

        switch (buffer[ISO7816.OFFSET_INS]) {
        case INS_GET_PUBLIC_KEY:
            getPublicKey(apdu);
            break;
        case INS_AUTH:
            auth(apdu);
            break;
        case INS_SEND_PIN:
            sendPin(apdu);
            break;
        case INS_SET_PIN:
            setPin(apdu);
            break;
        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }

    private void getPublicKey(APDU apdu) {
        if (!ownerPin.isValidated())
            ISOException.throwIt(SW_AUTH_REQUIRED);

        byte[] buf = apdu.getBuffer();
        byte[] temp = new byte[3];
        byte[] temp2 = new byte[192];

        short le = apdu.setOutgoing();
        RSAPublicKey publicKey = (RSAPublicKey)authKeyPair.getPublic();

        short expSize = publicKey.getExponent(temp, (short)0);
        short modulusSize = publicKey.getModulus(temp2, (short)0);

        short messageSize = (short)(expSize + modulusSize + 2);
        apdu.setOutgoingLength(messageSize);

        buf[0] = (byte)expSize;
        Util.arrayCopy(temp, (short)0, buf, (short)1, expSize);
        buf[(short)(expSize + 1)] = (byte)modulusSize;
        Util.arrayCopy(temp2,(short)0, buf, (short)(expSize + 2),
                       modulusSize);

        apdu.sendBytes((short)0, messageSize);
    }

    private void auth(APDU apdu) {
        if (!ownerPin.isValidated())
            ISOException.throwIt(SW_AUTH_REQUIRED);

        byte[] buf = apdu.getBuffer();

        short lc = apdu.setIncomingAndReceive();

        if (lc != AUTH_R1_SZ)
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

        sha1.reset();
        sha1.update(buf, ISO7816.OFFSET_CDATA, (byte)64);

        // Generate R2, and let's put into the message digest generator.
        // The result goes to the digest array for the signing.
        byte[] R2 = new byte[64];
        byte[] digest = new byte[20];
        short digestLength;
        rand.generateData(R2, (short)0, (short)64);
        digestLength = sha1.doFinal(R2, (short)0, (byte)64, digest, (short)0);
```

```
        // Let's change to outgoing mode.
        short le = apdu.setOutgoing();
        //      if (le != AUTH_RESPONSE_SZ)
        //          ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);


        // Create the signature.
        byte[] signature = new byte[256];
        rsa.init(authKeyPair.getPrivate(), Cipher.MODE_ENCRYPT);
        short sigLength = rsa.doFinal(digest, (short)0, (short)20, signature,
                                      (short)0);

        apdu.setOutgoingLength((short)192);

        // Copy the R2 first, then copy the signature.
        Util.arrayCopy(R2, (short)0, buf, (short)0, (short)64);
        Util.arrayCopy(signature, (short)0, buf, (short)64, (short)sigLength);

        // Send the data.
        apdu.sendBytes((short)0, (short)192);

    }

    private void sendPin(APDU apdu) {
        byte[] buf = apdu.getBuffer();

        short lc = apdu.setIncomingAndReceive();
        if (lc != PIN_SZ)
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

        if (!ownerPin.check(buf, ISO7816.OFFSET_CDATA, (byte)4))
            ISOException.throwIt(SW_PINVERIFY_FAILED);
    }


    private void setPin(APDU apdu) {
        if (!ownerPin.isValidated())
            ISOException.throwIt(SW_AUTH_REQUIRED);

        byte[] buf = apdu.getBuffer();

        short lc = apdu.setIncomingAndReceive();
        if (lc != PIN_SZ)
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

        ownerPin.update(buf, ISO7816.OFFSET_CDATA, (byte)4);
    }
}
```

Appendix 2. Terminology

APDU	Application Protocol Data Unit The application level unit of communication between a smart card and a card acceptance device.
CAD	Card Acceptance Device An electronic device that reads and communicates with smart cards.
JCDK	Java Card Development Kit A software development kit provided by Sun Microsystems Inc. to develop Java Card applets.
JCRE	Java Card Runtime Environment The Java Card Runtime Environment speci cation describes the additional services the JCVM should provide to ensure security and to optimize memory consumption.
JCVM	Java Card Virtual Machine A specialized Java Virtual Machine for use in smart cards.
JCWDE	Java Card Workstation Development Environment An application included in the JCDK which allows the simulated execution of a Java Card applet as if it was masked in ROM.
NFC	Near Field Communication A contactless communication protocol which enables mobile phones to communicate with wireless smart card readers, and to read RFID tags.
PIN	Personal Identifier Number A secret numeric password shared between a user and a system that can be used to authenticate the user to the system.
RFID	Radio Frequency Identification An automatic identification method, relying on storing and remotely retrieving data using devices called RFID tags or transponders.

Bibliography

Menzes et al., 1996. Handbook of Applied Cryptography. CRC-Press.

Chen, Z., Giorgo, R., 1998. Understanding Java Card 2.0. Referred to on March 17, 2008. http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev.html, JavaWorld.

Ort E., 2001a. Developing a Java Card Applet. Referred to on March 17, 2008. http://developers.sun.com/mobility/javacard/articles/applet/, Sun Developer Network.

Ort E., 2001b. Writing a Java Card Applet. Referred to on March 17, 2008. http://developers.sun.com/mobility/javacard/articles/intro/, Sun Developer Network.

Bishop, M., 2002. Computer Security: Art and Science. Addison-Wesley Professional.

Developer Kit User's Guide. 2003. Online Documentation, Java Card Developer Kit 2.2.1. Sun Microsystems Inc.

Kaliski, B., 2003. TWIRL and RSA Key Size. Referred to on March 17, 2008. http://www.rsa.com/rsalabs/node.asp?id=2004, RSA Laboratories.

Oritz, C. E., 2003a. An Introduction to Java Card Technology - Part 1. Referred to on March 17, 2008. http://developers.sun.com/mobility/javacard/articles/javacard1/, Sun Developer Network.

Oritz, C. E., 2003b. An Introduction to Java Card Technology - Part 2, The Java Card Applet. Referred to on March 17, 2008. http://developers.sun.com/mobility/javacard/articles/javacard2/, Sun Developer Network.

Oritz, C. E., 2003c. An Introduction to Java Card Technology - Part 3, The Java Card Host Application. Referred to on March 17, 2008. http://developers.sun.com/mobility/javacard/articles/javacard3/, Sun

Developer Network.

Hook, D., 2005. Beginning Cryptography with Java. Wronx Press.

Java Smart Card I/O API reference. 2006. Referred to on March 17, 2008. `http://java.sun.com/javase/6/docs/jre/api/security/smartcardio/spec/index.html`, Sun Microsystems Inc.