CRYPTOGRAPHIC SYSTEM FOR SUPPLY CHAINS OVER RFID IMPLEMENTATION

Daniel Moreno Rosselló

January 2008, Politecnico di Torino

I would like to thank Maurizio Rebaudengo and Filippo Gandino, at the Politecnico di Torino, for their support along the realization of the project, giving all the facilities to carry out.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

RFID is an identication technology that has a wide range of applications. One of the main areas of interest is the Supply Chain Management (SCM), traceability management, multimodality shopping, etc. In suppy chains readers are placed into manufacturing facilities, the supplier attaches RFID tags to all outgoing products and some data is stored on each tag , attached to the products. At each stage, every player in the supply chain will be able to track relevant information on the product. The information can be used to ensure quality, on time delivery, fast re-stocks, ad hence greater overall efficiency.

NSCCA (Nested Supply Chain Cryprographic System) is an algorithm derived from the Nested Cryptographic Algorithm, that aims to ensure authenticity, privacy and protection along one suppy chain, and is based on public cryptography.

The scope of this project is to develop a programming class library and an application that implement the NSCCA algorithm, as well as testing its performance, as well as an interface that permit to communicate through an RFID reader in order to save and load the result of the algorithm in RFID tags.

structure of this report is as follows: In Chapter 2, some important concepts involved in the project are explained, as well as the NSSCA algorithm, public key cryptography and RFID technology. Then, in Chapter 3 the development framework (programming language, ID) is explained, and the program structure is defined. Continuing, the Chapter 4some tests on the platform are performed, and the main results are discussed. After that, in Chapter 5 there is a little user manual for the application developed to test the libraries. Finally, in Chapter 6 the conclussions on the project realization are explained.

Chapter 2

PREVIOUS CONCEPTS

In the a secure pervasive information system (IS) is described, by employing RFID technology and the NSCCA (Nested Supply Chain Cryptographic Algorithm) algorithm to protect the data.

2.1 NSCCA

The NSCCA is an algorithm, derived from the Nested Cryptographic algorithm, based un public key cryptography, that uses a set of nested cypherytexts to protect the data, with small memory areas. Its structure is shown in Figure 2.1 on page 9. The memory tags are divided into Memory Slots (MS) of a certain length, where the cyphertexts generated with the algorithm will be written.

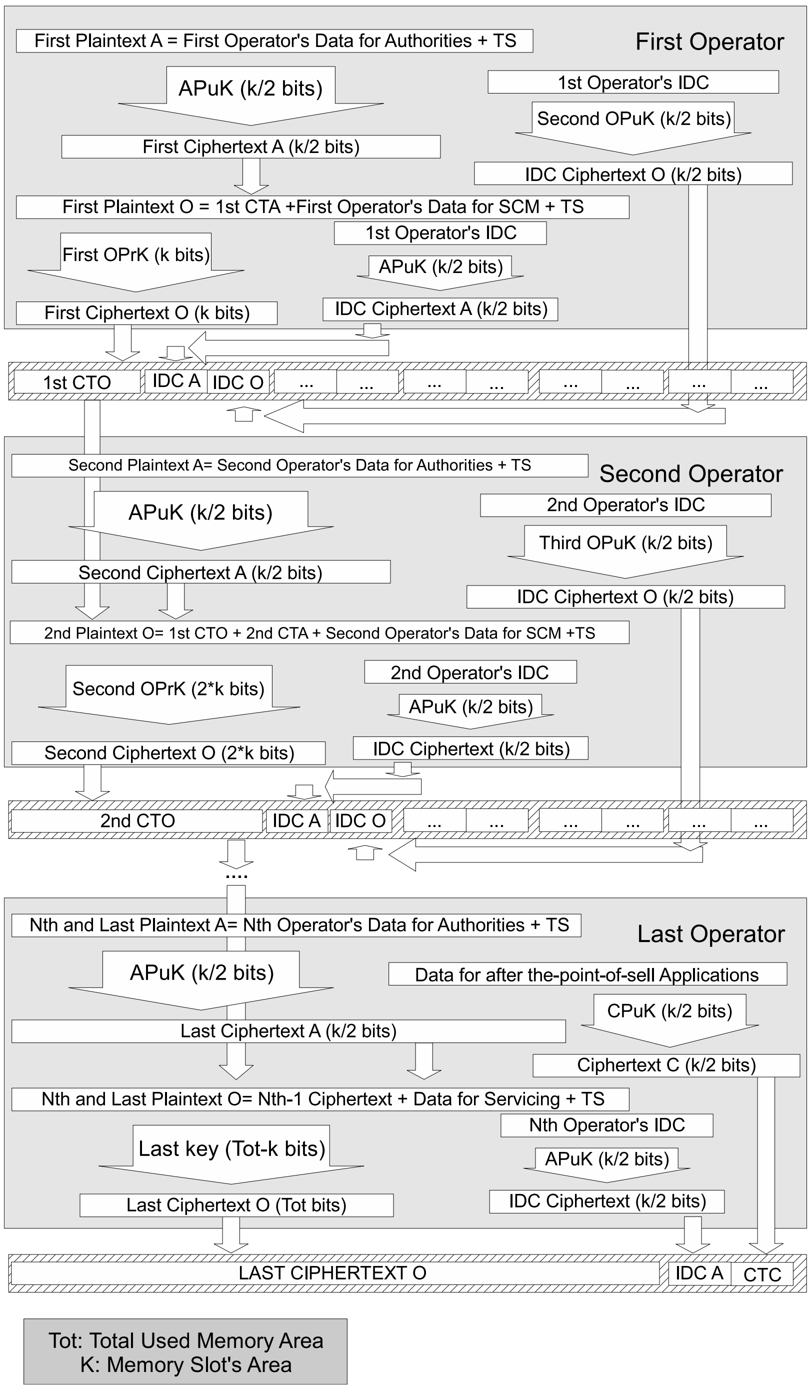
There are different set of keys involved with the algorithm:

- The *Competent Authority* (CA) owns 1 key set. The private (APrK) who only the CA holds, and the public key, which is public to all the Chain Members (CM). Both have a length of $\frac{1}{2}$ MS.
- Every CM owns 2 key sets, one ($\frac{1}{2}$ MS length) for IDCO encryption (a field with the identification code for the company), and one for the data encryption itself. This last set has a length of $i \cdot \frac{1}{2}$ MS, there i is the position of the CM in the chain. As the position is higher, the keys are longer, and so the complexity of the encryption processes.
- Also the customer owns 1 key set ($\frac{1}{2}$ MS length), for after the-point-of-sell applications.

Each CM inserts some information about the target product:

- Some data is for the *Competent Authority (CA)*, and is encrypted with the CA Public key, so that only the certification authority, with his own private key, is able to decrypt it and get the message. That allows the traceability management.

- Another data is for some *permitted members of the SCM*, and is encrypted with the writing company's private key. The corresponding public key is distributed only to all the members of the chain that have the access permission, and only they can decrypt the data and get the message. That allows the supply chain management functions.



First Operator
First Plaintext A = First Operator's Data for Authorities + TS
APuK (k/2 bits)
1st Operator's IDC
Second OPuK (k/2 bits)
First Ciphertext A (k/2 bits)
IDC Ciphertext O (k/2 bits)
First Plaintext O = 1st CTA +First Operator's Data for SCM + TS
1st Operator's IDC
First OPrK (k bits)
APuK (k/2 bits)
First Ciphertext O (k bits)
IDC Ciphertext A (k/2 bits)
1st CTO
IDC A
IDC O
...
Second Operator
Second Plaintext A= Second Operator's Data for Authorities + TS
APuK (k/2 bits)
2nd Operator's IDC
Third OPuK (k/2 bits)
Second Ciphertext A (k/2 bits)
IDC Ciphertext O (k/2 bits)
2nd Plaintext O= 1st CTO + 2nd CTA + Second Operator's Data for SCM +TS
2nd Operator's IDC
Second OPrK (2*k bits)
APuK (k/2 bits)
Second Ciphertext O (2*k bits)
IDC Ciphertext (k/2 bits)
2nd CTO
IDC A
IDC O
....
Last Operator
Nth and Last Plaintext A= Nth Operator's Data for Authorities + TS
APuK (k/2 bits)
Data for after the-point-of-sell Applications
CPuK (k/2 bits)
Last Ciphertext A (k/2 bits)
Ciphertext C (k/2 bits)
Nth and Last Plaintext O= Nth-1 Ciphertext + Data for Servicing + TS
Nth Operator's IDC
Last key (Tot-k bits)
APuK (k/2 bits)
Last Ciphertext O (Tot bits)
IDC Ciphertext (k/2 bits)
LAST CIPHERTEXT O
IDC A
CTC
Tot: Total Used Memory Area
K: Memory Slot's Area

Figure 2.1: NSCCA Algorithm

2.2 Public key cryptography

As explained in the previous section, NSCCA algorithm is based on public key cryptography. Public key cryptography (also known as asymmetric cryptography) uses two separate keys to exchange data, what differentiates it from symmetric cryptography. A public key Cryptography system is so constructed that calculation of one key (the 'private key') is computationally infeasible from the other (the 'public key'), even though they are necessarily related. Instead, both keys are generated secretly, as an interrelated pair.

One key is used to encrypt or digitally sign the data, and the other key is used to decrypt the data or verify the digital signature. These keys are often referred to as public/private key combinations. If an individuals public key (which can be shared with others) is used to encrypt data, then only that same individuals private key (which is known only to the individual) can be used to decrypt the data. If an individuals private key is used to digitally sign data, then only that same individuals public key can be used to verify the digital signature. Common algorithms that implement asymmetric cryptography include RSA, Digital Signature Algorithm (DSA), and Elliptic Curve DSA (ECDSA).

2.2.1 RSA

This RSA algorithm, named after the three inventors Ron Rivest, Adi Shamir and Leonard Adleman. This is an asymmetric cypher and as such is able to be used for public key cryptography. In simple terms you can send Bob a message encrypted with his public key and when he has received it he can decrypt it with his private key (from which his public key was derived).

The security of this system is based on the difficulty in factoring large numbers. The private and public keys can be functions of large prime numbers. While the process is known, recovering the plaintext from the public key is considered to be the equivalent to factoring the product of the two prime numbers. With large numbers this is considered a major computational task, even by to-days standards, and is believed to be, in terms of time, beyond the capability of any existing technique/computer combination.

The keys are generated given 2 large prime numbers p and q, and following some formulas:

$$n = pq \tag{2.1}$$

$$\Phi(n) = (p-1)(q-1) \tag{2.2}$$

$$d \cdot e = 1(mod\Phi(n)) \tag{2.3}$$

While calculating this parameters, we obtain the corresponding private and public keys:

- Public key corresponds to the comnbination of (e, n)

- Private key corresponds to (d, n)

The encryption and decryption processes are performed by calculating:

- Encryption of a message m: $c = m^d mod(n)$, where c is the cyphered message
- Decryption of a cyphered text c: $m = c^e mod(n)$, where m is the original message

PKCS 1.5 With development of cryptography methods software companies realized the importance of standards to define how to deal with data in secure and standard manner. Microsystem, Microsoft, Applet and others joined this process in RSA and together they defined PKCS standards (Public Key Cryptography Standards). The most important for this project is PKCS#1" The RSA encryption standard". It defines mechanisms for encrypting and signing data using RSA system. In this project PKCS 1.5 format is used. The encryption process consists on 4 basic steps:

1. Encryption-block formatting, to obtain the Encrypted Bytes
2. Octet-string-to-integer conversion, to obtain "x"
3. RSA computation
4. Integer-to-octet-string conversion, to obtain the Encrypted Data Octets (ED)

The decryption process consists of the same four steps as the encryption but reversed. The format of the encryption block in this standard is shown in Figure 2.2 on page 11.

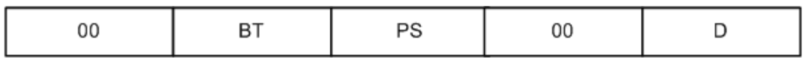


Figure 2.2: PKCS 1.5 frame structure

Where:

- BT is the type (02 for Public-key)
- PS is the Padding String
- D is de Data

The padding String consists of k-3-||D|| octets.

Input Data size limits One important aspect is how much information can we give to encrypt. It is determined by the length of the modulus, using the (2.4).

$$D < K - 11 \tag{2.4}$$

Where D is the Data length, and K the length of the modulus, both in bytes. The resulting length is always positive, because k (the length of the modulus) is always at least 12 octets. This limitation guarantees that the length of the padding string PS is at least eight octets, which is a security condition.

For example, for a 512-Bit Key, corresponding to 64 octets, we can only encrypt a maximum of 53 octets. For 1024-Bit key, 117 octets, and so on.

2.3 RFID

RFID is the technology used to write and read the NSCCA data into the memory tags. In the next sections this technology is introduced

2.3.1 Overview

RFID stands for Radio-frequency identification, is an identification method, relying on storing and remotely retrieving data on devices called RFID tags or transponders.Invented in 1948 bu Harry Stockman, and begin commercial use in 1990s. On RFID systems, the power supply to the data-carrying device and the data exchange between the data-carrying device and the reader are achieved using magnetic or electromagnetic fields.

An RFID system is made up of two components:

- Transponder, which is located on the object to be identified
- Reader, which, depending on the technology chosen, may read or read/write the transponder

2.3.2 RFID Systems

There are many different RFID systems that use different frequency bands and coupling techniques. Depending on their characteristics, different reading ranges, operations (write, write/read) and data rates can be achieved. Also around this technologies, several standards have been defined. The Table 2.2 on page 13 sumarizes them.

	LF	HF	UHF	Microwave
Freq. Range	125 - 134KHz	13.56 MHz	866 - 915MHz	2.45 - 5.8 GHz
Read Range	10 cm	1M	2-7 M	1M
Coupling	Magnetic	Magnetic	Electro magnetic	Electro magnetic
standards	11784/85, 14223	18000-3.1, 15693,14443 A, B, and C	EPC C0, C1, C1G2, 18000-6	18000-4

Table 2.2: RFID Technologies

This technologies differ in the way how the physical communication between the reader and the transponder is performed, the coupling method, depending on the frequency band.

- In the UHF (*ultra high frequency*) range (868 MHz and other frequencies), and in the *microwave* range (2.4 ... 2.5 GHz), the near-field is so small that the reader must use the far-field. The transponder may send its information actively back to the reader, or it operates as so called "backscatter" which reflects the received energy in a certain way. The transponder antenna modulates the reflected signal by changing its reflection characteristics between alignment and misalignment, optimally reflecting or deteriorating the reflected wave. Achievable read ranges are just a few metres.

- RFID readers for the *low frequency* (125 kHz, 134.2 kHz) and *high frequency* (13.56 MHz) ranges operate within the near-field with inductive coupling. The load modulation by the transponder is detected by the antenna of the reader. The typically achievable read ranges are about 0. . . 1 m.

The technology used is in this project the HF with a 13.56 MHz carrier frequency , and is represented in Figure 2.3 on page 13.

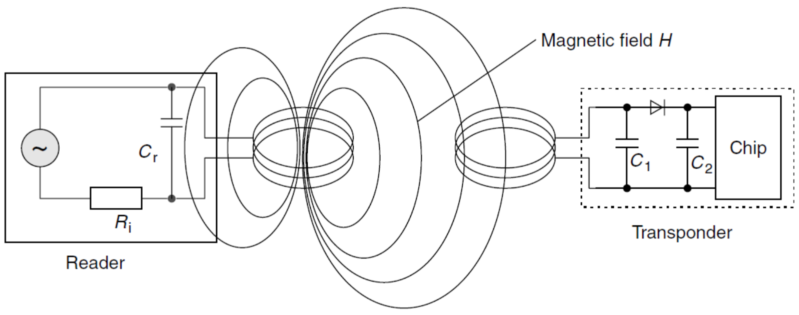


Figure 2.3: Inductive coupling

The main standard used for this application (and compliant with the equipment) is the *ISO/IEC 14443*, which consists of four parts and describes two types of cards: type A and type B. The main differences between these types concern modulation methods, coding schemes and protocol initialization procedures. Both type A and type B cards use the same transmission protocol, that specifies data block exchange and related mechanisms.

Chapter 3

DEVELOPING

This chapter describes the Programming framework, IDE and language chosen for the project, as well as it main characteristics. Also the main structure of the application is described, and how the application communicates with the RFID Reader.

3.1 Development platform

First election, after studying the functionalities that the libraries and application must have, has been choosing the programming platform used to develop it. Some important things have been considered, such as well RSA support, capable of running in smart devices (mobile), and compatible with the hardware used. This last require has been the most restrictive, as the API provided by the RFID Reader vendor is a DLL written in C++, and a wrapper to adapt the library for C# code use. Also the operative system used in the devices available, that was Windows Mobile running in the Pocket PCs available.

The main options were use *JavaME* (micro edition), widely adopted in most devices, and the *.NET Compact Framework* (from now .NET CF). Finally, the .NET Framework by Microsoft has been selected, compatible with the reader API and the Windows Mobile operative system, with RSA libraries implemented and well documented, and an intuitive development IDE as Visual Studio 2008.

3.1.1 .NET COMPACT FRAMEWORK

The Microsoft .NET Compact Framework (.NET CF) is a version of the .NET Framework that is designed to run on Windows CE based mobile/embedded devices such as PDAs, mobile phones, etc. The .NET CF uses some of the same class libraries as the full .NET Framework and also a few libraries designed specifically for mobile devices.

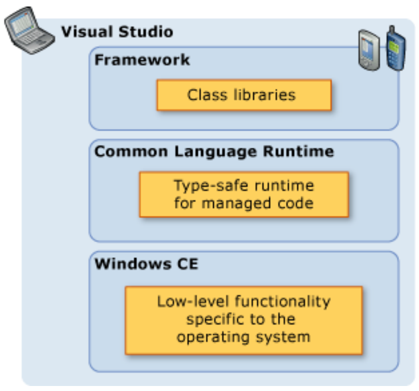

Visual Studio
Framework
Class libraries
Common Language Runtime
Type-safe runtime for managed code
Windows CE
Low-level functionality specific to the operating system

Figure 3.1: .NET Compact Framework architecure

The runtime is the foundation of the .NET Compact Framework. It is responsible for managing code at execution time, providing core services such as memory management and thread management while enforcing code safety and accuracy. Code that targets the runtime is known as managed code;

The .NET Compact Framework class library is a collection of reusable classes that you can use to quickly and easily develop applications.The .NET Compact Framework also implements a subset of the System.Windows.Forms and System.Drawing classes, which allows to construct a rich user interface for a device application.

- .NET Compact Framework 2.0
- .NET Developer Toys
- ActiveSync

3.1.2 Visual Studio .NET

Visual Studio is an Integrated Development Environment (IDE) that combines a number of features, such as en editor, debugger and forms designers.

In addition to all of the features found natively in Visual Studio .NET, there are the following device-specific features:

- **Device emulators:** Testing environments that simulate specific devices. Emulators run on the developer's PC, allowing for testing without the presence of a device.
- **Automatic deployment of applications:** allows you to easily test to either an emulator or a device, providing developers with a seamless testing environment.
- **Remote debugging:** allows you to leverage the debugging tools offered through the Visual Studio .NET IDE with your device applications. All of the debugging tools can be used with .NET Compact Framework-based applications running either in an emulator or on a device.

The .NET Compact Framework supports two development languages: C# .NET and Visual Basic .NET. While previous versions of Windows CE development tools favored C-based languages—namely eMbedded Visual C++—with the .NET Compact Framework it makes little difference which of the languages you choose, because both are equally powerful and functional.

here is another language limitation. Under the .NET Framework you can use mixed-language components within a single project. In .NET Compact Framework projects are restricted to a single language, either C# .NET or Visual Basic .NET. The workaround to this single-language project limitation imposed by .NET Compact Framework is to create additional projects using the Class template. Add your alternate language code to the template, and then simply add references to these classes in your application project.

3.2 Program structure

The whole program structure is represented in Figure 3.2 on page 17. The main components developed are the Class that implements the NSCCA algorithm and the one for the interaction with the reader. A Graphical User Interface (GUI) was also developed, in order to interact with all the functions through some buttons and text inputs that permit introduce data and visualize the results in an easy way.

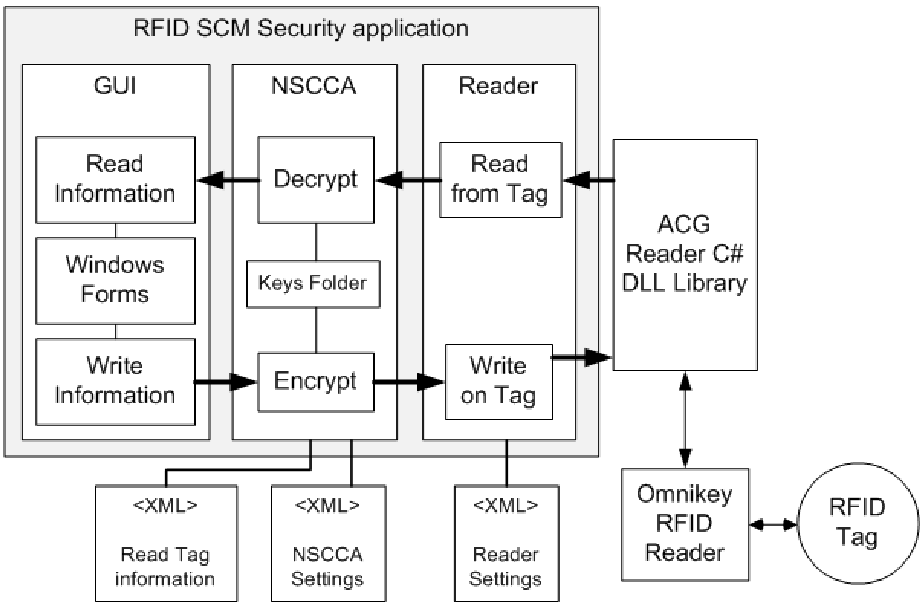


Figure 3.2: RFID SCM Security Application structure

The program is structure in different parts. There are two main classes:

- *ACGReaderInterface:* Implements all the methods needed to perform all the reading and writing functions with the RFID Reader, by also using a DLL, a library made by the Reader manufacturer that provides useful functions programmed in C++ to interact with the reader. As C# is the native language fot the .NET Framework, also a wrapper is provided to adapt the C++ library to C#. This class is based on one written by Erwing Sanchez for another project, and adapted for the project purposes.

- *NSCCA:* Is the core library for this project, and implements all the functionalities to perform the NSCCA Algorithm, through the use of the RSA implementation of the .NET Framework.

Also 4 main structures have been defined to exchain data between classes and save important parameters:

- *NDFrame and NEFrame:* This structures are used to save and load NSCCA decryption and encryption fields, such as CTA, CTO, IDC, etc, while performing the decryption and encryption functions.
- *ReaderParameters and NSCCAParameters:* These are structures that have the important parameters used in the respective classes, and are also used to exchange parameteres between the Forms and the classes.

3.2.1 NSCCA Implementation

3.2.1.1 Create Cryptosystem

The program provides a method that given as inputs a Minimum Key length and the Tag available memory, calculates the number of companies that can be part of the chain.

Also the user must select each company that will be in each position of the chain, and the method creates all the Key pairs of different sizes, to fit the algorithm needs, and saves all the generated keys to binary files, one for the public key, and another for the private key.

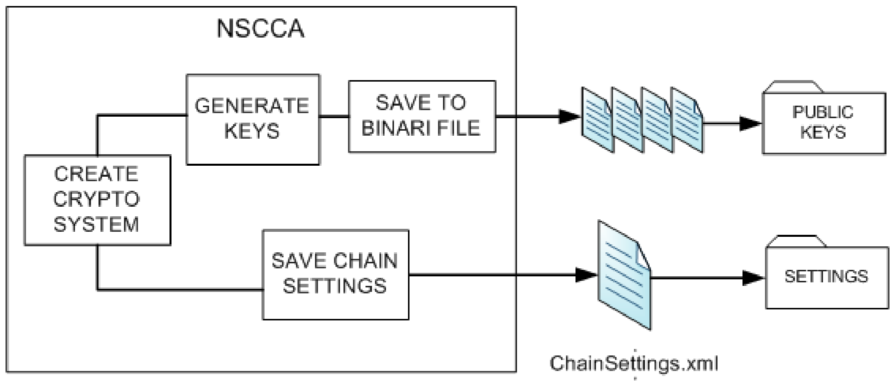


Figure 3.3: NSCCA Cryptosystem generation

Once the Crytosystem and all the components that comprises it are defined, and XML file is generated with all the parameters needed, stores them in XML files, inside a folder.. By doing this, the configuration settings for the NSCCA class can be saved and loaded as needed, and retrieved when the application is started. The XML file format is defined in Figure 3.4 on page 19.

```
<chain>
    <company>
        <cname>FirstComanyID</cname>
        <keysize>1024</keysize>
        <Pos>1</Pos>
    </company>
    <nextcomp>
        <cname>ACompany</cname>
        <cname>AnotherCompany</cname>
        <cname>OnemoreCompany</cname>
    </nextcomp>
</chain>
```

Figure 3.4: Chain XML file structure

The fields are almost save explaining:

- In <company>: *cname* (identification code of the company), *keysize* ()and *pos* (position in the chain), refer to the actual company.
- In <nextcomp>: The fields *cname* have the same meaning, but refer to the possible next companies in the chain that can be selected.

3.2.1.2 NSCCA Encrypt / Write

3.2.1.3 Memory Tag Formatting

In the 3.2is represented the memory structure of the SRI4XK Tag. It is formatted in 128 blocks of 32 Bit each. The first 7 positions are for OTP (One-time programable) and counters, and the next 7 can be locked. The rest is available to store data, there are 114 available slots of EEPROM memory, that make 3904 Bit.

ADDRESS	USE
07 to 0F	LOCKABLE EEPROM
00 to 06	OTP & COUNTER
10 to 7F	EEPROM

Table 3.2: Tag memory structure

4 KBit Memory Tag

If we chose a minimum key size of 512 Bit (k/2), only 2 companies would be permitted in the chain. At the exit of the first stage, the first company will write 2048 Bit (2*k), and at the exit of the second company, it will write 3072 Bit (2*k) on the memory. So, using a typical 4 Kbit RFID Tag, the number of companies allowed in the security chain would be as follows:

Minimum Key Length	Companies
384	3
512	2
1024	1
1952	1

Table 3.3: Minimum key length vs companies

3.2.1.4 NSCCA Read / Decrypt

Tag Information file saving One functionality offered is the possibility to save the Tag information read previously to an XML, which will have by name the corresponding Tag ID and extension XML. That is an interesting options in case of future threatment of the information obtained by other applications, as XML is an extended format for structuring and exchanging information. The information saved would have the structure showed in the next figure:

```
<tag>
   <lastread>Wed, 13 Apr 2005 15:24:09 GMT<lastread>
   <id>D0020CB3E3C6390E</id>
   <lastcomp>ITAL23412345678912</lastcomp>
   <smcinfo>SMCData</smcinfo>
   <Previnfo>5F0DB7A0B291B813A4D7A1DC90276B
   106EC8E1FF0F2B2C345C8DA4BEC9941EDC2E2107
   7D2463C6</Previnfo>
</tag>
```

Figure 3.5: Tag information XML File

3.2.2 Reader Interface

The reader operations follow serveral steps. The sequence (shown in the Figure 3.6 on page 21) is:

1. Detect the serial port: Needed to detect which PC port is the RFID Reader connected to, in order to interact with it.
2. Open the the reader: Turns on the RF circuitry of the reader.

3. Send command and retrieve data. Either for reading or writing, the reader sends the selected commands to the tag, and waits for a response, if the operation was successful or, in the case of writing, it performs a reading of the written block and returns the value, so that the application can check if the writing was performed well.

4. Close reader: Closes the RF circuitry, and it's not possible to read or write data.

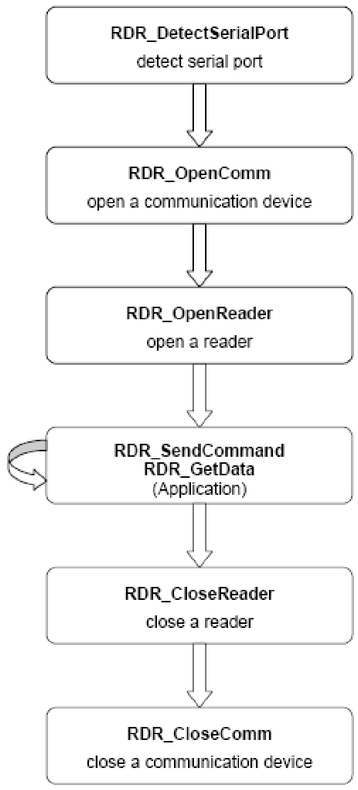


Figure 3.6: Tag Reading Sequence

The 3.2 shows how the memory is filled in every step of a 3-company chain, with a Memory Slot size of 1024 Bit.

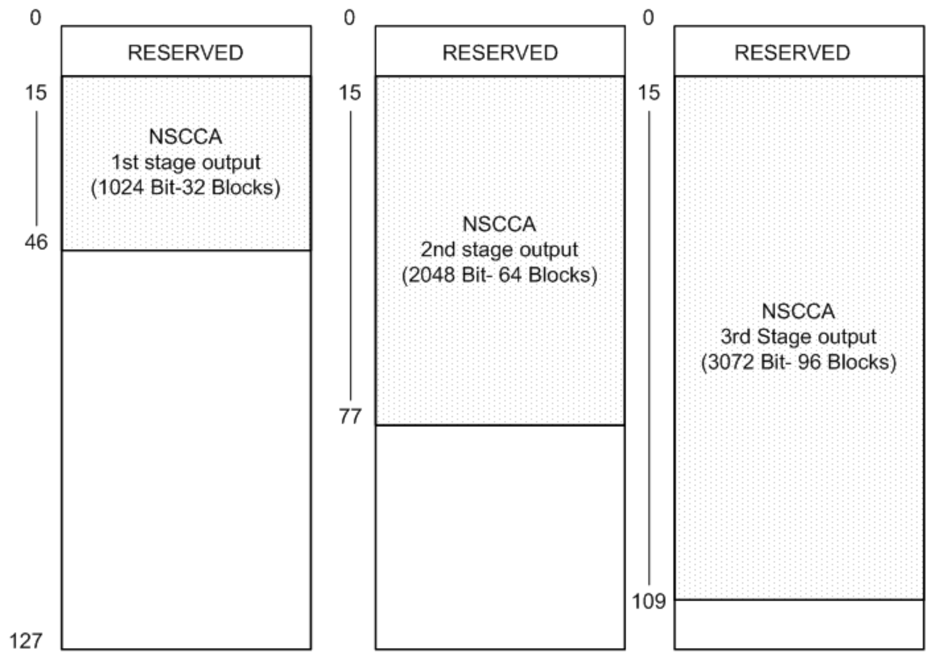


Figure 3.7: NSCCA memory allocation in Tag along the SCM

Chapter 4

TESTING AND RESULTS

4.1 Settings

The application scenario is shown in Figure 4.1 on page 23.The application was implemented in PocketPC from the PC using the Activesync program. The PocketPC used to test the performance was the HP iPAQ 4700, and the RFID tags the SRIX4K.

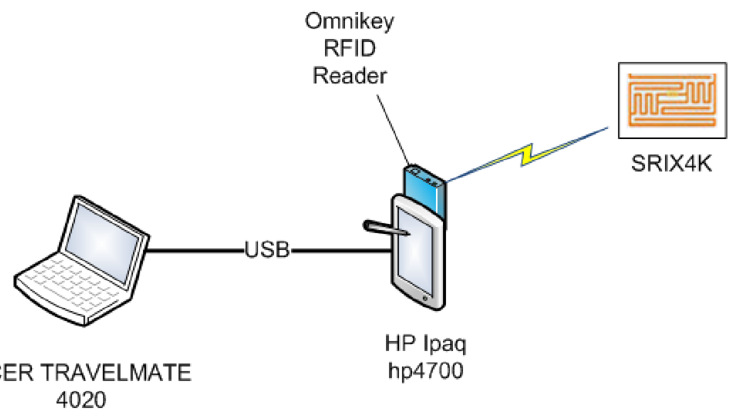


Figure 4.1: Testing scenario

4.1.1 HP IPAQ Pocket PC

- Processor Intel® PXA270 Processor 624 MHz
- Memory User Available Memory 192 MB total memory; (128 MB Strata Flash ROM and 64 MB SDRAM)
- Display Type 4" Resolution (W x H) 480 x 640 pixels
- SD Slot Support and CF Slot Type II

- Microsoft® Windows Mobile™ 2003
- WiFi and Bluetooth support

Figure 4.2: HP IPAQ PocketPC

4.1.2 Tag reader: OMNIKEY® 4543 E-ID Mobile

The RFID reader used to test the application is the OMNIKEY® 4543 E-ID, which can be connected to a PockecPC through a Compact Flash type II slot. Its main characteristics are:

Figure 4.3: Omnikey RFID Reader

- RF Transmit Frequency: 13.56MHz
- Supported Standards: ISO14443A, ISO14443B
- Communications Interface: CF Card Type II
- Communications Protocol: Specific ASCII or Binary Protocol

- Communications Parameter: 9600 Bit/s to 115kBit/s, 8, n, 1
- S/W Driver: Virtual COM port DLL (C++) & Reader Utility available for Pocket PC 2002 & 2003
- Reading distance: Up to 60 mm / 2,36 Inch, depending on tag
- RF Transmission speed: Up to 848 kBit/s

4.1.3 RFID Tags: SRIX4K

The tags used in this project are ST microelectronics SRIX4K model (see Figure 4.4 on page 25), which has this features:

- Anticollision and anti-clone functions
- ISO 14443-2 Type B air interface and ISO 14443-3 Type B frame format compliant
- 13.56 MHz carrier frequency
- 847 kHz subcarrier frequency
- 106 Kbit/second data transfer
- 8 bit Chip_ID based anticollision system
- 64-bit unique identifier
- 4096-bit EEPROM with write protect feature
- 1 million erase/write cycles and 40-year data retention

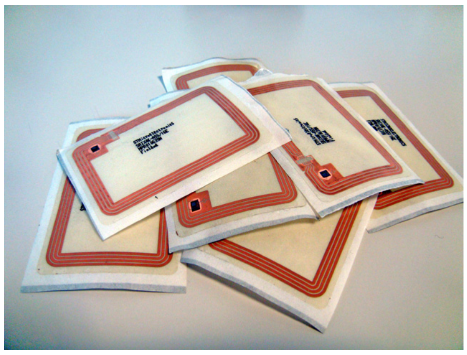


Figure 4.4: SRIX4K Tags

The reader is connected to the PocketPC via the Compact Flash Type II slot (Type II devices are 5 mm thick while Type I devices are 3.3 mm thick) . CompactFlash (CF) is a mass storage device format used in portable electronic devices.

4.2 Results

One defined the scenario, installing and configuring properly the application, some tests have been done over the platform to check that all components are working correctly, and check the performance time of the reader and the NSCCA functions.

4.2.1 Reader performance

Operation	Time
Open reader	3 s
Close reader	3 s
Select tag	21 ms / 4 Byte
Writing	37 ms / 4 Byte
Reading	2 s

Table 4.2: Reader operation timings

4.2.2 Encryption testing

This test compares the RSA encryption and decryption performing times for different key lengths, calculated on an HP IPAQ PocketPC and on a normal PC. It is important to remark that this test has been taken with key pairs generated with the .NET Framework RSA library. This implementation is made to put the the major part of complexity on the private key side, which is much longer than the public one.

The results are shown in the next figure, where we can see that the time spent increases exponentially with the key length. Till 2048-Bit key length we obtain acceptable times bellow 5 seconds, but with 3072-Bit the time spent with the PocketPC is about 20 seconds, and around 50 with a 4096-Bit key.

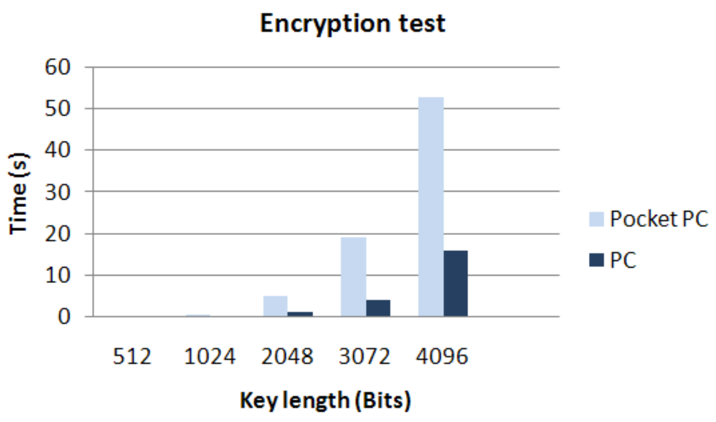


Figure 4.5: Encryption perform time

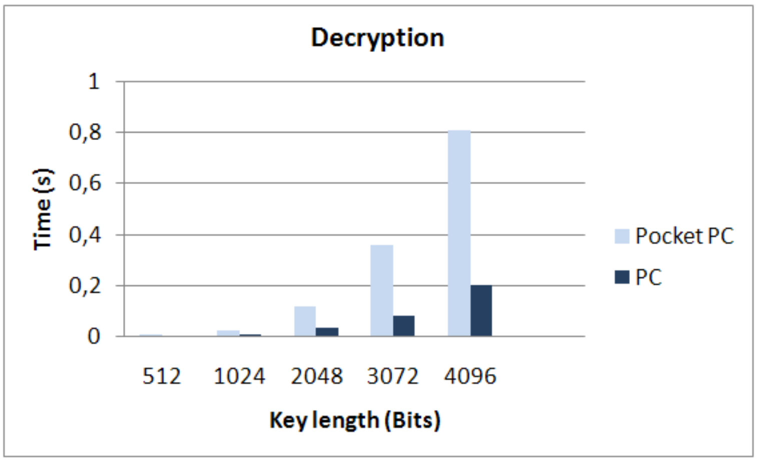


Figure 4.6: Decryption perform time

4.2.3 RFID functions testing

Some tests have been done over the platform in order to check the time spent to perform all the common operations with the RFID Reader, such as opening, closing the device, selecting a tag, or the most common, reading and writing information.

To do the next test, the fastest speed available has been selected, and the writing and reading time for different sizes have been measured. The main characteristic that can be observed is that writing in the Tag memory is slower than reading for every size. That is because while writing a block, a check is done by reading the same block writen.

The typical values for a 4-Byte memory slot are *21 ms* reading and *37 ms* writing time.

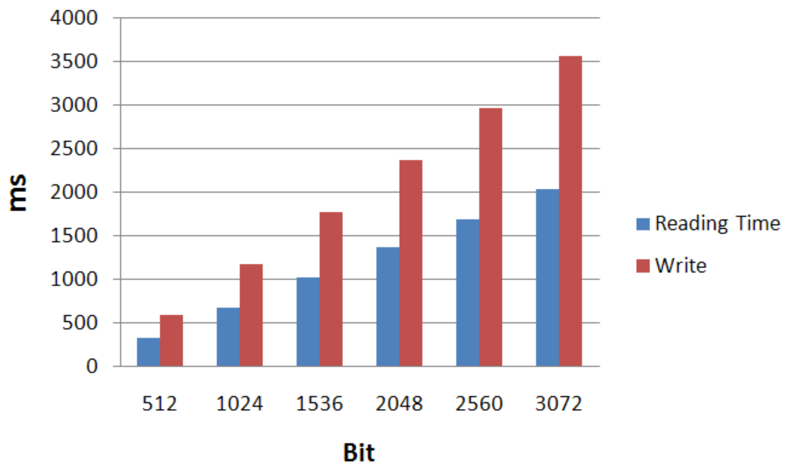


Figure 4.7: Reading vs writing time comparison

4.2.4 NSCCA Operations

Let's asume a chain with 2 companies, and a MS of 1024 Bit (k) , with 1024 and 2048 (2*k) keys, apart from the 512 (k/2) IDC encryption keys.

In the First stage, the corresponding company in the chain only can write NSCCA information, because there's no other company before that could insert information for it. For the first company the next steps are performed:

- NSCCA Encryption:Encrypts his CTO (1024-Bit) , adds the IDC codes for the CA and for the other SCM members (512-Bit each)
- RFID Writing: Tag selection and NSCCA output writing (2048 Bit).

Operation	Time (ms)
NSCCA Encrypting	116
RFID Tag selection	203
2048-Bit Writing	2496
Total time	2815

Table 4.3: First stage NSCCA Encryption and writing process

In the second stage, the company the information is addressed to rewrites the tag with his own NSCCA output:

- Reads the information from the previous stage, reading 2048 Bit.
- Decrypts the information to determine if is valid and address for him.
- Performs the NSCCA encryption.

- Writes the NSCCA output to the RFID Tag

Operation	Time (ms)
Tag selected	205
Read data: 2048 Bit	1363
Previous CTO decryption	67
Next CTO Encryption	16
Write data: 3072 Bit	3549
Total time	5200

Table 4.4: Reading, NSCCA decryption, encryption and writing process

Chapter 5

USER MANUAL

A program has been developed implementing the created library. In this section there would be explained all the components and screens, as well as all the functions provided, and how to use them correctly.

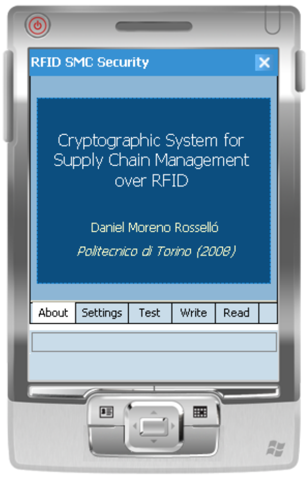
5.1 Installing

At the application development, there have been generated 3 different output files:

- A DLL Library grouping together the NSCCA and Reader classes, and their respective used structures, for future uses in other projects, making it easy to use just by importing them to the project.
- A folder with all the executables, folders and DLLs, that would work only by copying the folder into the device.
- A installation setup in a CAB (cabinet file), in order to make easy the installation in any mobile device.

Notice that the application has been developed to run with the .NET Compact Framework 3.5, which is actually a beta, but would work perfectly with .NET CF 2.0 either. In orther to make it run, first the .NET CF has to be installed on the device.

Also for copying the files needed from the PC to the device, some conversion and accomodation must be done. For that, ActiveSync should be used for Windows XP, while Windows Mobile Device Center (WMDC) for Windows Vista must be used.



Cryptographic System for Supply Chain Management over RFID

Daniel Moreno Rosselló

Politecnico di Torino (2008)

Figure 5.1: Application wellcome screen

5.2 Settings configuration

The program have some screens and methods to set up the parameters needed for the RFID reading/writing and NSCCA operations. This parameters are inserted in the applications by using the specific fields in the "settings" tab, and pressing the save button. This action imports the parameters to the structures, and at the same tame it saves the parameters into an XML file. There are two different configuration subscreens: "Chain" and "RFID Reader".

5.2.1 NSCCA Chain settings

In this screen (Figure 5.2 on page 32) the main parameters for the NSCCA algorithm can be loaded and set. As explained in the previous section, the field contents and the corresponding parameters values are also saved into an XML file. The one where the NSCCA Chain parameters are saved is called by default "ChainSettings.xml", and it is saved in the SETTINGS folder.

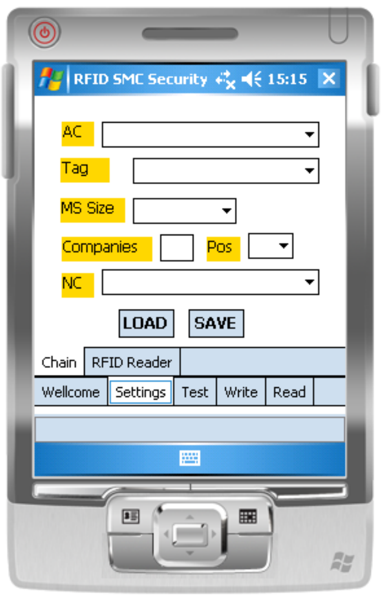


Figure 5.2: Settings screen

The parameters available are:

- AC: Identification string (ID) of the actual company that is running the application.
- Tag: RFID Tag model, i.e: SRIX4K (ST microelectronics)
- MS Size: Size of the memory slot
- Companies: Number of positions in the chain. For example, 3 positions (with many companies of each type as we want).
- Pos: Position of the actual company in the chain. Depending on it, only some functions can be performed. For example, the first company obviusly don't need to run the "read" function, because no one could have previously writen.
- NC: Next company ID, the company who the information is addressed to. That's important in order to encrypt the information with his key, and make it readable for it.

5.2.2 RFID Reader settings

In this screen, the main settings for the interaction with the Reader and reading / writting performance are provided.

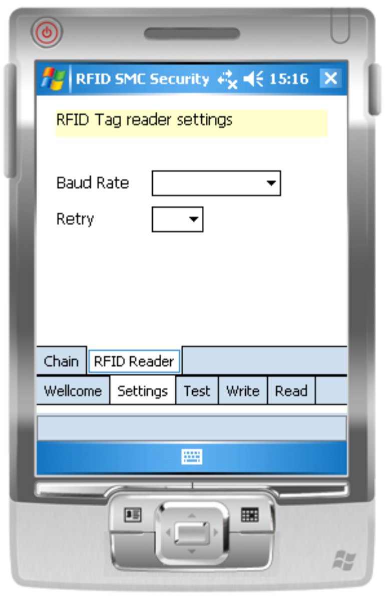


Figure 5.3: Reader settings

- *Baud rate*: Data transfer speed can be selected through this combo box, from the speeds range that the reader accepts.
- *Retry*: Number of maximum retries from a block reading / writing error. After this number of retries have resulted in error, the operation will be aborted.
- *Serial port:* Detect the serial port where the reader can be accessed.

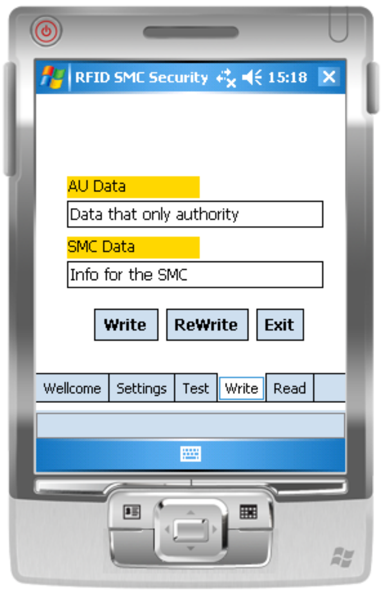
5.3 Writing Information

This screen (Figure 5.4 on page 34) allows selecting the information to be written, and also do the writings. There are two main fields, which are:

- AU Data: The data addressed the CA
- SCM Data: The data addressed to the allowed members of the chain

There is a maximum amount of data to be inserted, depending on the MS size. In this program, this data are just ASCII characters, but if necessary the library is built so that it is possible to insert the data in other formats.

Also there are 2 options for writing information to the tag: *Write*, allowed only for the first member of the chain, because it can't receive any previous data from previous stages, and *Rewrite,* for the rest of companies in the next chain positions. The difference between them is that *Rewrite* also reads the information stored previously in the Tag, decrypts it to check that it is correct, uses the information retrieved in the new frame and writes it to the tag.



AU Data
Data that only authority
SMC Data
Info for the SMC

Figure 5.4: Tag writing screen

5.4 Reading Information

In this screen (Figure 5.5 on page 35) user can perform a reading or an scanning on RFID Tags. The reading involves reading the information from the tag and decrypting it.

When the action is performed, the result is showed on different text fields:

- *Tag ID*: The identification code of the read Tag.
- *IDC*: Identification code of the last company who wrote information in the tag
- *SCM Data*: Data for the permitted companies of the supply chain

There are two basic reading functions provided:

- Pressing the *Read* button, one read is performed on the actual Tag.
- Pressing the *Scan* button, many reads in a row can be done, only passing the reader over each Tag.

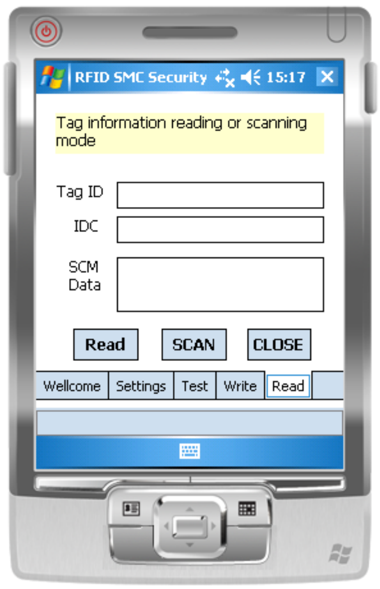


Figure 5.5: Read screen

While reading, many different situations may occur. In case that the information was corrupted, or not addressed to the company actually reading, a message will be shown on screen, and the program will continue working.

- Tag not found: Any RFID Tag is present in the reading area, or it isn't close enough.
- Error during reading: While reading, one or more reading operations have failed, it could possible be because while reading, the tag or the reader have been moved.

Chapter 6

Conclusions

Personally, the realization of this project has helped me in many facets. It helped me to learn and understand RFID technology and practice with different devices. Also helped to improve my cryptography knowledge, and finding new ways of applicating it, as it's the case of this project.

Related to the results obtained from this project, I think NSCCA is a good way of giving security and privacy in Supply Chain Management applications, but it also has actually some drawbacks, due to the limitations in the technical part, that I think will be solved in a nearly future, as the hardware will improve and speed up the processes.

The capacity of the actual RFID tag is not enough to store information for chains that have more than 3-4 players in a row. That should not be a big problem, because storage capacity is growing up in future, but has limited the testing of the project, as only little chains could be implemented in the system.

Another interesting fact is that encryption time is also limiting factor, as the processing time for encrypting and decrypting grows up exponientally with the key length, and that means the achievement of more security only can be done by wasting more time on encryption processes, that may be too longer for systems that require little time on every product, as supply chains are.

Bibliography

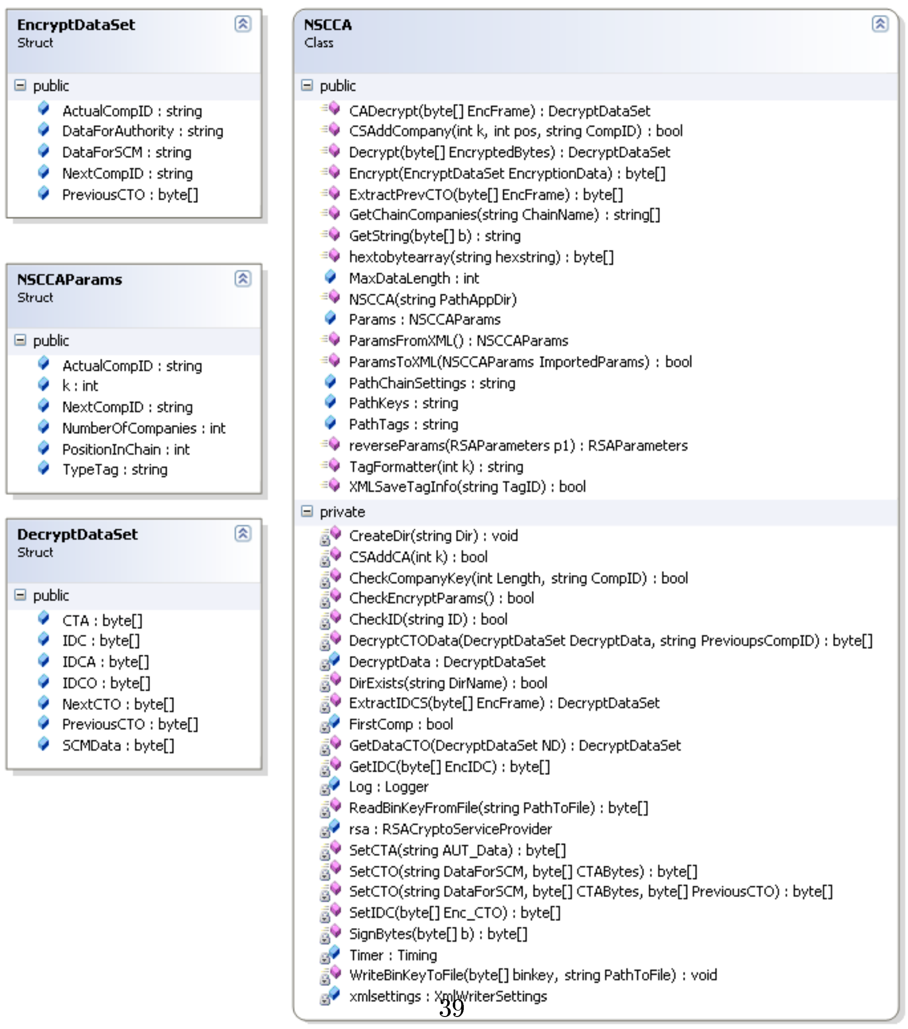
[1] [1] M. Rebaudengo F. Gandino, B. Montrucchio. Secure pervasive information system from producers to end users. 2009.

[2] Klaus Finkenzeller. *RFID HANDBOOK: Fundamentals and Applications in Contactless Smart Cards and Identification.* Wiley, 2003.

Chapter 7

Program class diagrams

7.1 NSCCA Algorithm Class and related structures



EncryptDataSet
Struct

public
- ActualCompID : string
- DataForAuthority : string
- DataForSCM : string
- NextCompID : string
- PreviousCTO : byte[]

NSCCAParams
Struct

public
- ActualCompID : string
- k : int
- NextCompID : string
- NumberOfCompanies : int
- PositionInChain : int
- TypeTag : string

DecryptDataSet
Struct

public
- CTA : byte[]
- IDC : byte[]
- IDCA : byte[]
- IDCO : byte[]
- NextCTO : byte[]
- PreviousCTO : byte[]
- SCMData : byte[]

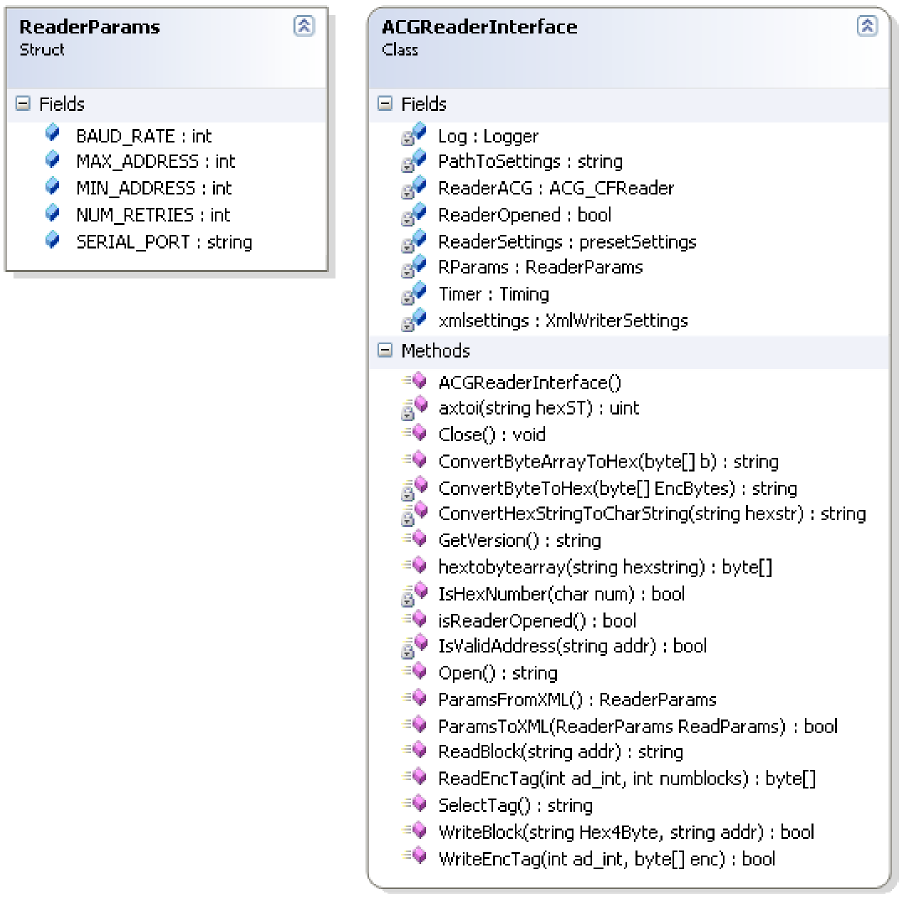
NSCCA
Class

public
- CADecrypt(byte[] EncFrame) : DecryptDataSet
- CSAddCompany(int k, int pos, string CompID) : bool
- Decrypt(byte[] EncryptedBytes) : DecryptDataSet
- Encrypt(EncryptDataSet EncryptionData) : byte[]
- ExtractPrevCTO(byte[] EncFrame) : byte[]
- GetChainCompanies(string ChainName) : string[]
- GetString(byte[] b) : string
- hextobytearray(string hexstring) : byte[]
- MaxDataLength : int
- NSCCA(string PathAppDir)
- Params : NSCCAParams
- ParamsFromXML() : NSCCAParams
- ParamsToXML(NSCCAParams ImportedParams) : bool
- PathChainSettings : string
- PathKeys : string
- PathTags : string
- reverseParams(RSAParameters p1) : RSAParameters
- TagFormatter(int k) : string
- XMLSaveTagInfo(string TagID) : bool

private
- CreateDir(string Dir) : void
- CSAddCA(int k) : bool
- CheckCompanyKey(int Length, string CompID) : bool
- CheckEncryptParams() : bool
- CheckID(string ID) : bool
- DecryptCTOData(DecryptDataSet DecryptData, string PrevioupsCompID) : byte[]
- DecryptData : DecryptDataSet
- DirExists(string DirName) : bool
- ExtractIDCS(byte[] EncFrame) : DecryptDataSet
- FirstComp : bool
- GetDataCTO(DecryptDataSet ND) : DecryptDataSet
- GetIDC(byte[] EncIDC) : byte[]
- Log : Logger
- ReadBinKeyFromFile(string PathToFile) : byte[]
- rsa : RSACryptoServiceProvider
- SetCTA(string AUT_Data) : byte[]
- SetCTO(string DataForSCM, byte[] CTABytes) : byte[]
- SetCTO(string DataForSCM, byte[] CTABytes, byte[] PreviousCTO) : byte[]
- SetIDC(byte[] Enc_CTO) : byte[]
- SignBytes(byte[] b) : byte[]
- Timer : Timing
- WriteBinKeyToFile(byte[] binkey, string PathToFile) : void
- xmlsettings : XmlWriterSettings

Figure 7.1: NSCCA Algorithm Class and structures

7.2 RFID Reader class



ReaderParams
Struct

Fields
- BAUD_RATE : int
- MAX_ADDRESS : int
- MIN_ADDRESS : int
- NUM_RETRIES : int
- SERIAL_PORT : string

ACGReaderInterface
Class

Fields
- Log : Logger
- PathToSettings : string
- ReaderACG : ACG_CFReader
- ReaderOpened : bool
- ReaderSettings : presetSettings
- RParams : ReaderParams
- Timer : Timing
- xmlsettings : XmlWriterSettings

Methods
- ACGReaderInterface()
- axtoi(string hexST) : uint
- Close() : void
- ConvertByteArrayToHex(byte[] b) : string
- ConvertByteToHex(byte[] EncBytes) : string
- ConvertHexStringToCharString(string hexstr) : string
- GetVersion() : string
- hextobytearray(string hexstring) : byte[]
- IsHexNumber(char num) : bool
- isReaderOpened() : bool
- IsValidAddress(string addr) : bool
- Open() : string
- ParamsFromXML() : ReaderParams
- ParamsToXML(ReaderParams ReadParams) : bool
- ReadBlock(string addr) : string
- ReadEncTag(int ad_int, int numblocks) : byte[]
- SelectTag() : string
- WriteBlock(string Hex4Byte, string addr) : bool
- WriteEncTag(int ad_int, byte[] enc) : bool

Figure 7.2: ACG RFID Reader class and structures

RFID SCM Security

1.0

Generated by Doxygen 1.5.7.1

Wed Jan 21 15:40:01 2009

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Namespace Documentation

4.1 ACGReader Namespace Reference

Data Structures

- struct ReaderParams

  Set of parameters needed to configure correctly the Device and the RFID reader connection, as well as reading and writing settings.

- class ACGReaderInterface

  This class permits to interact with the reader to perform read and write operations, as well as implements methods for opening, closing the device and selecting a tag in the coverage area. ///.

4.2 NSCCAlgorithm Namespace Reference

Data Structures

- struct ChainMember

  Defines a company member of the supply chain.

- struct DecryptDataSet

  Groups all the parameters needed to store the NSCCA decryption results.

- struct EncryptDataSet

  Groups all the parameters needed to store the NSCCA decryption inputs.

- struct NSCCAParams

  Stores all the parameters needed by the NSCCA algorithm methods.

- class NSCCA

  The NSCCA class is intended to implement the algorithm functions, such as encryption and encryption.

Chapter 5

Data Structure Documentation

5.1 ACGReader::ACGReaderInterface Class Reference

This class permits to interact with the reader to perform read and write operations, as well as implements methods for opening, closing the device and selecting a tag in the coverage area. ///.

Public Member Functions

- ACGReaderInterface ()

  Initializes a new instance of the ACGReaderInterface class.

- bool isReaderOpened ()

  Determines whether the reader is opened or not.

- void Close ()

  Close the RFID Reader.

- String Open ()

  Open the RFID Reader.

- String SelectTag ()

  Selects a tag situated on the coverage area.

- bool ParamsToXML (ReaderParams ReadParams)

  Save reading parameters to an XML file.

- ReaderParams ParamsFromXML ()

  Loads reading parameters from an XML file.

- String ConvertHexStringToCharString (String hexstr)

  Converts a string of hexadecimal values into a character string.

- String ConvertByteArrayToHex (byte[ ] b)

  Converts a byte array to a string of hexadecimal values.

- bool WriteEncTag (int ad_int, byte[ ] enc)

  Writes an encoding array of bytes into the RFID Tag.

- bool WriteBlock (string Hex4Byte, String addr)

  Reads a 4 Byte block from the Tag.

- String ReadBlock (String addr)

  Reads a block of 4 Bytes from the Tag.

- byte[ ] ReadEncTag (int ad_int, int numblocks)

  Reads information stored in the tag.

- byte[ ] ConvertHexToByteArray (string hexstring)

  Converts an hexadecimal string to byte array.

Private Member Functions

- bool IsValidAddress (String addr)
- uint axtoi (String hexST)
- bool IsHexNumber (char num)

  Checks if a character is a valid hexadecimal character.

Private Attributes

- ACG_CFReader ReaderACG
- ACG_CFReader.presetSettings ReaderSettings
- XmlWriterSettings xmlsettings = new XmlWriterSettings()

  Settings for the xml files.

- ReaderParams RParams = new ReaderParams()
- Timing Timer = new Timing()
- Logger Log
- string PathToSettings

  Path to the settings file.

- bool ReaderOpened = false

  Tells whether the reader is already opened or not.

5.1.1 Detailed Description

This class permits to interact with the reader to perform read and write operations, as well as implements methods for opening, closing the device and selecting a tag in the coverage area. ///.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 ACGReader::ACGReaderInterface::ACGReaderInterface ()

Initializes a new instance of the ACGReaderInterface class.

5.1.3 Member Function Documentation

5.1.3.1 uint ACGReader::ACGReaderInterface::axtoi (String *hexST*) `[private]`

5.1.3.2 void ACGReader::ACGReaderInterface::Close ()

Close the RFID Reader.

5.1.3.3 String ACGReader::ACGReaderInterface::ConvertByteArrayToHex (byte[ ] *b*)

Converts a byte array to a string of hexadecimal values.

Parameters:

b The byte array to convert

Returns:

The string containing the hexadecimal values

5.1.3.4 String AC-GReader::ACGReaderInterface::ConvertHexStringToCharString (String *hexstr*)

Converts a string of hexadecimal values into a character string.

Parameters:

hexstr String with hexadecimal values

Returns:

The string containing the converted values

5.1.3.5 byte [ ] ACGReader::ACGReaderInterface::ConvertHexToByteArray (string *hexstring*)

Converts an hexadecimal string to byte array.

Parameters:

hexstring The hexadecimal string

Returns:

The byte array containing the result

5.1.3.6 bool ACGReader::ACGReaderInterface::IsHexNumber (char *num*) `[private]`

Checks if a character is a valid hexadecimal character.

Parameters:

num Character to test

Returns:

true if is hexadecimal, false if not

5.1.3.7 bool ACGReader::ACGReaderInterface::isReaderOpened ()

Determines whether the reader is opened or not.

Returns:

`true` if the reader is opened; otherwise, `false`.

5.1.3.8 bool ACGReader::ACGReaderInterface::IsValidAddress (String *addr*) `[private]`

5.1.3.9 String ACGReader::ACGReaderInterface::Open ()

Open the RFID Reader.

Returns:

5.1.3.10 ReaderParams ACGReader::ACGReaderInterface::ParamsFromXML ()

Loads reading parameters from an XML file.

Returns:

`true` if the loading succeeds; otherwise, `false`.

5.1.3.11 bool ACGReader::ACGReaderInterface::ParamsToXML (ReaderParams *ReadParams*)

Save reading parameters to an XML file.

Parameters:

ReadParams The read params to be saved

Returns:

`true` if the saving succeeds; otherwise, `false`.

5.1.3.12 String ACGReader::ACGReaderInterface::ReadBlock (String *addr*)

Reads a block of 4 Bytes from the Tag.

Parameters:

addr Address to read

Returns:

String with the hexadecimal value read from the Tag

5.1.3.13 byte [ ] ACGReader::ACGReaderInterface::ReadEncTag (int *ad_int*, int *numblocks*)

Reads information stored in the tag.

Parameters:

ad_int Address to begin reading
numblocks Number of blocks to read

Returns:

Byte array containing the read data

5.1.3.14 String ACGReader::ACGReaderInterface::SelectTag ()

Selects a tag situated on the coverage area.

Returns:

string containing the identification code of the RFID Tag

5.1.3.15 bool ACGReader::ACGReaderInterface::WriteBlock (string *Hex4Byte*, String *addr*)

Reads a 4 Byte block from the Tag.

Parameters:

Hex4Byte The 4-Byte hexadecimal data to write

addr Address to write de data

Returns:

True if ok, false if not

5.1.3.16 bool ACGReader::ACGReaderInterface::WriteEncTag (int *ad_int*, byte[ ] *enc*)

Writes an encoding array of bytes into the RFID Tag.

Parameters:

ad_int Initial address to write

enc Byte array corresponding to the encoding text

Returns:

5.1.4 Field Documentation

5.1.4.1 Logger ACGReader::ACGReaderInterface::Log `[private]`

5.1.4.2 string ACGReader::ACGReaderInterface::PathToSettings `[private]`

Path to the settings file.

5.1.4.3 ACG_CFReader ACGReader::ACGReaderInterface::ReaderACG `[private]`

5.1.4.4 bool ACGReader::ACGReaderInterface::ReaderOpened = false `[private]`

Tells whether the reader is already opened or not.

5.1.4.5 ACG_CFReader.presetSettings ACGReader::ACGReaderInterface::ReaderSettings `[private]`

5.1.4.6 ReaderParams ACGReader::ACGReaderInterface::RParams = new ReaderParams() `[private]`

5.1.4.7 Timing ACGReader::ACGReaderInterface::Timer = new Timing() `[private]`

5.1.4.8 XmlWriterSettings ACGReader::ACGReaderInterface::xmlsettings = new XmlWriterSettings() `[private]`

Settings for the xml files.

The documentation for this class was generated from the following file:

- ACGReaderInterface.cs

5.2 NSCCAlgorithm::ChainMember Struct Reference

Defines a company member of the supply chain.

Data Fields

- int Position

  The position of the company in the chain.

- string ID

  The Identification Code for the company.

5.2.1 Detailed Description

Defines a company member of the supply chain.

5.2.2 Field Documentation

5.2.2.1 string NSCCAlgorithm::ChainMember::ID

The Identification Code for the company.

5.2.2.2 int NSCCAlgorithm::ChainMember::Position

The position of the company in the chain.

The documentation for this struct was generated from the following file:

- NSCCA.cs

5.3 NSCCAlgorithm::DecryptDataSet Struct Reference

Groups all the parameters needed to store the NSCCA decryption results.

Data Fields

- byte[ ] CipherTextA

  CTA structure for the authority.

- byte[ ] DataForSCM

  Data for the allowed supply chain members.

- byte[ ] IDCA

  Company Identification code encrypted for the authority.

- byte[ ] IDCO

  Company Identification code encrypted for the next member in the chain.

- byte[ ] IDC

  Company Identification code not encrypted.

- byte[ ] PreviousCTO

  CTO from the previous stage in the chain.

- byte[ ] NextCTO

  CTO for the next company in the chain.

5.3.1 Detailed Description

Groups all the parameters needed to store the NSCCA decryption results.

5.3.2 Field Documentation

5.3.2.1 byte [ ] NSCCAlgorithm::DecryptDataSet::CipherTextA

CTA structure for the authority.

5.3.2.2 byte [ ] NSCCAlgorithm::DecryptDataSet::DataForSCM

Data for the allowed supply chain members.

5.3.2.3 byte [ ] NSCCAlgorithm::DecryptDataSet::IDC

Company Identification code not encrypted.

5.3.2.4 byte [ ] NSCCAlgorithm::DecryptDataSet::IDCA

Company Identification code encrypted for the authority.

5.3.2.5 byte [ ] NSCCAlgorithm::DecryptDataSet::IDCO

Company Identification code encrypted for the next member in the chain.

5.3.2.6 byte [ ] NSCCAlgorithm::DecryptDataSet::NextCTO

CTO for the next company in the chain.

5.3.2.7 byte [ ] NSCCAlgorithm::DecryptDataSet::PreviousCTO

CTO from the previous stage in the chain.

The documentation for this struct was generated from the following file:

- NSCCA.cs

5.4 NSCCAlgorithm::EncryptDataSet Struct Reference

Groups all the parameters needed to store the NSCCA decryption inputs.

Data Fields

- byte[ ] PreviousCTO
  CypherText CTO from the previous chain stage.
- string DataForAuthority
  Data addressed to the certification authority (CA).
- string DataForSCM
  Data addressed to the allowed members of the chain.
- string ActualCompID
  Actual company identification code.
- string NextCompID
  Next company identification code.

5.4.1 Detailed Description

Groups all the parameters needed to store the NSCCA decryption inputs.

5.4.2 Field Documentation

5.4.2.1 string NSCCAlgorithm::EncryptDataSet::ActualCompID

Actual company identification code.

5.4.2.2 string NSCCAlgorithm::EncryptDataSet::DataForAuthority

Data addressed to the certification authority (CA).

5.4.2.3 string NSCCAlgorithm::EncryptDataSet::DataForSCM

Data addressed to the allowed members of the chain.

5.4.2.4 string NSCCAlgorithm::EncryptDataSet::NextCompID

Next company identification code.

5.4.2.5 byte [ ] NSCCAlgorithm::EncryptDataSet::PreviousCTO

CypherText CTO from the previous chain stage.

The documentation for this struct was generated from the following file:

- NSCCA.cs

5.5 NSCCAlgorithm::NSCCA Class Reference

The NSCCA class is intended to implement the algorithm functions, such as encryption and encryption.

Public Member Functions

- NSCCA (string PathAppDir)

  Initializes a new instance of the NSCCA class.

- RSAParameters ReverseParams (RSAParameters original)

  Reverses the RSA parameters, exchanging private and public keys, and recalculates other derived parameters needed for the encryption and decryption functions.

- bool ParamsToXML (NSCCAParams ImportedParams)

  Saves all the parameters needed to an XML File, usually called "ChainSettings.xml" situated in the configuration files path.

- NSCCAParams ParamsFromXML ()

  Loads all the parameters needed from an XML File, usually called "ChainSettings.xml" situated in the configuration files path.

- bool SaveTagToXML (string TagID)

  Save read information from RFID Tag into an XML tag.

- DecryptDataSet CADecrypt (byte[ ] EncFrame)

  CAs the decrypt.

- byte[ ] Encrypt (EncryptDataSet EncryptionData)

  Performs the Nested Supply Chain Cyphering Algorithm (NSCCA) Encryption and returns a Byte array with the encoded information, and length (Position + 1) $*$ k Bits.

- DecryptDataSet Decrypt (byte[ ] EncryptedBytes)

  Performs the Nested Supply Chain Cyphering Algorithm (NSCCA) decryption process of a byte array of length (Position + 1) $*$ k Bits and returns a structure with the decoded information fields contained in it.

- bool AddCompany (int k, int pos, string CompID)

  Adds a company to the chain, creating his own IDC and Encoding Keysets.

- void CreateNSCCASystem (int k, ChainMember[ ] CM)

  Creates the NSCCA system.

- string GetString (byte[ ] b)

  Gets the string from a byte array.

- byte[ ] ExtractPrevCTO (byte[ ] EncFrame)

This method extracts all the CTO from the previous frame without decrypting it.

- string[ ] GetChainCompanies (string ChainName)
- byte[ ] hextobytearray (string hexstring)

  Convert an hexadecimal string into a byte array.

Data Fields

- NSCCAParams Params = new NSCCAParams()
- string PathChainSettings

  Path to the Chain settings file.

- string PathKeys

  Path to the Key folder.

- string PathTags

  Path to the log file.

Private Member Functions

- void CreateDir (string Dir)
- byte[ ] SetCTA (string AUT_Data)

  with the information for the Certification Authority, and encrypt it with the CA Public key

- byte[ ] SetCTO (string DataForSCM, byte[ ] CTABytes, byte[ ] PreviousCTO)

  Gets the actual CTA from the SetCTA method, containing k/2 Bits, the data addressed to the SMC, and the CTO from the previus stage, and produces an output Byte array of (position) ∗ k Bits For example, if k = 512 and position 2 in the chain: PrevCTO: 1024 Bytes (position-1) ∗k Bits CTA: 512 Bytes (k/2) Bits Returned array: 2048 Bytes (position) ∗ k Bits.

- byte[ ] SetCTO (string DataForSCM, byte[ ] CTABytes)

  Gets the actual CTA from the SetCTA method, containing k/2 Bits, the data addressed to the SMC, and the CTO from the previus stage, and produces an output Byte array of (position) ∗ k Bits For example, if k = 512 and position 2 in the chain: PrevCTO: 1024 Bytes (position-1) ∗k Bits CTA: 512 Bytes (k/2) Bits Returned array: 2048 Bytes (position) ∗ k Bits

  .

- byte[ ] SetIDC (byte[ ] Enc_CTO)

  Constructs the IDCA and IDCO encrypting the IDC number of the actual company, concatenates them to the previous CTO from the SetCTO method and returns the resulting byte array.

- byte[ ] SignBytes (byte[ ] b)

  Makes a "sign" from a byte array, by extracting the content of some of the bytes.

- DecryptDataSet ExtractIDCS (byte[ ] EncFrame)
- byte[ ] GetIDC (byte[ ] EncIDC)

  Gets the Identity code of the last company who encrypted the information (IDCO field) in order to know who it was, load his public keys and decrypt the remaining information.

- byte[ ] DecryptCTOData (DecryptDataSet DecryptData, string PrevioupsCompID)

  Decrypts the CTO data extracted previously to a Decrypt structure.

- DecryptDataSet GetDataCTO (DecryptDataSet ND)

  Gets the CTO data.

- bool AddCertAuthority (int k)

  Adds the certification authority to the chain, by creating its key pairs.

Static Private Member Functions

- static bool DirExists (string dirName)

  Check if the specified directory exists, and if not, creates it.

- static void WriteBinKeyToFile (byte[ ] binkey, string PathToFile)

  Writes the bin key to file.

- static byte[ ] ReadBinKeyFromFile (string PathToFile)

  Reads the bin key from file.

Private Attributes

- DecryptDataSet DecryptData = new DecryptDataSet()
- Logger Log
- Timing Timer = new Timing()

  A Timer to calculate the time spent to perform some actions.

- RSACryptoServiceProvider rsa = new RSACryptoServiceProvider()
- XmlWriterSettings xmlsettings = new XmlWriterSettings()

  xml file settings, to define some properties on the file format, like identation, line spacing, etc

5.5.1 Detailed Description

The NSCCA class is intended to implement the algorithm functions, such as encryption and encryption.

Author:

Daniel Moreno Roselló

Version:

4.0

Date:

January 2009

5.5.2 Constructor & Destructor Documentation

5.5.2.1 NSCCAlgorithm::NSCCA::NSCCA (string *PathAppDir*)

Initializes a new instance of the NSCCA class.

Parameters:

PathAppDir	The path to the application running, used later to determine some paths to configuration files, key storing folders, etc

5.5.3 Member Function Documentation

5.5.3.1 bool NSCCAlgorithm::NSCCA::AddCertAuthority (int *k*) `[private]`

Adds the certification authority to the chain, by creating its key pairs.

Parameters:

k	The memory slot size

Returns:

`true` if created successfully; otherwise, `false`.

5.5.3.2 bool NSCCAlgorithm::NSCCA::AddCompany (int *k*, int *pos*, string *CompID*)

Adds a company to the chain, creating his own IDC and Encoding Keysets.

Parameters:

k

pos Position of the company in the chain

CompID Identification of the company

Returns:

Whether the creation was successfull or not

5.5.3.3 DecryptDataSet NSCCAlgorithm::NSCCA::CADecrypt (byte[ ] *EncFrame*)

CAs the decrypt.

Parameters:

EncFrame Byte array containing the encrypted information

Returns:

DecryptDataSet with the decryption results

5.5.3.4 void NSCCAlgorithm::NSCCA::CreateDir (string *Dir*) `[private]`

5.5.3.5 void NSCCAlgorithm::NSCCA::CreateNSCCASystem (int *k*, ChainMember[ ] *CM*)

Creates the NSCCA system.

Parameters:

k The size of the memory slot

CM The ChainMember to add to the chain

5.5.3.6 DecryptDataSet NSCCAlgorithm::NSCCA::Decrypt (byte[ ] *EncryptedBytes*)

Performs the Nested Supply Chain Cyphering Algorithm (NSCCA) decryption process of a byte array of length (Position + 1) $*$ k Bits and returns a structure with the decoded information fields contained in it.

Parameters:

EncryptedBytes The encrypted bytes.

Returns:

An NDFrame structure contained the data obtained by the method

5.5.3.7 byte [ ] NSCCAlgorithm::NSCCA::DecryptCTOData (DecryptDataSet *DecryptData*, string *PrevioupsCompID*) `[private]`

Decrypts the CTO data extracted previously to a Decrypt structure.

Parameters:

DecryptData The data to be decrypted

PrevioupsCompID The previoups company Identification Code

Returns:

5.5.3.8 static bool NSCCAlgorithm::NSCCA::DirExists (string *dirName*) `[static, private]`

Check if the specified directory exists, and if not, creates it.

Parameters:

dirName Name of the directory

Returns:

False if there was an exception, true if not

5.5.3.9 byte [ ] NSCCAlgorithm::NSCCA::Encrypt (EncryptDataSet *EncryptionData*)

Performs the Nested Supply Chain Cyphering Algorithm (NSCCA) Encryption and returns a Byte array with the encoded information, and length (Position + 1) $*$ k Bits.

Parameters:

EncryptionData The structure with the data to be encrypted

Returns:

Byte array contained the result data

5.5.3.10 DecryptDataSet NSCCAlgorithm::NSCCA::ExtractIDCS (byte[ ] *EncFrame*) `[private]`

5.5.3.11 byte [ ] NSCCAlgorithm::NSCCA::ExtractPrevCTO (byte[ ] *EncFrame*)

This method extracts all the CTO from the previous frame without decrypting it.

Parameters:

EncFrame

Returns:

5.5.3.12 string [ ] NSCCAlgorithm::NSCCA::GetChainCompanies (string *ChainName*)

5.5.3.13 DecryptDataSet NSCCAlgorithm::NSCCA::GetDataCTO (DecryptDataSet *ND*) `[private]`

Gets the CTO data.

Parameters:

ND The ND.

Returns:

5.5.3.14 byte [ ] NSCCAlgorithm::NSCCA::GetIDC (byte[ ] *EncIDC*) `[private]`

Gets the Identity code of the last company who encrypted the information (IDCO field) in order to know who it was, load his public keys and decrypt the remaining information.

Parameters:

EncIDC Byte array containing the encrypted IDCO Field

Returns:

A

5.5.3.15 string NSCCAlgorithm::NSCCA::GetString (byte[ ] *b*)

Gets the string from a byte array.

Parameters:

b The byte array

Returns:

The resulting string

5.5.3.16 byte [ ] NSCCAlgorithm::NSCCA::hextobytearray (string *hexstring*)

Convert an hexadecimal string into a byte array.

Parameters:

hexstring String containing the hexadecimal data

Returns:

A byte array with the information converted

5.5.3.17 NSCCAParams NSCCAlgorithm::NSCCA::ParamsFromXML ()

Loads all the parameters needed from an XML File, usually called "ChainSettings.xml" situated in the configuration files path.

5.5.3.18 bool NSCCAlgorithm::NSCCA::ParamsToXML (NSCCAParams *ImportedParams*)

Saves all the parameters needed to an XML File, usually called "ChainSettings.xml" situated in the configuration files path.

Parameters:

ImportedParams The imported params.

Returns:

5.5.3.19 static byte [ ] NSCCAlgorithm::NSCCA::ReadBinKeyFromFile (string *PathToFile*) `[static, private]`

Reads the bin key from file.

Parameters:

PathToFile The path to the file where the key is stored

Returns:

Byte array with the read key

5.5.3.20 RSAParameters NSCCAlgorithm::NSCCA::ReverseParams (RSAParameters *original*)

Reverses the RSA parameters, exchanging private and public keys, and recalculates other derived parameters needed for the encryption and decryption functions.

Parameters:

original The original RSA Parameters

Returns:

RSAParameters object containing reversed parameters

5.5.3.21 bool NSCCAlgorithm::NSCCA::SaveTagToXML (string *TagID*)

Save read information from RFID Tag into an XML tag.

Parameters:

TagID RFID Tag's identification number

Returns:

Boolean value indicating if the saving was successful

5.5.3.22 byte [ ] NSCCAlgorithm::NSCCA::SetCTA (string *AUT_Data*) `[private]`

with the information for the Certification Authority, and encrypt it with the CA Public key

Parameters:

AUT_Data Data for the Certification authority

Returns:

Byte array containing the CTA

5.5.3.23 byte [ ] NSCCAlgorithm::NSCCA::SetCTO (string *DataForSCM*, byte[ ] *CTABytes*) `[private]`

Gets the actual CTA from the SetCTA method, containing k/2 Bits, the data addressed to the SMC, and the CTO from the previus stage, and produces an output Byte array of (position) $*$ k Bits For example, if k = 512 and position 2 in the chain: PrevCTO: 1024 Bytes (position-1) $*$k Bits CTA: 512 Bytes (k/2) Bits Returned array: 2048 Bytes (position) $*$ k Bits

.

Parameters:

DataForSCM Data for other suppy management chain members

CTABytes CTA obtained from the setCTA method

Returns:

Byte array containing the CTO

5.5.3.24 byte [ ] NSCCAlgorithm::NSCCA::SetCTO (string *DataForSCM*, byte[ ] *CTABytes*, byte[ ] *PreviousCTO*) `[private]`

Gets the actual CTA from the SetCTA method, containing k/2 Bits, the data addressed to the SMC, and the CTO from the previus stage, and produces an output Byte array of (position) $*$ k Bits For example, if k = 512 and position 2 in the chain: PrevCTO: 1024 Bytes (position-1) $*$k Bits CTA: 512 Bytes (k/2) Bits Returned array: 2048 Bytes (position) $*$ k Bits.

Parameters:

DataForSCM Data for other suppy management chain members

CTABytes CTA obtained from the setCTA method

PreviousCTO CTO from the previous stage

Exceptions:

ArgumentException If *CTOPrev* or *Enc_CTA* are longer than permitted

Returns:

byte[] containing the CTO

5.5.3.25 byte [ ] NSCCAlgorithm::NSCCA::SetIDC (byte[ ] *Enc_CTO*) `[private]`

Constructs the IDCA and IDCO encrypting the IDC number of the actual company, concatenates them to the previous CTO from the SetCTO method and returns the resulting byte array.

Parameters:

Enc_CTO Previous stage CTO

Returns:

Byte array containing the final frame

For example, if k = 512 and position 2: Previous CTO: 2048 Bytes ((position)$*$ k) IDCA and IDCO: 512 Bytes (k/2) Returned array: 3072 Bytes ((position+1)$*$ k)

First we generate IDCA Bytes

Then we generate IDCO Bytes

5.5.3.26 byte [ ] NSCCAlgorithm::NSCCA::SignBytes (byte[ ] *b*) `[private]`

Makes a "sign" from a byte array, by extracting the content of some of the bytes.

Parameters:

b The byte array to be signed

Returns:

A byte array containing the signature

5.5.3.27 static void NSCCAlgorithm::NSCCA::WriteBinKeyToFile (byte[ ] *binkey*, string *PathToFile*) `[static, private]`

Writes the bin key to file.

Parameters:

binkey The binkey.

PathToFile The path.

5.5.4 Field Documentation

5.5.4.1 DecryptDataSet NSCCAlgorithm::NSCCA::DecryptData = new DecryptDataSet() `[private]`

5.5.4.2 Logger NSCCAlgorithm::NSCCA::Log `[private]`

5.5.4.3 NSCCAParams NSCCAlgorithm::NSCCA::Params = new NSCCAParams()

5.5.4.4 string NSCCAlgorithm::NSCCA::PathChainSettings

Path to the Chain settings file.

5.5.4.5 string NSCCAlgorithm::NSCCA::PathKeys

Path to the Key folder.

5.5.4.6 string NSCCAlgorithm::NSCCA::PathTags

Path to the log file.

5.5.4.7 RSACryptoServiceProvider NSCCAlgorithm::NSCCA::rsa = new RSACryptoServiceProvider() `[private]`

5.5.4.8 Timing NSCCAlgorithm::NSCCA::Timer = new Timing() `[private]`

A Timer to calculate the time spent to perform some actions.

5.5.4.9 XmlWriterSettings NSCCAlgorithm::NSCCA::xmlsettings = new XmlWriterSettings() `[private]`

xml file settings, to define some properties on the file format, like identation, line spacing, etc

The documentation for this class was generated from the following file:

- NSCCA.cs

5.6 NSCCAlgorithm::NSCCAParams Struct Reference

Stores all the parameters needed by the NSCCA algorithm methods.

Properties

- string ActualCompID `[get, set]`

  Gets or sets the actual comp ID.

- string NextCompID `[get, set]`

  Gets or sets the next comp ID.

- string TypeTag `[get, set]`

  Gets or sets the type tag.

- int K `[get, set]`

  Gets or sets the K.

- int PositionInChain `[get, set]`

  Gets or sets the position in chain.

- int NumberOfCompanies `[get, set]`

  Gets or sets the number of companies.

- int MaxDataLength `[get, set]`

  Gets the length of the max data.

- bool FirstComp `[get, set]`

  Gets a value indicating whether is the first comp in a chain.

Private Attributes

- string actualCompID

  Actual company identification code.

- string nextCompID

  Next company identification code.

- string typeTag

  Type of RFID Tag used.

- int k

  Size of the memory slot.

- int positionInChain

  Position of the actual company within the chain.

- int numberOfCompanies

  Number of companies in a row of the chain.

- int maxDataLength

  Maximum amount of data that we can insert, either SMC or CA Data.

- bool firstComp

  Indicates if the actual company is the first in the chain or not.

5.6.1 Detailed Description

Stores all the parameters needed by the NSCCA algorithm methods.

5.6.2 Field Documentation

5.6.2.1 string NSCCAlgorithm::NSCCAParams::actualCompID `[private]`

Actual company identification code.

5.6.2.2 bool NSCCAlgorithm::NSCCAParams::firstComp `[private]`

Indicates if the actual company is the first in the chain or not.

5.6.2.3 int NSCCAlgorithm::NSCCAParams::k `[private]`

Size of the memory slot.

5.6.2.4 int NSCCAlgorithm::NSCCAParams::maxDataLength `[private]`

Maximum amount of data that we can insert, either SMC or CA Data.

5.6.2.5 string NSCCAlgorithm::NSCCAParams::nextCompID `[private]`

Next company identification code.

5.6.2.6 int NSCCAlgorithm::NSCCAParams::numberOfCompanies `[private]`

Number of companies in a row of the chain.

5.6.2.7 int NSCCAlgorithm::NSCCAParams::positionInChain `[private]`

Position of the actual company within the chain.

5.6.2.8 string NSCCAlgorithm::NSCCAParams::typeTag `[private]`

Type of RFID Tag used.

5.6.3 Property Documentation

5.6.3.1 string NSCCAlgorithm::NSCCAParams::ActualCompID `[get, set]`

Gets or sets the actual comp ID.

The actual comp ID.

5.6.3.2 bool NSCCAlgorithm::NSCCAParams::FirstComp `[get, set]`

Gets a value indicating whether is the first comp in a chain.

`true` if it is; otherwise, `false`.

5.6.3.3 int NSCCAlgorithm::NSCCAParams::K `[get, set]`

Gets or sets the K.

The K.

5.6.3.4 int NSCCAlgorithm::NSCCAParams::MaxDataLength `[get, set]`

Gets the length of the max data.

The length of the max data.

5.6.3.5 string NSCCAlgorithm::NSCCAParams::NextCompID `[get, set]`

Gets or sets the next comp ID.

The next comp ID.

5.6.3.6 int NSCCAlgorithm::NSCCAParams::NumberOfCompanies `[get, set]`

Gets or sets the number of companies.

The number of companies.

5.6.3.7 int NSCCAlgorithm::NSCCAParams::PositionInChain `[get, set]`

Gets or sets the position in chain.

The position in chain.

5.6.3.8 string NSCCAlgorithm::NSCCAParams::TypeTag `[get, set]`

Gets or sets the type tag.

The type tag.

The documentation for this struct was generated from the following file:

- NSCCA.cs

5.7 ACGReader::ReaderParams Struct Reference

Set of parameters needed to configure correctly the Device and the RFID reader connection, as well as reading and writing settings.

Data Fields

- int NumRetries

  Maximum number of reading / writing retries in case of error.

- int MinAddres

  Minimum writable address in the RFID tag memory.

- int MaxAddress

  Maximum writable address in the RFID tag memory.

- int BaudRate

  Bauds per second of tag writting / reading speed.

- string SerialPort

  Serial port where the reader can be accessed.

5.7.1 Detailed Description

Set of parameters needed to configure correctly the Device and the RFID reader connection, as well as reading and writing settings.

5.7.2 Field Documentation

5.7.2.1 int ACGReader::ReaderParams::BaudRate

Bauds per second of tag writting / reading speed.

5.7.2.2 int ACGReader::ReaderParams::MaxAddress

Maximum writable address in the RFID tag memory.

5.7.2.3 int ACGReader::ReaderParams::MinAddres

Minimum writable address in the RFID tag memory.

5.7.2.4 int ACGReader::ReaderParams::NumRetries

Maximum number of reading / writing retries in case of error.

5.7.2.5 string ACGReader::ReaderParams::SerialPort

Serial port where the reader can be accessed.

The documentation for this struct was generated from the following file:

- ACGReaderInterface.cs

Chapter 6

File Documentation

6.1 ACGReaderInterface.cs File Reference

Data Structures

- struct ACGReader::ReaderParams

  Set of parameters needed to configure correctly the Device and the RFID reader connection, as well as reading and writing settings.

- class ACGReader::ACGReaderInterface

  This class permits to interact with the reader to perform read and write operations, as well as implements methods for opening, closing the device and selecting a tag in the coverage area. ///.

Namespaces

- namespace ACGReader

6.2 NSCCA.cs File Reference

Data Structures

- struct NSCCAlgorithm::ChainMember

  Defines a company member of the supply chain.

- struct NSCCAlgorithm::DecryptDataSet

  Groups all the parameters needed to store the NSCCA decryption results.

- struct NSCCAlgorithm::EncryptDataSet

  Groups all the parameters needed to store the NSCCA decryption inputs.

- struct NSCCAlgorithm::NSCCAParams

  Stores all the parameters needed by the NSCCA algorithm methods.

- class NSCCAlgorithm::NSCCA

  The NSCCA class is intended to implement the algorithm functions, such as encryption and encryption.

Namespaces

- namespace NSCCAlgorithm

Index